ABERYSTWYTH UNIVERSITY

DOCTORATE OF PHILOSOPHY

G4313: COMPUTER SCIENCE

Neuromodulatory supervised learning

James Finnis (jcf12@aber.ac.uk)

7th February 2020

Supervisors

Dr. Frédéric Labrosse
Dr. Christine Zarges

Examiners

Prof. Susan Stepney
University of York

Dr. Patrícia Amâncio Vargas
Heriot-Watt University

This thesis is submitted in candidature for a Doctorate of Philosophy in Computer Science (G4313)

Final Copy

Declarations

Candidate Name: James Finnis
Word count of thesis: 88069

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signature: _____ Date: _____
James Finnis

Statement 1

This thesis is the result of my own investigations, except where otherwise stated, and has not been altered by correction or editing services.

Sources are acknowledged by endnotes giving explicit references. A full bibliography is appended.

Signature: _____ Date: _____
James Finnis

Statement 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signature: _____ Date: _____
James Finnis

Abstract

Biological nervous systems exist in an environment in which chemicals — neuromodulators — modify their function, notably the strengths of the neuron connections. This allows global modulation of the entire network by a parameter which, in artificial neuroendocrine systems, is typically an artificial "hormone" decoupled from the network itself. Such systems have shown promise in implementing adaptive behaviour, particularly homeostasis. However, current implementations typically have their parameters set by hand, use pre-trained networks modulating only the output layer, or use complex and limited learning rules or evolutionary algorithms.

The network described in this thesis is a simplified Neal/Timmis artificial neuroendocrine system with a single hidden layer, named UESMANN[1]. It is trained using a modified back-propagation of errors, with two sets of examples: one for each extremum of the modulator. Thus it uses a supervised learning algorithm.

The work is divided into four parts, each consisting of a number of chapters. Part I (p. 3) consists of an introduction and outline of the thesis including the motivation behind it and a methodology, and an extensive literature review with a particular focus on sub-symbolic biologically-inspired adaptive systems.

This is followed by Part II (p. 61), which introduces and describes the UESMANN network and performs an Monte Carlo analysis of random UESMANN nodes and networks performing pairings of boolean functions. This shows that a subset of boolean pairings is possible in a single node, but that all possible pairings of binary boolean functions can be performed with two hidden nodes: the same as the minimum required for a single boolean in a standard multi-layer perceptron. The modified back-propagation algorithm is then developed and tested on the boolean pairings with good results, with the resulting networks analysed in depth.

Part III (p. 163) shows that the network is capable of learning to transition between two line recognition functions (vertical and horizontal) and to transition between nominal and alternate labellings of the MNIST handwriting recognition database, with comparable performance with two other modulatory paradigms.

In Part IV (p. 229) the network performed comparably with other techniques in a robot homeostasis task both in simulation and reality, showing particularly interesting transitional behaviour.

The final part, Part V (p. 319), is a conclusion to the thesis as a whole with reference to the initial hypothesis and research questions, and describes further work which should be undertaken.

[1]Uniformly Excitatory Switching Modulatory Artificial Neural Network — the acronym is pronounced "WES-mun"

Contents

List of Figures

List of Tables

List of Algorithms

Glossary

action A single, discrete event performed by an agent. It is different from a behaviour, which is a group of actions to be performed when a set of criteria apply. What constitutes an action is determined by the level of abstraction: actions at a higher level may consist of engaging behaviours or multiple actions at a lower level [31]. 18

action selection The problem of which behaviour or action to select, given a known repertoire of such actions. Slightly more formally, "the problem of motor resource allocation an autonomous agent is faced with, when attempting to achieve its long-term objectives." (Girard et al. [100].) 7, 18

adaptation A change in the internal structure or parameters of a system which is adaptive, the act of changing in an adaptive manner, or the degree to which such a system is adapted to its environment. 5, 15, 56

adaptive Of a cybernetic system, able to respond to environmental changes in an analogous manner to evolutionary adaptation or physiological homeostasis in biology. Refers to changes which help the system survive, or perform "better" (according to some metric determined by the system designer). Alternatively, descriptive of such a change. xiii, 4, 56, 57, 230

adaptivity Of a cybernetic system, the property of being adaptive, being able to change in response to environmental changes. Not to be confused with the "adaptability," which in this work means "capable of being adapted [by an outside agency]." 4, 15, 56

arg min Notation for "argument of the minimum of a function." Thus

$$\arg\min_{x} f(x)$$

specifies the value of x for which $f(x)$ has its minimum value: the "argument of the minimum of $f(x)$ with respect to x." 93

artificial immune system A system inspired by the processing paradigms inherent in the vertebrate immune system. xix, 53, 54

artificial endocrine system An artificial model of a biological endocrine system, or a system inspired by the operation of a biological endocrine system. xix, 44

artificial neural network A system inspired by the animal nervous system, made up of a large number of similar (usually identical) units operating in parallel with interconnections between them. xiv, xix

back-propagation Short for back-propagation of errors, a gradient descent technique used to train feedforward neural networks. Works well with networks containing only a few hidden layers, but requires enhancements to work with deeper or recurrent networks. xv, 7, 26, 28, 29, 30, 31, 32, 34, 35, 36, 47, 51, 52, 91, 92, 320

batching A form of gradient descent — a supervised learning technique — in which the solution is updated from the mean of all gradients in the error surface detected at the current solution, given all the examples. Contrast with stochastic gradient descent. xvii, 88

behaviour Strictly, a set of actions operating over time which make up a coherent group, typically in response to internal or external stimuli. Sometimes a behaviour may loosely be considered a "high level" action, in that it can be selected by an "action selection" method. This thesis uses "behaviour" with both this sense and also with the common English meaning "the general way a system acts under different stimuli." 3, 18, 20

biological endocrine system The endocrine system of organisms, as opposed to artificial endocrine systems. Endocrine systems release and receive hormones, chemicals which circulate through the bloodstream and decay over time. xix, 42

bio-mimetic Descriptive of techniques which attempt to imitate biological mechanisms in some detail, with the assumption that it is the details of the mechanism which will generate the required performance. Contrast with biologically inspired. xiv, 56, 325

biologically inspired Inspired by biological mechanisms as a loose foundation, but not imitative of those mechanisms in detail. Contrast with bio-mimetic. xiv, 5, 54, 56, 325

connectionism A form of artificial intelligence in which connections between similar or identical nodes forms a massively parallel network. The archetypal connectionist paradigm is the artificial neural network (ANN). 13, 16

continuous time recurrent neural network A dynamical system of differential equations, each describing a "leaky integrator" unit which has an activation which decays over time [15, 16]. Frequently used in evolutionary robotics. xv, xix

evolutionary robotics An approach to generating adaptive behaviour in robots typically using a genetic algorithm (GA) to evolve a continuous time recurrent neural network (CTRNN) to control the robot, though other approaches also fall within the field [212, 288]. Some of these also generate physical parameters, such as body plans. xv, xix

genetic algorithm The most commonly used evolutionary algorithm, based on work by Holland [125]. xv, xix, 53, 56

homeostasis An adaptive response to environmental changes which keeps an "essential variable" or set of such variables within a fixed range. , xiii, 11, 13, 17, 40, 49, 56, 230, 231, 233, 257, 281, 282, 307, 321, 330

hormone In biology, a chemical released by an endocrine gland into the bloodstream which modifies the behaviour of a distant organ, and decays over time. They can be seen as sending a signal to that organ. In the artificial systems described in this thesis this term is often synonymous with neuromodulator, because artificial "hormones" often modulate artificial neural networks. xiv, 7

hyperparameter A parameter used to control the training of a system, as opposed to those parameters trained by the system. For example, hidden node count, learning rate, and initial weight range are hyperparameters for back-propagation training of a multilayer perceptron, while the parameters are the weights and biases generated by that algorithm. 89, 99, 156, 230

iteration (UESMANN) In UESMANN back-propagation training with stochastic gradient descent, an iteration consists of an entire pass through the shuffled example set. See Algorithm 1 on page 97. 96, 103, 165

multilayer perceptron A feedforward neural network consisting of nodes similar to those of Rosenblatt's perceptron, but typically modified to use a sigmoid activation function to make it amenable to training by gradient descent. xix, 7, 27

Neal/Timmis system A form of neuromodulatory artificial endocrine system in which a set of hormones modulate the weights in a multilayer perceptron in a multiplicative manner. Described more fully in Sec. 2.5.6. xix, 7

neuromodulation In biology, the process by which the function of neurons can be modified by chemicals. These chemicals may be released far from the modulated neuron, and may persist for some time. They are often also neurotransmitters. xv, 4, 7, 44, 45

one-hot encoding An encoding often used for the output of multi-class classifier systems in which there is one output for each class, and in the examples the output whose index corresponds to the label is "true" (e.g. 1) while the other outputs are zero [204, p. 215]. See Sec. 7.1 (p. 198). 165, 198

ontogenetic Describes a form of learning which takes place in a single individual throughout its operation. The opposite of phylogenetic. xvi, 22, 55

ordinary differential equation A differential equation which contains only ordinary derivatives: derivatives of single-variable functions with respect to that variable. Compare with partial derivative equations, in which partial derivatives can appear: derivatives with respect to one variable of a multi-variable function, with the others held constant. xix, 42

pair presentation In UESMANN back-propagation training using stochastic gradient descent, each example consists of an input and the required outputs for modulator (hormone) levels $h = 0$ and $h = 1$. An update step involves presenting the $h = 0$ example, calculating the gradient and applying it, followed by the same procedure for the $h = 1$ example. This is referred to as a "pair presentation." See Algorithm 1 on page 97: the pair presentation is the inner loop of the main part of the algorithm. 96, 165

phylogenetic Describes a form of learning which takes place in an entire population across multiple generations. The opposite of ontogenetic. xvi, 22, 55, 56

reality gap The discrepancy between a simulated environment and the complex and unpredictable real world [140], which often causes robotic controllers designed or trained in simulation to fail when run on a real robot. 5, 41, 240

reinforcement learning A form of learning in which the agent learns a strategy (known as a *policy*) to select actions which will maximise the cumulative sum

over time of a single scalar value, the *reward*. For example, in a game playing agent, the cumulative reward to be maximised might be the score. The agent would learn how to perform actions given each situation which would maximise the score at the end of the game. 56

stochastic gradient descent A supervised learning technique, particularly in neural networks, which updates the solution from each example individually using the gradient in the error surface detected at the current solution, given the example. Constrast with batching. xix, 96

sub-symbolic A style of "artificial intelligence" which does not involve the designer explicitly building models of the world, in order to reduce the assumptions made in so doing. 5, 56

temporal difference A reinforcement learning technique which uses various techniques (such as dynamic programming using the Bellman equation) to update estimates of the outcomes of potential actions. xix, 56

ultrastability "The ability of a system to change its internal organization or structure in response to environmental conditions that threaten to disturb a desired behavior or value of an essential variable. The changes such systems are capable of are qualitative in the sense of changing the mode of interaction with an environment in steps or jumps, not along a continuum, and they are purposeful because such systems seek a behavior that is disturbance defying. Ultrastability is stability of a logical level higher than the stability to which a system converges without change of its internal organization or structure." [156]. 13

Uniformly Excitatory Switching Modulatory Artificial Neural Network (UESMANN) The network under investigation in this thesis: a neuromodulatory network in which the modulator (or "hormone") functions in a uniform manner (i.e. does the same thing to all weights), is always excitatory, and can be trained to switch (or rather, smoothly transition) between two behaviours. The acronym is pronounced "WES-mun." xix, 5

Acronyms

AES artificial endocrine system 44, 45, 46, 47, 49, 50, 52, 323

AIS artificial immune system 53, 54, 55, 56

ANN artificial neural network xiv, 22, 27, 29, 30, 38, 40, 50, 52, 53

BES biological endocrine system 7, 42

CTRNN continuous time recurrent neural network xv, 7, 38, 39, 40, 41, 42, 43

ER evolutionary robotics xv

GA genetic algorithm xv, 31, 53, 56

MLP multilayer perceptron 7, 27, 28, 29, 47, 61, 79, 123, 322

NTS Neal/Timmis system 7, 8, 46, 49, 50, 51, 52, 319, 322

ODE ordinary differential equation 42, 43

PRNG pseudo-random number generator. 63, 78

SGD stochastic gradient descent 88, 96

TD temporal difference 56

UESMANN Uniformly Excitatory Switching Modulatory Artificial Neural Network 5, 61

Symbols

This section describes the symbols used in this thesis.

Boolean functions

Binary boolean functions (boolean operators) are dealt with in several ways. In diagrams, because of the difficulty of creating LaTeX symbols in PDF images in some software packages, the common English names of the functions are used along with the "!" symbol for negation. As is the standard practice, "!" binds with a high precedence. In the body of the work the appropriate mathematical symbol is generally used, although the English name may sometimes be used for clarity. Truth tables of the 16 binary boolean functions are given below in Table 1 with their English names and the symbols used in this document.

Function transitions

This thesis deals frequently with transitions between binary boolean functions under the influence of a modulator h. For example, we might perform the function $x \vee y$ when $h = 0$ and change to $x \wedge y$ when $h = 1$. These are notated using the right arrow $\rightarrow$, so the example would be written as

$$x \vee y \rightarrow x \wedge y$$

These transitions are often referred to in the text as "pairings."

Common parameters and metrics

Table 1: The 16 binary boolean functions expressed as $f(x, y)$ where $x, y \in \{0, 1\}$, with the English abbreviation and symbols used in this work.

$f(0,0)$	$f(0,1)$	$f(1,0)$	$f(1,1)$	English abbreviation	Symbol
0	0	0	0	F	F
0	0	0	1	and	$\wedge$
0	0	1	0	x and !y	$x \wedge \neg y$
0	0	1	1	x	x
0	1	0	0	!x and y	$\neg x \wedge y$
0	1	0	1	y	y
0	1	1	0	xor	$x \oplus y$
0	1	1	1	or	$x \vee y$
1	0	0	0	nor	$\neg(x \vee y)$
1	0	0	1	xnor	$\neg(x \oplus y)$
1	0	1	0	!y	$\neg y$
1	0	1	1	x or !y	$x \vee \neg y$
1	1	0	0	!x	$\neg x$
1	1	0	1	!x or y	$\neg x \vee y$
1	1	1	0	nand	$\neg(x \wedge y)$
1	1	1	1	T	T

Table 2: Symbols used for common parameters and metrics

Symbol	Meaning	See section
η	The learning rate in the back-propagation algorithm	4.2.1
h	The modulation parameter for modulatory networks	2.5.6.1 and 4.2.1
ϕ	The Matthews correlation coefficient, used as a metric in classification problems	5.1
ϕ_{min}	The minimum ϕ over the two modulator levels h, used as a metric in classification problems	5.1

Acknowledgements

Firstly, I must thank Fred and Christine, my supervisors through the final half of this thesis; and Mark Neal who supervised initially before moving on. Each of you provided valuable advice, inspiration and guidance without which I would have been wandering in the dark (and disappearing down rabbit holes). I would also like to thank Hannah Dee and Roger Boyle for their kind advice and support, my former employer John Jones-Steele for many years of on-the-job mentoring, and Joanna Bryson for putting me right on a small but important matter.

More personally, I would like to thank my family, particularly my mother and late father, for their love and support over the years, and for encouraging my peculiar fascination with "harnessing the lightning to teach sand to think."

Heartfelt thanks are also due to the Robert Jones and Agnes Hunt Hospital in Oswestry, and the Christie Hospital in Manchester, without whom I would not be here to write these words; and to the Computer Science Department and the University as a whole for being understanding and helpful through a difficult time.

Finally, I would like to thank my wonderful wife Catrin for all her help, encouragement and love through a tumultuous few years.

Part I

Introduction and Literature Review

Chapter 1

Introduction

> Ce n'est pas ici ma doctrine, c'est mon étude; et n'est pas la leçon d'autrui, c'est la mienne; et ne me doit-on pourtant savoir mauvais gré si je la communique; ce qui me sert aussi, par accident, peut servir à un autre. Au demeurant, je ne gâte rien, je n'use que du mien; et si je fais le fou, c'est à mes dépens, et sans l'intérêt de personne; car c'est en folie, qui meurt en moi, qui n'a point de suite[1].
>
> *Essais de Montaigne, Chapitre XXV*

Many modern robotic systems must adapt to their environment, changing their behaviour according to the situation, as well as reacting to the environment in the short term [38].

Consider a robot which must explore as much of the environment as possible with limited resources which can be replenished at a set location (a problem considered later in this thesis). Let us assume that this robot is "reactive" (as opposed to "deliberative")[2]: such a robot does not make plans in advance but simply reacts to the stimuli with which it is presented in the moment. Thus it must react by

[1] *"What I write here is not my teaching, but my study; it is not a lesson for others, but for me. And yet it should not be held against me if I publish what I write. What is useful to me may also by accident be useful to another. Moreover, I am not spoiling anything, I am only using what is mine. And if I play the fool, it is at my expense and without harm to anyone. For it is a folly that will die with me, and will have no consequences."*

[2] The "reactive/deliberative" terminology is taken from Arkin's work on the AuRA system, which is a hybrid system with components of both types (see Sec. 2.2.3, p. 20). Deliberative systems, which use symbolic reasoning on models of the system and environment (see Sec. 2.1.1, p. 14), are also a component of "cognitive robotics" [52], which is concerned with constructing a cognitive, reasoning architecture with which a robot can learn to achieve complex goals in a complex environment.

avoiding obstacles, but it must also adapt as its resources are expended[3], changing its behaviour to return to its "home."

This might be achieved using two levels of reactive operation. At the lower level are multiple behaviours such as "avoid obstacles" or "head for home". Considered separately, each behaviour does not account for environmental changes and typically runs atemporally (i.e. in the instant, with no time-dependent effects). At the higher level, an adaptive system acts over longer timescales to change how the low-level system reacts as the environment changes, by selecting (see Sec. 2.2, p. 18) from the lower-level behaviours. An example of such a system is "Sozzy", a hormone-driven autonomous vacuum cleaner [306] which uses the subsumption architecture of Brooks [33]. It should be noted, however, that many systems (notably those built on the subsumption architecture) use elements of "emergent behaviour", where desired behaviours result from non-obvious interactions between lower level behaviours [32].

Robots controlled by artificial neural networks — systems inspired by the biological nervous system (see Sec. 2.3, p. 22 and particularly Sec. 2.3.9.1, p. 37) — may benefit from biological inspiration in how they adapt. Neurons act over very short timescales and are thus "reactive" (in the sense used above). To change the behaviour of the animal over longer timescales the behaviour of the neurons must be modified. In biology, one way this is achieved is through *neuromodulators*: chemicals flowing through the space around the cells which modify the behaviour of large groups of neurons. These may be released some distance away and may persist for some time. In a simulated system this can create a division between temporal dynamics, handled by the simulated chemistry of the neuromodulator (typically a simple release/decay model[4]), and the atemporal behaviour of the network itself.

This thesis describes a novel neural network architecture inspired by neuromodulation which is capable of transitioning between two behaviours in a single set of parameters (i.e. weights and biases) learned by the well-established technique of the back-propagation of errors. The transition is controlled by a global modulator value, which affects all the weights in the same way and can be continuously varied between 0 and 1.

While much of the thesis does not deal with the temporal dynamics of neuromodulation, the robot problem in Part IV contains a simple temporal model in the form of a simulated battery, which charges and discharges over time.

[3] In this system the resource level, while part of the robot, is separate from the robot's control system and so is considered part of its environment.

[4] See Sec. 2.5.6 (p. 47) for a description of the temporal model used by the Neal/Timmis system as an example.

1.1 Outline of the thesis

This introduction forms part of Part I of the thesis. Section 1.3 gives the motivation and methodology for the work: briefly, most existing artificial neuromodulatory systems are designed by hand and are not trained, while those which are trained are often complex. Our goal is to find a simple system which can be trained (albeit in a supervised way, rather than the reinforcement method which would be required for true learned adaptivity), and investigate the performance and behaviour of such a system.

The next chapter, Chapter 2, is an extensive literature review which discusses the theoretical background to the concept of "adaptation" (and "intelligent behaviour" in general), with is a particular focus on sub-symbolic and biologically inspired (bio-inspired) techniques. We then discuss neural networks and artificial endocrine systems: systems inspired by the hormonal systems of animals, which have come to include artificial neuromodulatory systems.

Part II introduces the network architecture under discussion, which has been named Uniformly Excitatory Switching Modulatory Artificial Neural Network (UESMANN)[5]. As well as introducing the network and deriving the training method, this part of the thesis studies the architecture's behaviour in a single node performing pairings of binary boolean functions. We then look at the minimal network for such pairings, with the finding that all possible pairings may be expressed in the same number of parameters (i.e. weights and biases) which are required for any single function. Comparisons are made to two other simple neuromodulatory methods: linear interpolation between the outputs of two separately trained multi-layer perceptrons, and adding an extra modulator input to such a network.

In Part III the network is evaluated in some more complex classification problems. The first is a binary problem: the network is trained to recognise horizontal lines in images when the modulator $h = 0$, transitioning to recognising vertical lines when $h = 1$. The second problem is a multi-class problem: recognising handwritten digits from the well-known MNIST database.

Part IV deals with the robot problem described above: how well does UESMANN perform (in comparison with the two methods described above) in a task which requires balancing exploratory behaviour with returning to "recharge" a simulated battery from time to time? Experiments were performed using two simulators of differing complexity and on a real robot, to study how well the different networks crossed the "reality gap" from simulator to robot.

[5]Pronounced "WES-mun."

1.2 Prior publications

- Some of the work in Chapter 4 (on training UESMANN on boolean pairings with back-propagation) and Part III (on classification problems) has been published previously in [88].
- Some of the work in Part IV (on a robotic control problem) has been published previously in [85].

1.3 Motivation

The term "artificial endocrine system[6]" currently has a very wide scope, covering any technique inspired by features of the biological endocrine system (BES). These features include leaky integrators[7], global messaging with selectivity on reception[8], and diffusion through space[9]. If we define an AES as any system which uses such features we could include systems such as CTRNNs[10] and GasNets[11], which each use several.

Therefore in this work the term "artificial endocrine systems" will cover only those techniques directly inspired by the BES. Even these systems are wildly disparate — from the "emotional" systems of Yamamoto [306] and Cañamero [46], in which hormones are leaky integrators driving subsumption architectures or action selection mechanisms, to systems like the Neal/Timmis system (NTS) and GasNets where the hormone modulates the behaviour of a neural network.

The system developed in the rest of this thesis is motivated primarily by the Neal/Timmis AES (see Sec. 2.5.6, p. 47), and is intended to be a component of such a system. That is, it is an artificial neural network which is capable of being modulated by a single AES hormone (or neuromodulator). This thesis is agnostic on the matter of how the hormone is generated — indeed, it is possible that the network described here may be used without a "hormone" in the sense described in the previous sections. In practice it is likely to be some form of leaky integrator with a saturation facility as described in Sec. 2.5.6.2.

In a robotic system the network essentially performs action selection (see Sec. 2.2, p. 18), given that it is trained with two different behaviours between which the modulator selects. However, the modulator is continuous and thus the behaviour actually performed may be a blend or compromise between the two trained behaviours.

The majority of the systems described above have difficulties in training — that is, difficulties in learning parameters which are adaptive to their environment. Some, such as the NTS, opt for either hand-design[12] while others use some form of evolutionary algorithm. If the Neal/Timmis modulation model — which is already fairly simple — were simplified further, reducing the number of parameters down to those for a multilayer perceptron, it may be amenable to training techniques used for an

[6] see Sec. 2.5 (p. 42)

[7] As used in the Continuous Time Recurrent Neural Network, see Sec. 2.3.9.1 (p. 37)

[8] as used in the Neal/Timmis system, see Sec. 2.5.6 (p. 47)

[9] As used in GasNets, see Sec. 2.3.10 (p. 39)

[10] See Sec. 2.3.9.1 (p. 37)

[11] See Sec. 2.3.10 (p. 39)

[12] Although the networks may initially be trained with back-propagation, sensitivities must be either uniform or hand-designed.

multilayer perceptron (MLP), such as back-propagation of errors. While not ideal from the point of view of learning adaptivity in a wider sense, developing a gradient descent method to train Neal/Timmis style networks would provide several benefits.

Firstly, a system which could be trained to modulate smoothly between two given functions would demonstrate that the basic approach of the NTS modulatory scheme is more versatile than the current literature suggests. These tend to modulate the entire network uniformly in an attempt to provide more or less of a single behaviour [278, 243, 209], often resulting in undesirable non-linearities which require more selective modulation. Often the decision is made to modulate only the output layer [243] or calculate neuronal hormone sensitivities by hand [118]. There are trivial solutions to this problem, which we shall investigate for comparison: linear interpolation between two sets of weights, and linear interpolation between the outputs of two different networks. However, while these reproduce their "parent" network outputs completely at the extrema, they have different transition characteristics from each other and possibly from other modulatory approaches.

Secondly, such a simplified system may have practical uses. As noted above, existing systems are either hand-designed, hand-assembled from pre-trained parts, or trained using evolutionary techniques. Online training is not possible, or may be prohibitively expensive, notwithstanding the speculative "Breach" system of Neal and Timmis [208]. However, a system which is trained from examples of desirable behaviour but can generalise from those examples may be as useful, practically speaking, as part of a system which learns to be adaptive at a higher level. This supervised system may behave more predictably than a system generated by the minimisation of a single cost function relating to the system's performance as a whole through reinforcement learning, or it may simply be that it is possible to find or generate plenty of suitable training examples, such that a supervised method is more appropriate. Such a system is still useful in building adaptivity — consider the hexapod walking robot of Henley and Barnes [118] (see Sec. 2.5.6.5, p. 51) or even the lower levels of the "Breach" system.

Finally, if such a simplified system can be built it may have implications for both neuromodulatory AES and neurobiology. In both systems, we might intuitively believe that if a modulator has a uniformly excitatory action on a group of neurons which produce a given behaviour, then more modulator will produce more of that behaviour. If an artificial network is able to learn two different behaviours with such a uniform and simple modulatory regime, this may form a simple "existence proof" that such a system might exist in biology — that two qualitatively different learned behaviours can result from a simple, globally excitatory modulator. Even if this is

well understood in biological circles, it is counterintuitive and a simple model which demonstrates it may be of value.

1.4 Hypothesis and research questions

Our hypothesis is:

> *"An extremely simple neuromodulatory network can be trained by supervised learning to perform two different functions at the modulator extrema, smoothly modulating between these two functions, with comparable performance to other methods at the extrema in multiple problem domains."*

Thus our research questions are:

1. Is it possible to build an extremely simple neuroendocrine system whose response to a neuromodulator (a global parameter) can be trained such that the network performs qualitatively different functions at different modulator levels?

2. How simple is it possible to make such a system, so that we can make the fewest possible prior assumptions about it?

3. What engineering advantages might such a system have?

4. If such a simple system can be built, can we learn from it anything about the nature and evolution of the biological systems which inspired it?

1.5 Methodology

As has been discussed, the first research question will be answered by developing the UESMANN network. This is a reformulation of the Neal/Timmis system of Sec. 2.5.6 designed to operate with a single hormone and global sensitivity to that hormone, such that it may be trained by gradient descent, and is discussed in Chapter 3. We then proceed to analyse the performance and behaviour of the system in increasingly complex problems. We start with boolean algebra to understand the basic characteristics of the individual network nodes, before developing the training algorithm. We then move on to more complex classification problems, analysing the convergence performance to understand the nature of the error surface being traversed. We also analyse the transition behaviour: how the function being

performed changes as the modulator is varied between the extrema. We compare both these properties with those of alternative modulation methods (see Sec. 4.5.1, p. 122). Finally, we compare the performance and behaviour of UESMANN with the other methods in a moderately complex regression/control problem.

Initial studies of the ability of UESMANN to represent multiple functions were made with respect to the boolean functions, by both analysis and Monte Carlo simulations of 3-node networks (the minimum number of nodes required for a network to learn any binary boolean function). The transitional behaviours of the network — how the function changes as the modulator moves from 0 to 1 — are compared with other methods for a small subset of the 256 possible pairings of binary boolean functions in Sec. 3.1. Sec. 3.2 briefly investigates variants of the UESMANN modulation method, comparing the solution spaces of the boolean pairings with those for UESMANN proper.

In Chapter 4 a training algorithm for UESMANN is presented and compared with two of the other methods of Sec. 3.2: output blending and h-as-input (weight blending is discounted — at the end-points it is identical to output blending, and at intermediate values of the modulator it produces "nonsense" networks due to competing conventions, as shall be shown). This is done for simple boolean functions, with classification problems of increasing complexity dealt with in the following chapters. A very simple training method was deliberately chosen: stochastic gradient descent with no form of batching, no momentum and no other enhancements. This is to determine a baseline performance for further improvement in future work.

The chapter then covers how training UESMANN networks with back-propagation compares with the other modulatory methods on learning pairings of boolean functions, and analyses the clusters of solutions found in order to gain insights into the underlying principles behind their operation, such as the natures of the error surfaces the networks traverse as they learn.

Part III deals with more complex classification problems: recognising horizontal and vertical lines in images (Chapter 6), and recognising handwritten digits from the MNIST data set [168] (Chapter 7). The convergence performance of UESMANN is compared with h-as-input and output blending, to expose differences in the complexity of the error surfaces traversed. For the line recognition experiments, the resulting solutions are analysed to gain insight into how UESMANN and the other network types modulate between the two end-points. This is not done in the handwriting recognition case because the complexity of the resulting patterns makes such an analysis intractable. In both experiments the performances of the different network types are compared over several different learning rates and hidden node

counts, primarily using the Matthews correlation coefficient in preference to simple accuracy for reasons which are explained in Sec. 5.1.

Part IV deals with applying this network to a robotic environment, both under simulation and on a physical robot. Because the primary application for the system is foreseen to be adaptive behaviour, particularly homeostatic behaviour (as this is the most important role for biological endocrine systems), the task chosen is a simple homeostatic exploring and recharging task (see Sec. 8.1 for a full description). The task was also chosen so that the hormone value varies across the [0,1] range, rather than remaining at the extrema, to highlight any differences in the transitional behaviour. Because the network is trained from a large number of continuously-valued sensor input and motor output examples for each behaviour this is also a test of the network's effectiveness in a regression problem.

Training data was generated using simple rule-based controllers in a simple simulated environment. A number of networks were trained for each network type, each using different training data: there is a stochastic element to training in that each simulated robot generating the data randomly changes direction and position from time to time. This mitigates against problems from a single set of training data, although not entirely; and ideally multiple networks should have been trained for each data set, but available resources did not permit this. The performances of the networks are shown for the simple simulator using a new metric which combines survival time, distance variation and edge-weighted distance travelled, in order to score highly robots which range widely and survive well (see Sec. 8.1.2.4). Details of the simple simulator and its use in both training and evaluation are given in Chapter 9.1.

The best network of each type from the simple simulator experiments is carried forward into Chapter 10, in which they are tested using the Gazebo robot simulator and on a real robot. Because the number of runs was limited by time, only quantitative analyses could be performed. However, some simple qualitative analyses are presented.

Chapter 2

Bio-inspired intelligent behaviour

This literature review will deal with the problem of artificial intelligence as applied to robotics, looking at cybernetic notions of adaptivity, neural networks, and extensions thereof such as artificial neuroendocrine systems. It will deal primarily with these sub-symbolic connectionist systems, but some approaches which use explicit symbolic models of the world are also discussed. The intention is to begin with a broad focus on definitions of intelligent behaviour and adaptivity, narrowing the scope to sub-symbolic connectionist systems as applied to achieving adaptive behaviour.

2.1 "Intelligent behaviour" and adaptivity

Many robotic systems are designed to perform a well-defined task in a structured environment, for example a factory, where it is possible for the system to have a precise, predictable model of its environment, and where changes within that environment can be kept to a minimum [20, 185]. Sensors can be designed and calibrated for the domains known to be relevant in that environment, and the effects of actuators can be modelled fairly accurately.

However, an increasing number of systems are being deployed in much more complex and unstructured environments, such as homes or the surfaces of distant planets [145, 186]. To continue performing their stated goals (or perform them better), such systems need to be able to adapt to changes in the environment. This typically involves *homeostasis*, changing configuration or behaviour so that certain "essential variables" are kept within range [14]. These changes are typically quantitative, and take the form of negative feedback control. At a higher level, systems may also be *ultrastable* (to use Ashby's terminology [14]), making qualitative changes in their behaviour in order to maintain a failing homeostasis. Ashby describes ultrastability as a core aspect of biological behaviour, and constructed an electromechanical

homeostat which attempted to maintain a set of variables at a fixed level, reconfiguring itself randomly until it was able to do so.

Later, Maturana and Varela developed the idea of *autopoiesis*: an autopoietic system is one in which the essential variables are those which maintain the cohesion of the system and its ultrastable state [187]. For Maturana and Varela, this is a working definition of life. If such a system becomes maladapted, it will always attempt to adjust in order to maintain its own ability to so adjust. For Maturana and Varela, and also Ashby, the highest level of ultrastability is cognition — Maturana and Varela argue that furthermore all autopoiesis, all life, is cognition.

Thus in order for a system to be *adapted* to its environment, it must be performing homeostasis within that environment. In order for it be be *adaptive* within a changing environment, it must change its homeostatic mechanism to remain adapted when the environment changes; that is, it must exhibit some form of ultrastability [65, Sec. 4].

A system in a complex environment will require some form of adaptive behaviour — homeostasis or ultrastability — in order to achieve its goals. If an essential variable goes out of range, simple homeostasis may resolve the problem. If this is insufficient, a ultrastable change in the behaviour may be required at a higher level.

In this kind of environment the goals may be simply stated (e.g. collect data, maintain battery charge) but are often considerably more complex than they initially appear and may contradict each other. For example it is possible to maximise battery life by doing nothing, but no data will be collected; and while we can maximise data collection by constant exploration, the battery will become depleted. Thus, the problem of control becomes a complex, possibly multi-objective optimisation problem operating on uncertain information.

2.1.1 Symbolic and sub-symbolic solutions

2.1.1.1 Symbolic reasoning

Most early solutions to this problem follow a symbolic route: construct a model of the system and its environment, and logically deduce which action to perform. This is sometimes described as the Physical Symbol Systems Hypothesis approach, after Newell and Simon:

> A physical symbol system has the necessary and sufficient means for general intelligent action [210].

In such a system, the sensor percepts are converted into symbolic data. For example, there may be a symbol “ball” and a symbol “chair”, and a symbol representing the

relationship "on top of". The model might contain another symbol made up of these symbols, representing the fact that "a ball is on top of a chair." This has some advantages, a key one being that the decisions of such a system are predictable and transparent. However, over the last half century a number of problems have been described, such as the "frame problem" [189], the "symbol grounding problem" [252, 112] and the "problem of situatedness" [269]. All these problems can be viewed as part of the "problem of *a prioris*" [292]. This problem occurs whenever a system is built which relies to any degree on the *a priori* specification of a world model. Such an approach risks creating a system whose symbols are inadequately grounded in the world, which mixes levels of domain ontology and which in order to be accurate must represent the world with such a high granularity (due to the frame problem) that the cognitive load of performing inference on the model prohibits real-time operation.

An influential paper by Di Paolo [65] argues that mooted solutions to the problem of adaptivity do not go far enough: that we need to think beyond merely grounding our symbols in a closed sensorimotor loop and should return to Ashbeian and autopoietic thinking. Adding modular solutions for "motivations" and even "emotions" do nothing to create systems which are like animals; rather they create systems which behave superficially like animals in a rather brittle fashion. He argues that another approach is needed to achieve true adaptivity. While his actual suggestion — a system of constructing and preserving "habits" of behaviour — may be worth developing, the real strength of the work is the clear statement of the problems involved in conceptualising "adaptation" and "adaptivity."

In spite of these problems, symbolic or representational AI has enjoyed many successes — AIs can beat grand masters at chess [115], solve complex mathematical problems [113, 191] and interpret natural language reasonably well (typically by using machine learning techniques on vast text corpora, often incorporating non-symbolic subsystems [45]). Many aspects of what is considered to be "intelligent behaviour" have been simulated, and these simulations perform scientifically and commercially useful functions.

2.1.1.2 Sub-symbolic AI

During the rise of symbolic AI, a few dissenters called for alternative approaches. Hubert Dreyfus, basing his work on the existential phenomenology of Heidegger and Merleau-Ponty, noted failures of AI in fields such as pattern recognition and problem solving (with particular relevance to the frame problem), and concluded that computers "must have bodies in order to be intelligent" [71]. Dreyfus was calling

for what is now known as a *situated* approach to artificial intelligence. He decries a tendency towards "associationism", which he defines as the belief that "thinking must be analysable into simple determinate operations" [70, p.48] despite (at the time)

> "mounting evidence in both experimental psychology and the artificial intelligence field itself that, although machines do, people do not perform intelligent tasks by simple determinate steps." [70, p.49].

Dreyfus' associationism is quite a close match to the modern "symbolic reasoning."

Even before some of the great advances in symbolic AI, several strands of thought existed outside the symbolic camp, processing data without converting it to and from symbols for manipulation. We can call all of these "sub-symbolic" after Smolensky [257, p. 7][1]. In contrast to explicitly modelling the world (i.e. the autonomous agent's internal and external environment) and manipulating symbols which represent items within it, sub-symbolic techniques eschew explicit representation in favour of processing sensor data directly. Many of these — linear classifiers and support vector machines, Bayesian systems, etc. — are based on mathematical techniques, far from biological inspiration and with little direct relevance to adaptivity, although like many sub-symbolic techniques they come under the broad umbrella of "machine learning" [27]. Those models which are relevant to us are artificial immune systems, which we will leave for the moment, and explicitly connectionist models.

"Connectionism" implies computation performed by large groups of extremely simple nodes, inspired by the operation of the animal nervous system [95]. Most such systems are not remotely biologically accurate: in many cases our knowledge of biology has moved on since the models were developed, while in others the biological accuracy has been sacrificed for engineering expediency. However, they have had remarkable success, particularly in supervised learning problems. The system discussed in this thesis is a supervised connectionist system, and connectionism will be discussed later.

2.1.1.3 Behaviour-based robotics

In the late '80s and early '90s, strong advances were made in a situated, sub-symbolic — but not connectionist — approach to adaptivity through work in *behaviour-based*

[1] McHale [193] views "sub-symbolic" in Smolensky's work as a rather mistaken defence of the connectionist stance: Smolensky sees connectionism as a paradigm which processes the elements of symbols, rather than symbols directly — implying that the system is perhaps generating and processing symbols, but indirectly. However, it is a useful term: it embraces connectionism as well as architectures such as subsumption, support vector machines and Bayesian inference techniques.

robotics [9] by Rodney Brooks and others, who believed that building "intelligence" should start at the level of the simpler animals, rather than at the human level, thus leading to building agents adapted to their environments for survival. This work, sometimes known as *nouvelle AI*[2], signalled a schism in the AI community [193]: Brooks criticised the mainstream paradigm as not being physically grounded [35], and proposed an architecture of low-level modules whose emergent behaviour in interaction with a complex physical environment could be exploited. This he named the *subsumption architecture*, because higher-level behavioural modules subsume lower-level modules [33]. For example, "wander" and "avoid objects" modules might be subsumed into an "explore" module. A key philosophy is that "the world is its own best model" [35]: rather than building complex models which represent the world and reacting to them, an agent should react to the world directly. Brooks denied that symbolic representation was necessary for apparently intelligent behaviour, believing that such behaviour could result from "a collection of competing behaviours" [38].

In many respects, Brooks and his colleagues were continuing the work of the early cyberneticist William Grey Walter. His simple analogue *Machina speculatrix*, more commonly known today as the "tortoise", demonstrated homeostasis with complex emergent behaviour due to environmental and self interactions [105]. Later work by Braitenberg (but preceding Brooks) would explore a taxonomy of similar "vehicles" [30], vividly pointing out the emergent complexity in their extremely simple behaviours:

> And I am sure you will feel that their motives and tastes are much too varied and intricate to be understood by the observer. These vehicles, you will say, are governed by INSTINCTS of various sorts and, alas, we just don't know how Nature manages to embody instincts into a piece of brain.
>
> You forget, of course, that we have ourselves designed these vehicles. [30, p. 17]

Brooks' systems, while being made up of many interconnected modules, are not connectionist: the modules are not homogeneous as they tend to be in an artificial neural network [38, p.155], and do not change their connections dynamically[3]. Work by Randall Beer and Dave Cliff developed a new conceptual framework which is explicitly connectionist: *computational neuroethology*. This is "concerned with the

[2]Both from "new" and after the French style of cookery, which advocated minimalism and a return to simplicity.

[3]Notwithstanding the ability of one module to suppress or inhibit another.

computational modelling of the neural basis of animal behaviour" [54, 53]. Despite this definition, the modelling used is typically not a rigorous neural model.

In robotics, a fruitful way of thinking about the problem of intelligent behaviour has been to consider which action to perform next at any given time: an approach called "action selection." This has generally been approached symbolically, since actions are predetermined and thus symbolic in nature. Because the network described in this thesis can be seen as related to action selection (although it smoothly interpolates between trained actions, rather than selects one), and because it can also drive or be driven by systems working in that paradigm, action selection will be discussed further in the next section.

2.2 Action selection

In its simplest form, the central problem of intelligent behaviour can be reduced to finding what action or behaviour[4] will maximise the system's positive outcome measures and minimise the negative measures. Brom and Bryson describe this in simple terms:

> "... the most basic problem of intelligent systems: *what to do next*." [31]

Thus one way of abstracting this optimisation problem is to work in terms of what action should be performed at any given instant [254, 31]. If we move down this route, the key questions are:

- What is being selected?
- How is it being selected?

"Action selection" as a problem statement deals with anything which answers these questions, and so theoretically encompasses many — if not all — forms of embodied systems. Another statement of the problem from Girard et al. [100] is "the problem of motor resource allocation an autonomous agent is faced with, when attempting to achieve its long-term objectives."

We can consider the solutions to this problem as falling on a spectrum depending on the level of modelling of the environment performed. At one end are "deliberative" systems with a "Sense→Plan→Act" loop, working on a model of the world to

[4]Here, an "action" is a single, discrete event caused by the system in response to some internal or external stimulus, while a "behaviour" is a coherent set of actions or tendency for actions to occur elicited by such a stimulus.

select optimal actions which fit into an overall plan; while at the other there are completely dynamic "reactive" models in which there are no discrete "actions" *per se*, but multiple continuous processes running in parallel to produce the desired behaviour as an emergent property [254]: connectionist systems fall into this latter category[5]. Bryson [40] provides a useful history of the different strands of thought across this spectrum. As such, much of the history of wider "action selection" reiterates the history of AI as a whole: a Hegelian dialectic with symbolic/deliberative systems as thesis, connectionist/reactive systems as antithesis, and hybrid/behaviourist systems as a synthesis.

As an area of solutions, however, what we might term "classical action selection" tends to denote a particular approach involving algorithms to select from a known set of actions, given stimuli and varying amounts of state. This differs from approaches which emphasise the role of actions as emergent behaviour from many simple processes: while those elementary behaviours may be explicitly coded into the system, as they are in Brooks' subsumption system and its descendants, the "actions" selected emerge from the interplay between the behaviours[6].

2.2.1 Hierarchical action selection

The study of action selection in animals, beginning with the hierarchical action selection mechanisms observed by Lorenz [175, 176] (cited in [284]) and Tinbergen [280] has inspired many artificial constructions. Tinbergen's model consists of an acyclic directed graph of nodes through which some notional "activation" flows, with the graph inputs being fundamental internal and external stimuli and the outputs being "consummatory" nodes which generate behaviour. Nodes pass their activation to other nodes if an "innate releasing mechanism" is active, which in turn is activated by a weighted sum of external stimuli exceeding a threshold. This scheme superficially resembles some simple models of neural networks, as we shall see. Much later, an influential work by Minsky [196] on cognition postulated the existence of many cognitive "agents" partially arranged in hierarchies (agents have

[5]Note that this does not imply that connectionist systems cannot deliberate — the brain, upon which most connectionist systems are modelled, is clearly capable of building models and reasoning on them to perform the "Sense→Plan→Act" loop. However, any apparent deliberation in a connectionist system arises as an emergent property from the architecture of reactive components designed by the experimenter. As such, the connectionist approach is closer to the reactive than to the deliberative paradigm: the deliberation is not engineered in, and each component reacts in a simple manner to its external stimuli (and optionally its internal state).

[6]Here, "action" refers to an entity selected by action selection, while "behaviour" is used in Brooks' sense of "a module in a subsumption architecture."

"sub-agents") and partially interacting in other ways (e.g. horizontal activation of agents across hierarchies by "*k*-lines").

Tyrrell [284] considers several kinds of hierarchical action selection techniques (prior to 1993) in his PhD thesis. He does not include the subsumption architecture (q.v.), although it would appear to resemble Tinbergen's hierarchical model, because it is "neither a computation description of the problem of action selection, nor an algorithmic description of how to select actions, but rather it is an implementation technique, which can be used to implement any number of different algorithms." [284]. Additionally, subsumption architectures often compose simple "behaviours" (i.e. subsumption architecture modules) into complex behaviours (in the sense usually used in this thesis) through emergence rather than selecting between a set of known actions. He does, however, include early work by Beer and Gallagher [18] involving neural network architectures.

In all the architectures considered by Tyrrell, and in more recent works by Bryson et al. ([234, 31, 41] etc.) there is a definite symbolic aspect: a known range of actions is determined by the researcher before action selection is attempted. Thus, "action selection" as a solution family (as opposed to a problem statement) appears to reflect a particular approach: we know the actions, how do we select between them?

2.2.2 "Reactive planning" action selection

One class of action selection technique involves pre-calculating actions to satisfy goals in a hierarchy, to avoid expensive goal planning: for example, in order to throw the Ring into the fire, Frodo must have the Ring and be at Mount Doom. In order to achieve the first goal, he must keep the Ring safe and recover it if lost. In order to achieve the second goal, he must travel to Mount Doom. In order to travel to Mount Doom he must travel to Mordor — and so on. Bryson [41] provides a useful classification of successively more complex techniques of this type, starting with completely reactive systems, passing through finite state machines and ending with systems with more complex transitions and a little more state (such as POSH reactive planning).

2.2.3 Hybrid architectures

In robotics, great strides have also been made in reconciling the two extremes of "deliberative" action selection and "reactive" systems which do no explicit modelling. These are "hybrid" systems, in which upper layers using symbolic, deliberative sense-plan-act loops connect to the environment through reactive, sub-symbolic

lower layers. In such systems, action selection is performed by the reactive part, while the planner typically alters the configuration of the reactive part. This leaves the upper layer free to plan over the long term, dealing symbolically with the environment, while the lower layer acts asynchronously with the upper, performing the high-level behaviours requested [12].

This is analogous to human conscious behaviour: we do not have to think about how to place our feet and shift our body weight as we walk; instead we specify that we are going to walk in a particular direction — our frontal lobes give the order, our motor cortex, cerebellum and spinal reflexes perform the actions.

One of the most successful hybrid systems is AuRA, a mid-1980s system using a deliberative planner, spatial reasoner and plan sequencer, layered over a reactive system termed the "schema controller" [11]. This primarily navigational system ultimately generates vector fields which are summed, normalised and passed to another process for execution.

2.2.4 Neurological action selection

Some recent work on action selection is based on recent developments in neuroscience. It has become possible to trace the relevant pathways in the vertebrate brain, and to develop simple models of the relevant regions in order to emulate their behaviour. Structures known as the basal ganglia are believed to be implicated ([229], [159] cited in [100]). These are believed to function by disinhibition: targets (such as the target area for a visual saccade) are maintained under constant inhibition, and selected when the inhibition is removed [50, 229]. This is superficially similar to a negated version of Tinbergen's "innate releasing mechanism" (q.v. above). Attempts have been made to build models of this neuronal arrangement with some success [100, 107]. Neurologically-inspired action selection has also been implemented at a higher level of abstraction in cognitive architectures such as ACT-R and Soar [163, 6, 207, 265].

2.2.5 Summary

"Action selection" as a problem statement is straightforward: from a given repertoire of actions and a given set of inputs, which action should be performed? This selection may be achieved on a reactive basis as a mapping from sensor states to actions, or by more deliberative reasoning involving the processing of these states, some form of memory, and a goal hierarchy. Much work has been done on the development of

action selection algorithms inspired by the ethology and neurology of vertebrates, but typically not at the level of the individual neuron.

However, action selection is not the only kind of adaptive behaviour: at a lower level, biological organisms do not select discrete actions to perform. Instead, they smoothly shift between different behaviours or perform behaviours with lesser or greater intensity dependent on external and internal factors. Other behaviours may emerge from the interplay of these fundamental functions, which may be advantageous (and thus be learned ontogenetically through reinforcement, or phylogenetically through natural selection).

The term "action" is to be taken loosely here — actions may be discrete, such as feeding, but may also include entities which act continuously but to a lesser or greater degree, such as those involved in the insulin-regulated maintenance of blood sugar levels. They may also include behaviours made up of several actions or simpler behaviours. Even if discrete actions are involved, the system driving them may use continuous quantities. For example, Walker and Wilson [295] use an artificial endocrine system with a number of hormones, each of which drive competing actions. These hormones are released into "pools", which fill until a threshold is crossed. All the hormone is then released and starts to decay. The action is selected by the hormone with the highest released concentration. However, the underlying methods are continuous: the result may be that the system oscillates between several behaviours, with each behaviour taking up a different amount of time depending on the release and decay rates of its hormone. Although action selection takes place in such systems, our interest is primarily in the systems which produce the continuous values which might feed into action selection.

2.3 Artificial neural networks

In parallel with the symbolic line of study, and (later) the reactive and behavioural work of Brooks, Braitenberg et al., which can trace their sources back to the cybernetic theories of Ashby et al. [13, 14], other researchers were building non-symbolic systems more directly inspired by biological nervous systems. ANNs are typically used in supervised learning, although the unsupervised and reinforcement paradigms also have examples.

2.3.1 The biological nervous system

The nervous system of an animal is made up of neurons, all of which have approximately the same structure, as shown in Figure 2.1. Each neuron can receive connections

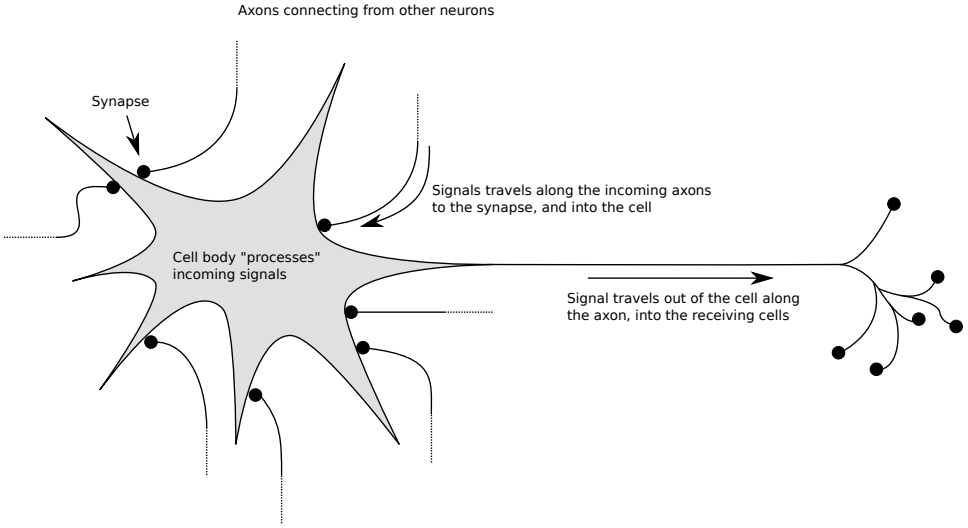


Figure 2.1: Structure of a neuron. This diagram shows a theoretical "typical" neuron; in practice neurons have extremely varied structures depending on their location and function.

from many others, and can itself transmit signals to other neurons, sometimes some distance away (via the axon). Where the axon joins the receiving cell is the *synapse*. The synapse can vary in connectivity — a "weak" synaptic join may only transmit a small part of any received signal. It is largely through making new synapses and varying their strengths that learning is thought to take place.

Communication is mediated at the synapse by chemicals known as *neurotransmitters*, which travel from the presynaptic (transmitting) neuron across a short gap (the *synaptic cleft*) to the postsynaptic (receiving) cell, where they combine with receptors. There are very many different neurotransmitters, including glutamate, GABA, serotonin, acetylcholine and dopamine, each with their own receptors. The action at the receiving neuron does not depend on the neurotransmitter, but on the postsynaptic receptor. This is either an excitatory action, making the potential across the cell membrane positive; or inhibitory, pushing the charge negative.

If the potential rises to a threshold level (of around -50mV), channels in the cell membrane open causing the cell to *depolarise*: the membrane potential rapidly increases, rising as high as +100mV, before dropping to below its resting potential. This process is shown in Figure 2.2.

This *action potential* triggers a wave of depolarisation which moves along the entire length of the axon, causing the release of neurotransmitters into more cells at the axon's synapses. After depolarisation, the cell returns to a level below the resting potential where a greater stimulus will be required for another action potential, before recovering to the normal resting potential. This is the *refractory period.*

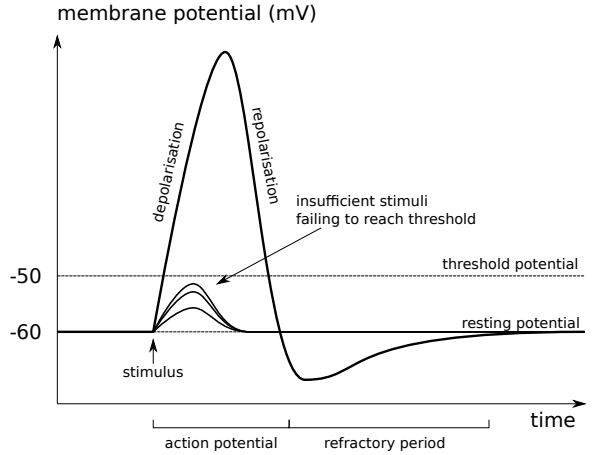


Figure 2.2: The neuronal responses to an excitatory stimulus of different levels; some failing to trigger an action potential, with one succeeding.

Thus it can be seen that both the intensity and the frequency of the stimuli are important: a number of spikes, each of which might not reach the threshold, might trigger the cell if they arrive in quick succession. In addition, the environment of the cell can modulate this behaviour in complex ways — some neurotransmitters are not broken down or reabsorbed, and thus can spend some time in the intercellular fluid, modulating the behaviour of the neurons therein.

2.3.2 McCulloch and Pitts

Much of this behaviour was described and modelled in the 1950's by Hodgkin and Huxley [124], but earlier McCulloch and Pitts [190] had produced a much simpler model of a neuron, which is still much used in simulation. This McCulloch-Pitts (MCP) neuron relies solely on intensity, without a refractory period or any temporal dynamics within the neuron itself. The only temporal element is a single cycle "synaptic delay", such that the output of a neuron appears at the corresponding inputs one cycle later (a discrete time model is used). Thus, the MCP neuron is far simpler than the Hodgkin-Huxley model, which is itself an extreme simplification.

The MCP neuron consists of a sum of weighted inputs, which is then passed to an "activation function" to produce the output. In addition, a single inhibitory input is provided: if this is active, the output of the cell will be zero (after one cycle). We can write:

$$y_{t+1} = \begin{cases} f\left(\sum_i w_i x_{ti}\right) & \text{if} \quad z_t > 0 \\ 0 & \text{if} \quad z_t \leq 0 \end{cases}, \tag{2.1}$$

where x_{ti} are the excitatory inputs at time t, z_t is the inhibitory input at time t, w_i are the weights, and f is the activation function. In the original MCP neuron, the

activation function is a step function with a constant threshold k:

$$f(x;k) = \begin{cases} 1 & \text{if} \quad x > k \\ 0 & \text{if} \quad x \leq k \end{cases}. \tag{2.2}$$

This allowed McCulloch and Pitts to represent any boolean function of real values with fairly complex temporal dynamics, although they make no attempt to describe how such networks can acquire the correct weights to approximate a desired function, i.e. "learn."

2.3.3 Hebbian learning and STDP

In 1949, neuropsychologist Donald Hebb developed a theory of learning at the neuronal level. In this theory, if an action potential from a neuron takes part in causing another neuron to itself form an action potential (to "fire"), the link between the two is strengthened such that the firing of the first neuron is more likely to cause the second to fire in the future [117]. This is now known as *Hebb's Law*. Though this is often summarised as "cells that fire together, wire together," the causative aspect is important — the first cell must take part in firing the second.

Although the mechanism was not fully understood at the time, more recent work has demonstrated its essential accuracy [171, 58]. If a neuron receives an input spike immediately before it activates, the synaptic connection between the two inputs is made stronger. "Anti-Hebbian" learning has also been demonstrated, in that if a neuron receives an input spike immediately after it activates, the connection is weakened. This biological process was termed *spike-timing-dependent plasticity* by Song, Miller and Abbott [260], who note a weakness it shares with other Hebbian rules:

> STDP, while making an important and novel contribution to competition, probably cannot be the sole source of plasticity in Hebbian learning situations. Like any other Hebbian modification rule, STDP cannot strengthen synapses in the absence of postsynaptic firing. If for some reason the excitatory synapses to a neuron are too weak to make it fire, STDP cannot rescue them. A non-Hebbian mechanism, such as synaptic scaling, may serve this function instead. [260]

Other disadvantages of the Hebbian learning rule are its inherent instability (synaptic weights will tend to increase or decrease exponentially) [213] and the potential for non-orthogonal patterns to interfere with each other [214].

Since this pioneering work, Hebbian learning has been a major component of artificial neural networks, particularly in unsupervised learning. In 1954, Farley and Clark [82] described a "self-organising system", consisting of a network of simplified MCP neurons and a modified Hebbian learning rule, which was able to perform simple pattern discrimination. Later, more widely used variants which deal with the inherent instability include Oja's rule [213], which can be used to generate a Principal Component Analysis; and the Generalised Hebbian Algorithm [240] of which Oja's rule is a special case.

2.3.4 The perceptron

The *perceptron* of Rosenblatt [236] is a simplification of the MCP neuron, removing the temporality and the inhibitory input and replacing the threshold with a constant bias. The actual calculation consists of a simple weighted sum, with the Heaviside step as the activation function:

$$y = \begin{cases} 1 & \text{if } \bar{w} \cdot \bar{x} + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

in vector notation. The perceptron algorithm specifies rules for updating the weights in order to perform supervised learning of classes of visual patterns. In the case of a binary classifier, a single perceptron is all that is required[7]. For each training example $\bar{x}$ with the desired class d and actual output y, the actual output is calculated as given in the equation above, then the weights are updated thus:

$$w_i \leftarrow w_i + \eta(d - y)x_i, \tag{2.4}$$

where x_i is input i, w_i is the weight associated with that input, d is the desired output for the class of the input (0 or 1) and y is the output after thresholding. The value η is a parameter called the "learning rate" which affects how quickly the weights change. Thus, for each training example, the weights are pulled towards those which would provide the correct classification[8]. The perceptron was originally envisaged as a physical machine rather than an algorithm, and while it was initially developed in software it was later rebuilt using photocells, with potentiometers and motors to

[7]Extending the perceptron for multiple classes simply requires creating a new perceptron for each class.

[8]This is essentially identical to the delta rule used for the output layer in back-propagation but assuming a linear activation function, so the derivative term is 1. However, because the step function is not differentiable, we cannot derive the perceptron learning rule from the delta rule.

manage weights and weight updates. This version, the "Mark I Perceptron", had 400 photocells in the input layer.

Rosenblatt, in a 1958 press conference, made statements about the perceptron which caused controversy in the AI community, and there were many reports in the press exaggerating its abilities. For example, the *New Yorker* wrote

> "The Perceptron... as its name implies, is capable of what amounts to original thought... it strikes us as the first serious rival to the human brain ever devised." (cited in [135])

This sensational reporting in the press, combined with Minsky and Papert's 1969 book *Perceptrons* [197], which deeply analysed the capabilities — and limitations — of Rosenblatt's machines, led to a marked reduction in connectionist AI research.

In fact, the book was not the unalloyed condemnation of perceptrons it is often perceived to have been. Minsky and Papert did point out that perceptrons *in their single layer form* were unable to detect global properties of the input fields they were given, such as parity (whether the number of 1 values is even or odd) and topological connectedness (whether all the 1 values are joined together, with no isolated islands). The parity problem is particularly significant, as its two-input form is the exclusive-OR function. In fact, the perceptron is only able to classify *linearly separable* problems: those which separate the input space's "true" and "false" regions by a hyperplane.

They did not, however, assert that a multilayer perceptron (MLP) could not implement such functions. However, the book was widely read — or perhaps not read — in this way. Indeed, Minsky himself has said

> "It would seem that *Perceptrons* has much the same role as the *Necronomicon*[9] — that is, often cited but never read." (personal communication to Istvan Berkeley, cited in [24]).

Although MLPs are able to implement non-linearly separable classifiers, provided the activation function is non-linear (since the sum of a set of linear functions is just another linear function), it was unclear how to train them efficiently. The fundamental problem is that of *credit assignment* — how to determine which connections contributed to a correct answer, to strengthen them; and which to an incorrect answer, to weaken them.

2.3.5 ADALINE and MADALINE

ADALINE [300], developed by Widrow and Hoff in 1960 independently from Rosenblatt's work [301], is a single-layer ANN largely identical to the perceptron in op-

[9] the famous *grimoire* in the works of H.P. Lovecraft

eration: the output is a thresholded weighted sum of the inputs, but the threshold operation typically used is the sign function. The learning rule is slightly different, however. Rather than adjusting the weights using the output error after thresholding, the pre-threshold output is used:

$$w_i \leftarrow w_i + \eta \left(d - f\left(\sum w_j x_j\right)\right) x_i \quad \text{(Perceptron update rule)} \tag{2.5}$$

$$w_i \leftarrow w_i + \eta \left(d - \sum w_j x_j\right) x_i \quad \text{(ADALINE update rule).} \tag{2.6}$$

The same definitions apply as in Eq. 2.4, with the thresholded weighted sum of all inputs substituted for y, using f to denote the threshold operator. The ADALINE will converge to the least squares error.

Like the perceptron, ADALINE is unable to learn non-linearly separable functions such as XOR and the parity problem, because there is no non-linearity in any model it generates. However, a multilayer ADALINE, or MADALINE, can express this by combining several ADALINES feeding into an OR gate. Three training rules exist, of which Rule III is essentially back-propagation of errors [301].

2.3.6 Back-propagation of errors

Although multilayer networks could be trained in limited ways by the MADALINE Rule I in 1962, their use was limited to classification problems because of the thresholding operation (the "quantizer" in Widrow's terminology [300]).

In 1974, Werbos [298] developed a gradient descent technique for supervised learning in MLPs with continuous outputs, suitable for regression problems. In order for this to work, the activation function must be differentiable so that there is a finite gradient to be descended at all points. The function used is typically a sigmoid function, such as the easily-differentiable hyperbolic tangent $\tanh(x)$ or the standard logistic function

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.7}$$

Both these curves have a similar S-shape, hence the "sigmoid" description, and both are easily differentiable. The range of $\tanh(x)$ is (-1,1), while the range of the standard logistic is (0,1).

As in the basic perceptron, each neuron consists of a weighted sum of the inputs plus a bias, run through an activation function ψ. In vector notation:

$$y = \psi(\bar{w} \cdot \bar{x} + b) \tag{2.8}$$

where y is the output, $\bar{w}$ is the weight vector, $\bar{x}$ is the input vector, and b is the bias. In this work we will use the form

$$y = \sigma\left(b + \sum_i w_i x_i\right), \tag{2.9}$$

with the activation function, typically a sigmoid, represented by $\sigma(x)$.

Back-propagation works, as the name suggests, by propagating the error term back from the output layer through the entire network, such that all weights and biases are moved towards values which would cause the network as a whole to move towards the correct output.

For each training example, the network is run "forwards" (i.e. in the fashion described above) to produce a result, calculating the error from the desired result. For each neuron the direction in which the weights and biases need to move to correct the error can be calculated by using the derivative of the activation function. The same calculation can then be applied to the values of the previous layer, and so on until we reach the input layer. This gives a complete set of corrections to apply.

While initially developed in the mid-1970s, back-propagation languished with the rest of ANN research in the "connectionist winter" brought about by the perceived failure of the perceptron. It was independently reinvented in the 1980s by Rumelhart, Hinton and Williams [237, 238], triggering renewed interest in the field. Feed-forward networks (i.e. no loops back from outer to inner layers) trained by back-propagation, as shown in Figure 2.3, are now a standard tool in the machine learning community. Although the differentiable activation function and continuous output makes this rather different from Rosenblatt's perceptron, much of the literature uses term "multilayer perceptron" to refer to this style of network. This term will therefore be used throughout the rest of this thesis to refer to a network of this type trained in this way.

Such a network is shown in Fig 2.3. This particular network has six inputs, a single hidden layer with three nodes, and six outputs. As such, it might be referred to as a "6-3-6" network, with nine nodes in total (the inputs are, of course, not nodes).

Back-propagation, like Rosenblatt's perceptron and the ADALINE variants, is a supervised learning technique: it relies on a large set of examples of desired inputs and outputs, and learns how to generalise from them so that unseen inputs will produce outputs congruent with the examples. It can also perform unsupervised learning by being provided with examples which have inputs identical to the outputs. In this "auto-encoder" case, the hidden layer, which should have far fewer nodes

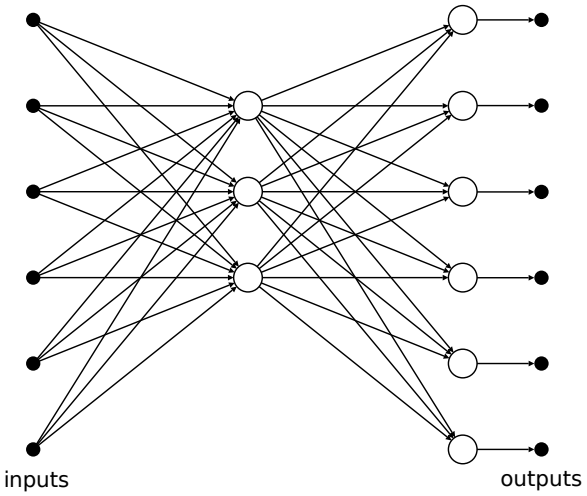


Figure 2.3: A feed-forward neural network (or multilayer perceptron) with a single hidden layer. All connections run left to right. Inputs and outputs are indicated as black dots, while the "neurons" or processing nodes are circles.

than the input/output layers, will learn a compressed representation of the inputs from which they can be regenerated.

Because the UESMANN network described in this thesis is trained with a modified back-propagation algorithm, back-propagation will be discussed in more depth in Sec. 4.1, along with some of the many enhancements which can provide better convergence behaviour.

2.3.6.1 Adaptive resonance theory

Many other forms of network have been described in the literature of varying levels of complexity. Adaptive Resonance Theory (ART) is one such family of architectures designed to solve the *stability/plasticity dilemma* of traditional ANN systems for both unsupervised and supervised learning. This dilemma involves finding a balance between plasticity (being able to learn new things) and stability (not forgetting older learned information [48]). It is included in this section as an illustration of how network models can be extended.

The basic unsupervised ART system consists of two fields of neurons, one holding the input and one holding "prototype patterns." A resonance between the two layers occurs when the input matches a prototype, in which case the prototype is updated. If no resonance is found, a new prototype is created and the input vector stored in it. Thus, input data will both update old stored patterns and store new data. A "vigilance parameter" is used to manage how close the patterns need to be to existing patterns to achieve resonance. A system could vary the vigilance

parameter depending on its state — in fact, this is a possible application for an artificial endocrine system.

Many variations of the core ART-1 binary pattern storage architecture exist, including ART-2 for real-valued patterns and the various ARTMAP systems, which use two ART modules to learn and correlate input and output patterns, thus achieving supervised learning [49].

A later version of ART, ART-3, attempts to model neurotransmitter regulation by incorporating Na^+ and Ca^+ ion concentrations [47]. Besides ART-2 and ART-3 there are a large number of variants of ART; a recent review of some of these variants and the recent state of ART can be found at [106].

2.3.7 Recurrent networks

We have so far discussed feed-forward networks[10] in which data always flows in one direction, from the input towards the output. Recurrent networks are those which have loops such that the input of a node relies — directly or indirectly — on its output, typically in a previous iteration of the algorithm. Given this delay, temporal dynamics may develop similar to those Ashby had already discovered in his homeostats [14]. Although not directly relevant to this thesis, it may be useful to briefly describe some of the more important developments.

2.3.7.1 Hopfield networks and Boltzmann machines

In a Hopfield network [126], nodes are connected to all others but not to themselves. Training is done using a Hebbian-like rule to form an associative memory, i.e. to remember a set of patterns and reproduce the full pattern when presented with a partial or erroneous version[11]. This network is similar to Ashby's homeostat, and thus is usually interpreted in dynamical systems terms.

A related network is the Boltzmann machine [79, 1], which uses a stochastic update mechanism and simulated annealing. Analysis of the information entropy in Hopfield and Boltzmann networks has led to a thermodynamic analogy of the function of biological neural networks [92, 253]. This view sees the nervous system as a mechanism for reducing predictive error in the form of information entropy.

2.3.7.2 Back-propagation through time

Back-propagation can be used to train feed-forward networks, but because the error must be propagated back through the network layers recurrent networks present

[10] With the exception of ART, which has two-way links between the two fields.

[11] Other rules with improved capabilities have also been devised [266].

difficulties. However, it is possible to treat the previous time iterations' networks as "virtual" layers of the current iteration, "unfolding the network" through time in a process called back-propagation through time [299]. However, this can lead to very deep networks, which back-propagation has difficulty training due the *vanishing and exploding gradient* problems — the magnitude of the gradients of components further from the outputs either increases rapidly as the algorithm moves backward through the layers, or approaches zero. Both problems make it difficult to learn long-term temporal patterns [21, 22]. For this reason, stochastic search techniques such as GAs are used, or back-propagation modified with regularisation [216] and other "deep learning" techniques such as alternative activation functions.

2.3.7.3 Real Time Recurrent Learning

Real Time Recurrent Learning (RTRL) [302] is a method for calculating the error gradient at every time step which does not require "unfolding" the network. It is suitable for online training because it does not require the sequence to be complete, unlike BPTT. Unfortunately, it has a time complexity of $O((N + L)^4)$ where N is the number of internal nodes and L is the number of output nodes [137].

2.3.7.4 Jordan and Elman networks: supervised learning of temporal patterns

These "simple recurrent networks" are two-layer feed-forward perceptrons with a set of "context neurons", which feed back from single neurons in the outer (Jordan) or hidden (Elman) layer to those same neurons [75, 76]. An Elman network is shown in Figure 2.4.

These networks are significant in that they can provide a feed-forward network — which can be trained with back-propagation — with a "memory" of previous activations in the form of the context neurons[12]. An Elman network was shown to be able to learn a "temporal version" of the exclusive-OR problem [75], in which the input consisted of an unbroken stream of bit triplets (p, q, r), in which $r = p \oplus q$. Elman also showed that the same network could learn to predict vowels and consonants in word-like sequences generated partially randomly, and partially from a simple set of generative rules [75]. This required the ability to "remember" over variable timescales.

Elman networks and the earlier Jordan network show that it is possible for a feed-forward neural network to be trained to respond appropriately to temporal sequences, provided there is a feedback element. A useful summary of early work in dynamic recurrent neural networks can be found in [218].

[12]The recurrent connections have fixed weights and are omitted from the back-propagation process.

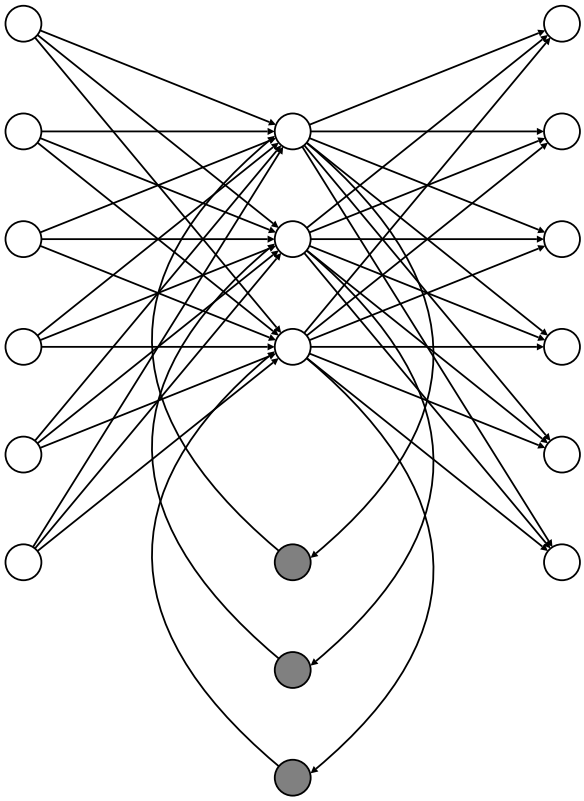


Figure 2.4: An Elman network with a hidden layer of 3 neurons and a context layer of 3 neurons.

2.3.7.5 Self-organising (Kohonen) maps

Self-organising maps are an unsupervised learning technique inspired by the architecture of the cerebral cortex, particularly how information appears to be represented spatially in 2-dimensional sheets with many lateral connections (connections between the neurons in the same layer) [153].

SOMs are often used to preprocess perceptual data into a lower dimensionality, often for use in sensor-space navigation [160, 245] — effectively learning a model (or map) of the environment.

2.3.7.6 Reservoir computing techniques

These systems use a "reservoir" consisting of a recurrent network of nodes with fixed and random weights on their connections, acting as a dynamical system. The output of the reservoir consists of the vector of the outputs for all the reservoir nodes, which will be a complex, time-dependent function of the inputs. These feed into an output (or "readout") layer which can be trained by learning the appropriate weights to generate the required output (e.g by linear regression). Thus the reservoir maps the

inputs into a high-dimensional space and provides a "memory" of previous states, in which patterns are analysed by readout layer.

The first example of such a technique was the "context reverberation network" of Kirby [148], but later work has been concentrated on Echo State Networks [136] and Liquid State Machines [180]. The former use "leaky integrator" nodes (q.v. Sec 2.3.9.1), while the latter use a spiking neuron model. Reservoir computing has also been amenable to implementation in unconventional physical substrates (due to the fixed dynamical system), such as memristor arrays and networks of analogue (or mechanical) oscillators [274]. It has also been suggested that some parts of the brain function as reservoirs [43].

2.3.8 Deep learning

As noted in Sec. 2.3.6, feed-forward networks in supervised learning are typically trained using back-propagation of errors, a gradient descent technique. However, this runs into problems when training "deep" networks — networks with more than a single hidden layer. This is because the error gradient becomes unstable as the algorithm moves further away from the output, either approaching zero (a "vanishing gradient") or (more rarely) increasing exponentially (an "exploding gradient") [21, 22]. Additionally, hidden nodes can "saturate" at values close to 0 or 1 where the derivative is zero with a logistic sigmoid activation function[13], leaving the algorithm unable to learn [101]. See Sec. 4.1 for more on these problems.

While a single hidden layer is all that is strictly required to perform any function [56], each layer provides an extra level of abstraction — typically feature detectors working on data provided by the deeper levels — which makes it easier to represent and thus learn a given problem [211, Chapter 1], [121]. Hence the need for a solution to the unstable gradient and saturation problems.

"Deep learning" as a term was first coined by Dechter [63], but was originally applied to the depth of search involved in satisfying complex systems of constraints in symbolic backtracking systems such as Prolog. However, "deep learning" in its current sense probably begins with the Neocognitron of Fukushima and Miyake [93] in 1982, which has been used for visual pattern recognition tasks. As such, the term usually describes supervised learning in a neural network as described above, but with modifications which allow the network to be deeper than the single hidden layer considered heretofore. This was inspired by the layered architecture of the primary visual cortex, which extracts local features in lower layers, and then detects

[13] or -1 and 1 with tanh()

these features in upper layers despite positional shifts and other deformations [127, 184].

The Neocognitron later served as inspiration for a 3 hidden layer convolutional network designed by LeCun et al. [169]. Used primarily in image pattern recognition, convolutional networks replace the weights on each individual connection in Fig. 2.3 with a convolution kernel, so that each node (representing a pixel) in a layer is a convolution of the pixels around a node in the previous layer. This adds translational invariance and vastly reduces the number of weights compared with a fully connected layer, permitting layers to learn filters which previously were often applied by hand, such as blurring, or orientation and edge detection. Pooling layers downsample previous layers, replacing a group of pixels with a single pixel (typically by finding the maximum), reducing the number of weights and also providing a small amount of translational invariance.

Much later, work by Hinton [121][14] and Bengio et al. [23] established that convergence in deep networks could be aided by *unsupervised pre-training*, which may be as simple as running each layer as an auto-encoder, reconstructing the training inputs. Once all layers are pre-trained, back-propagation is run on the entire network as normal. It is possible that pre-training helps guide the network towards minima which generalise better and that it effectively acts as a regulariser [77].

More improvements followed: Martens [183] demonstrated Hessian-free second order methods[15] which converge more directly to the minima. In 2011 the rectified linear unit activation function (ReLU) $f(x) = \max(0, x)$ began to replace the logistic sigmoid as the activation function of choice because of its lower susceptibility to unstable gradients, permitting deeper networks to be trained without pre-training. In 2013, Sutskever et al. [270] showed that well-chosen momentum and weight initialisation functions could aid convergence considerably.

The generative adversarial network (GAN) [102] is useful, as its name suggests, for generating data which is similar to the input data. These consists of two networks: a generator which takes a noise vector to produce images, and a discriminator which compares the generated images with the inputs. The two networks are (in a sense) competing: the generator learns to increase the discriminator's output error, while the discriminator learns to decrease it.

[14] This paper also has a useful history of the various independent discoveries of back-propagation and other techniques for training networks with hidden layers.

[15] These approximate the error with a second order quadratic and use conjugate gradient descent to avoid calculating the Hessian.

More recently, batch normalisation [134] at each layer has been used to normalise the activations of the previous layer per training batch[16], which may reduce "covariant shift" — changes in distributions of the training data. It also prevents nodes saturating due to this shift, and regularises the data. Finally, deep residual networks provide a way to train very deep networks of potentially hundreds of layers by introducing a "shortcut connection" which adds in the result of a previous layer [116]. In recurrent networks, gated recurrent units (GRUs) [51] and long short-term memory units (LSTMs) [123] have complex activation functions which ameliorate the vanishing gradient problem.

Given such a large repertoire of techniques, deep learning has been described as an "alchemical" discipline which applies methods without a rigorous understanding of how they work and interrelate with each other [226, 227, 133]. This has provoked a fierce debate [165, 220]:

> This [sudden failure of networks to converge when tiny changes to internals are made] is happening because we apply brittle optimisation techniques to loss surfaces we don't understand, and our solution is to add more mystery to an already mysterious technical stack. [226]

A recent (pre-print) survey of top-n recommendation techniques based on deep learning neatly illustrates the problem:

> Specifically, we considered 18 algorithms that were presented at top-level research conferences in the last years. Only 7 of them could be reproduced with reasonable effort. For these methods, it however turned out that 6 of them can often be outperformed with comparably simple heuristic methods, e.g., based on nearest-neighbor or graph-based techniques. The remaining one clearly outperformed the baselines but did not consistently outperform a well-tuned non-neural linear ranking method. [57]

Nevertheless, deep learning continues to be very widely used, and has come to dominate the machine learning field.

2.3.9 Evolutionary approaches

It is possible to use an evolutionary algorithm, such as the genetic algorithm of Holland [125], to evolve neural networks. In 1989, Montana and Davis [200] showed the feasibility of training feed-forward networks in a supervised learning environment

[16] See Sec. 4.1 for more details on batching.

using a genetic algorithm. A key advantage of such an algorithm is that, unlike with back-propagation, other learning paradigms than supervised learning are possible.

To learn adaptivity, adaptive systems require a function to be maximised or minimised (reward or cost), which requires reinforcement learning techniques or a stochastic technique such as evolution. In such a system an evolutionary algorithm cannot usually run online (i.e. in a running embodied system), because a population of networks must be tested for fitness repeatedly, with the process resulting in an average increase in the fitness of the entire population[17]. This may require running a large number of candidates for a long period of time, which is prohibitively expensive[18].

Because of these constraints, evolving adaptive behaviour usually deals with running simulations of robots to increase their adaptivity, typically with some kind of neural network component as the genotype. Many approaches also generate body plans and other physical parameters [66]. This is a large field, so this review will focus on two different neuroevolutionary approaches — evolutionary robotics [212] and NEAT [264] — because of their quite different methodologies. There are others, such as coevolutionary techniques like SANE [201], the evolved spiking neural network [90, 60]) and the Analog Genetic Encoding network [73, 288] — more examples can be found in [248, p. 32]. GasNets are often considered part of evolutionary robotics [288, Chapter 2], but due to their resemblance to artificial endocrine systems they will be considered separately. An important approach to note in passing is the Evolved Plastic Artificial Neural Network, or EPANN: this term covers several different architectures (including modifications to existing architectures) which are evolved to learn better by finding better plasticity rules (rules for modifying connections) [258].

2.3.9.1 Evolutionary robotics and the CTRNN

While a general trend, evolutionary robotics [212, 288] has the following three features [114]:

- minimal cognitive abilities: rather than attempting to emulate human reasoning, attempt to emulate the cognition of other forms of life (taking into account the wider definition of "cognition" implied by Ashby, Maturana and Varela [14, 187]);

[17]This is due to Holland's *Schema Theorem*: "schemata with above-average fitness (especially short, low order schemata), increase their frequency in the population each generation at an exponential rate, when rare [5, 125]."

[18]However, robots have been trained online by permitting the candidates to time share the robot, each operating for a few seconds. For example, in [91] a single robot is trained online with a population of 80 candidates, each running for 24 seconds.

- minimal prior assumptions: the architecture of an adaptive system is likely to bias how it adapts, constraining it to the capabilities of that architecture. For this reason, evolutionary robotics tends to use the *simplest possible* system which can generate the required behaviour, using the dynamical systems approach pioneered by Ashby, Maturana and Varela.
- existence proof approach: if an experiment shows that a minimal system can generate a given behaviour, then it will have shown a minimal set of conditions for that behaviour. This is useful knowledge for future engineering, but also from the biological standpoint: it may introduce new hypotheses, or refute requirements for a more advanced processing capability.

One common form of ANN used in evolutionary robotics is the Continuous Real-Time Recurrent Neural Network, which is based on a "leaky integrator" neuron model of the form [17, 15]

$$\tau_i \frac{d\gamma_i}{dt} = -\gamma_i + \sum_{j=1}^{n} w_{ji}\sigma(\gamma_j + \theta_j) + I_i(t), \tag{2.10}$$

where

- γ_i is the activation of neuron i;
- θ_i is a threshold term;
- $\sigma_(x)$ is the standard logistic function $(1 + e^{-x})^{-1}$;
- τ_i is a time constant associated with the "leakiness" of the notional cell membrane;
- w_{ji} is the weight of the connection from neuron j to i;
- $I_i(t)$ is the external input to neuron i at time t.

This simulates the firing rate of spiking neurons over time without having to simulate the complex dynamics of individual spikes shown in Figure 2.2 on page 24. The spike rate, rather than being some function of the inputs only, is increased by the inputs and drops over time. Topologically, the nodes in a CTRNN are not constrained in their connectivity: all nodes are connected to each other, including themselves. As a set of differential equations, a CTRNN constitutes a dynamical system. It has been shown that a CTRNN of this form is capable of approximating any other dynamical system with arbitrary precision [94]. Beer [16] views CTRNNs as dynamical rather

than connectionist: rather than modelling a network of layers of nodes with each layer learning a set of representations, the system is seen as a set of differential equations which describe how the state changes over time. The inputs serve as a perturbation to the system dynamics, rather than as a specification for an internal state:

> Abstractly, we can think of continuous-time recurrent neural networks as simply a basis dynamics out of which to build whatever agent dynamics is required and we can think of GAs as simply a technique for searching the family of flows defined by the parameterised network architecture for one whose dynamics [match the requirements]. [15]

2.3.9.2 Neuroevolution through augmenting topologies (NEAT)

In most evolutionary artificial neural networks, the topology of the network is fixed and the genome contains only weights and biases. This is true of CTRNNs — the network is fixed in size and fully connected. Attempts to evolve network topologies suffered from the "competing conventions" problem: the genomes for two candidates which might usefully be combined using the "crossover" genetic operator are often incompatible because the nodes they encode are used in different ways in each candidate [246]. This results in a "nonsensical" child candidate with a poor fitness. We will encounter the competing conventions problem again when we look at linear interpolation between feed-forward networks (see Sec. 4.5.6.4).

The NEAT algorithm of Stanley and Miikkulainen [264] solves this problem using historical markers on the genome: portions of the two parent genomes which come from the same historical mutation are aligned with each other. Other improvements include a form of speciation through "fitness sharing" within a niche and a minimal starting case. Several extensions of the basic algorithm exist — rtNEAT runs a large number of "agents" which are replaced continuously rather than in generations [262], while HyperNEAT uses indirect encoding to generate larger structures [263]. HyperNEAT has successfully been used in robotic gait control [170]. It should be noted that NEAT is not generally considered part of the larger evolutionary robotics field: because of the complexity of its mutation operation, it does not have the guiding principle of simplicity which is at the heart of ER.

2.3.10 GasNets and their relatives

GasNets were initially developed by Husbands [130, 132] at Sussex. They are evolutionary neural networks, with nodes located as points on a 2D plane. Node activation

functions are modulated in a complex way by the local concentration at the node of a "gas" quantity (originally modelled on nitric oxide, NO) which diffuses through the space. The "electrical" connections are determined by relative position and have weights +1 or −1, and nodes can be added during mutation.

As a recurrent ANN, there is a single time-step delay between inputs and outputs, which means that GasNets have temporal dynamics, even before the addition of any "gas". However, the chemical connections are partially decoupled from the electrical connections and typically operate over much longer time spans (owing to the time it takes for changes in the gas concentration to diffuse through the plane).

In the initial tests, GasNets performed well in tasks such as T-maze navigation [130] and target discrimination [131], showing themselves to be more evolvable than conventional binary networks without gas diffusion. Magg and Philippides [181] have demonstrated that GasNets outperform CTRNNs in tasks which require a timer (analogous to a biological central pattern generator, or CPG), such as temporal pattern discrimination, but are outperformed by CTRNNs in tasks with no such requirement. Magg also shows that another disadvantage of GasNets is the discrete binary weighting scheme — real weights help CTRNNs perform better.

Later work by Vargas, Di Paolo and Husbands [287] has shown that it may be the decoupling of the long-term chemical connections from the short-term electrical connections, rather than the spatial nature of GasNets, which leads to their increased evolvability. The NSGasNet does away with the notional 2D plane, replacing it with a matrix of weights between nodes. The temporal element is still present, because the gas still takes time to build up and decay, but the concentration is considered to be equal everywhere and a node's sensitivity is encoded in the node itself rather than a function of distance.

Indeed, NSGasNets are typically more evolvable than normal GasNets [287, 286], showing that the spatial nature of the network is unimportant:

> Results ... seem to indicate that the explicit use of spatial constraints and a spatially embedded diffusion process is not necessary to explain the success of GasNet models. Rather, the interplay between two distinct processes (electrical signals and gas modulation) acting on different timescales, and the multiplicative modulation effect of the gases appear to be the important factor. [286]

Further work by Moioli et al. [199] has attempted to combine an AES with two NSGasNets to evolve an artificially homeostatic system.

Philippides et al. [222] show that it is the loose coupling between the chemical and electrical processes which leads to improved evolutionary performance, because mutations in each of the two systems is unlikely to cause interference with the other.

In summary, GasNets can perform better than CTRNNs on tasks which require a pattern generator because the chemical system works over long time spans, and is decoupled from the electrical system. Thus, changes in the genome which affect the chemical system are unlikely to act to the detriment of the electrical system and *vice versa*.

In brains, nitric oxide diffuses freely through the intercellular space. As a neurotransmitter it has a modulatory effect on the functioning of nerve cells. However, this is a *paracrine* action, rather than an *endocrine* action: the modulator is released by cells and acts on nearby cells over time. In an endocrine system, hormones are released by specialised organs in response to internal or external stimuli, which circulate through the bloodstream much faster than paracrine intercellular diffusion. Thus GasNets are not artificial endocrine systems, but the NSGasNet could be considered as one.

2.4 The "reality gap"

The reality gap is the discrepancy between a simulated environment and the complex and unpredictable real world [140]. Control systems trained to perform well in simulation often fail to do so when transferred into the real world. To quote Brooks:

> There is a real danger (in fact, a near certainty) that programs which work well on simulated robots will completely fail on real robots because of the differences in real world sensing and actuation — it is very hard to simulate the actual dynamics of the real world. [34]

This problem can be dealt with by attempting to make the simulation as accurate as possible, with with detailed noise profiles of each sensor and actuator, with the simulator carefully validated against the real world environment, and the use of adaptive elements (such as neural networks) to "soak up" the discrepancies between the real world and the simulation [129, 140].

This can result in large and complex simulations which may run slowly, and still perform poorly, since no simulation can be perfect. This is a particular problem in evolutionary robotics, where each controller needs to be evaluated on several trials with many controllers created in each generation [139].

In evolutionary robotics, Jakobi [138, 139] describes the "minimal simulations" concept, which carefully considers which interactions between the controlling system

and the real robot and environment to model in the simulation: the so-called *base set* of interactions. The behaviour of base set aspects of the simulation is "hidden in an envelope of noise" [154] by varying their noise characteristics to ensure that controllers do not evolve to rely on a particular noise profile. The behaviour of other, implementation-dependent aspects is randomly varied between each trial, so that controllers cannot evolve to rely on these aspects.

A different approach is used by Koos, Mouret and Doncieux [154], involving finding a measure for the "simulation to reality" (STR) disparity and using this as one of the two objectives in a Pareto-based multi-objective evolutionary algorithm [62].

The minimal solutions methodology relies on modifying the simulation between trials during the evolution of a controller, effectively changing the adaptive landscape between each trial so that the controller does not fall into the constantly-changing minima introduced by the variation. As such, it may be possible to apply it outside the evolutionary paradigm. The approach of Koos, Mouret and Doncieux, however, relies fundamentally on an evolutionary approach.

2.5 Artificial endocrine systems

This next section will discuss artificial endocrine systems — adaptive systems which are inspired by the biological endocrine system (BES). These systems use chemicals known as *hormones* to transmit messages around the body. Hormones have two important properties: they decay slowly over time (rather like the nodes in a CTRNN but generally on much longer timescales) and they are broadcast rather than point-to-point [119].

Hormones operate over many different timescales. Some, like adrenaline, act over seconds; while others, like insulin, act over minutes or hours. These latter hormones are of particular interest because they are typically involved in homeostatic processes: insulin, for example, is released when blood sugar is high, prompting its uptake and storage. If blood sugar is too low, another hormone — glucagon — is released, prompting sugars to be released into the bloodstream. This process keeps blood sugar levels within a suitable range. Some hormones can act over much longer timescales, such as those which control the reproductive cycle.

Many tissues of the body are affected by hormones, including the nervous system, and the nervous system can itself signal the release of some hormones. This provides a way for the nervous system to signal to itself over the medium to long term, which could be a useful tool for neural network controllers.

2.5.1 Mathematical models of biological endocrine systems

In 1957, Danziger and Elmergreen [59] developed a ordinary differential equation (ODE) system model of the biological endocrine system:

$$\frac{\mathrm{d}h_i}{\mathrm{d}t} = x_i + \sum_j w_{ji} h_i - \lambda_i h_i \tag{2.11}$$

where h_i is hormone concentration, x_i is an external input, w_{ji} is sensitivity of hormone i to hormone j, and λ_i is a decay constant. Note the similarities between this and the CTRNN model in Eq. 2.10: this is essentially a fully-connected "leaky integrator" network. Later work has produced more accurate models [161, 81, 172, 80], but they are all essentially ODE leaky integrator systems, sometimes with the addition of Hill functions: sigmoids which model the binding of biochemical ligands (i.e. hormones) to their receptors [99]. Such models are criticised by Xu and Wang [305] as being difficult to analyse and hard to combine with other models. Therefore much later work has been on using "hormones" in a much looser biological sense as a metaphor for global leaky integrator communications.

2.5.2 Reactive and hybrid endocrine controllers

To this end, work has been done on building extremely simple controllers based on the interactions between the nervous and endocrine systems. Arkin [10] used a AuRA motor schema model (see [11] and Section 2.2.3) system to build an artificial endocrine system which modulated the path planning of a robot: under low power conditions, the robot was induced to take straighter (and more dangerous) paths around obstacles. This line of thought goes as far back as Walter's tortoise, whose perceptions were modulated by its power level such that when low on power, it saw the normally repulsive bright light of its charging station as attractive [105].

"Sozzy," the robotic vacuum cleaner of Yamamoto [306] uses a hormonal layer over a subsumption architecture, selecting between different behavioural repertoires, dubbed "emotions". A simpler system was implemented by the author in his undergraduate dissertation, controlling a switch between rolling and wheel-walking in an exoplanetary rover platform [86].

Work by Cañamero [46] notes the importance of emotions in behavioural adaptivity, and works from Brooks' approach to build a modular system of "motivations" (homeostatic drives) and "emotions" (which modulate the motivations and its self-perceived body state). Motivations and emotions are modelled through "hormones", values which decay over time.

Brooks himself has worked with artificial hormones [36, 37] using a version of Kravitz' model of the lobster endocrine system [155] which Brooks summarises in [37].

2.5.3 Connectionist, "neuroendocrine" controllers

Later work uses a lower-level, overtly connectionist paradigm, typically taking the form of just a few neurons and a single hormone. In one early example by Neal and Timmis [209], a robot was built with a simple (four unit) neural network which avoided obstacles. This network was then enhanced with a "hormone" — a value which increases the effective weights of all the neurons, strengthening the connections between them. This hormone is set to decay over time, and be "released" (i.e. increased) when the robot encounters an obstacle. This resulted in a robot which, when trapped (and thus constantly encountering obstacles) became increasingly active until this activity allowed it to escape, whereupon it "calmed down" gradually. It typically escaped from its entrapment sooner than a robot without the hormone, while suffering less damage than a robot in which hormone was released constantly.

As such, the artificial endocrine system resulted in a more adapted robot by smoothly moving between two patterns of behaviour: a less active pattern, which results in fewer potentially damaging collisions; and a more active pattern, which leads to quicker escapes. It is also worth noting that the robot often provoked an emotional response in those watching it, prompting calls to "leave the poor thing alone" when it was deliberately trapped. This is philosophically interesting, and points to possible methods for building more robots with more naturalistic behaviour patterns, such as those being designed to support the disabled or elderly [294].

Vargas et al. [289] have extended the Neal/Timmis system (see below) with positive and negative feedback mechanisms on hormone production to generate homeostatic behaviour. This system was then combined with a pair of previously evolved NSGasNets (see Sec 2.3.10) [199, 290]. Results indicated a considerably more robust performance when encountering novel environments (i.e. those for which the robot was not evolved), such as are found when transferring from simulation to a real robot: the so-called "reality gap" (See Sec. 2.4, p. 41).

2.5.4 Hormones and emotions

There are two major strands in the artificial endocrine system (AES) community. The high level strand exemplified by Cañamero [46] deals primarily with the modelling of emotional states, while lower level work by Neal and Timmis [209] and Vargas

et al. [290] deals with modelling the global effect of ambient substances on neural elements. The former strand therefore works in a "top-down" manner, observing the perceived emotional response of animals and attempting to model it, while the latter strand works "bottom-up", observing the neuromodulatory effect of chemicals and attempting to model those to produce responses which might be labelled "emotional."

Arbib and Fellous [8] describe work in delineating the neurological pathways involved in emotions, discussing how these interact with neuromodulators (which may be hormonal) with particular relevance to robotic applications. Fellous argues that the main purpose of emotions is to "achieve a multi-level communication of simplified but high-impact information." [84, p.3]. Thus, when an organism requires its entire behaviour, at all levels, to be changed in response to a stimulus, emotions provide a way to transmit information to multiple systems, allowing those systems to provide support for a course of action. They provide a way to abstract out irrelevant information, such as what a threat actually is — all the autonomic systems need to know is that a threat needs to be responded to, and the relevant systems should be functioning at high efficiency. He also mentions that such an efficient communications channel is advantageous when used between members of a species. If a threat message can be conveyed as a single, simple message (such as an alarm call or pheromone release), the emotion can be communicated to other organisms.

Fellous believes that "it may be crucial to understand emotions as dynamical patterns of neuromodulations, rather than patterns of neural activity, as is currently done" [84, p.6]. In his model, the common emotional states such as "anger" and "disgust" are the attractors of a neuromodulatory dynamical system. Other emotions between these states are possible ("a little angry but mostly depressed"), but these generally converge to the attractor states.

Another somewhat speculative and simplistic view of emotion is described by Lövheim [177], who models eight basic emotions as the combinations of binary values of the three neurotransmitters (in a neuromodulatory role) dopamine, serotonin and norepinephrine. The emotions modelled are those of Tomkins [281]: shame, distress, fear, anger, contempt/disgust, surprise, joy and excitement. For example, shame is modelled as low levels of all three, while surprise is low dopamine, but high serotonin and norepinephrine. While very simplistic, this model may have value in robotic emotional modelling [206, 273].

A wide review of recent work in "emotion augmented machine-learning" can be found in [268].

2.5.5 Neuromodulators and hormones

Returning to the lower, biochemical level, a biological neuromodulator is a neurotransmitter which is not reabsorbed in the synapse, and so continues to influence the synapse's behaviour and that of nearby cells — or even distant cells, since neuromodulators can circulate throughout the cerebrospinal fluid. Thus the "hormones" of the Neal/Timmis AES (see below) are more correctly termed "neuromodulators" in that they modulate neural behaviour. However, not all neuromodulatory hormones are secreted by neural tissue, and not all affect the neural cells directly — ghrelin, the "hunger hormone", is secreted by ghrelinergic cells in the intestinal tract [239] and modulates the behaviour of cells in different parts of the brain through complex cascades of chemical reactions [217]. Thus the terminology can become somewhat complicated, with many grey areas.

In the context of AES, both "hormone" and "(neuro)modulator" will be used in this work to describe any substance which acts globally to modulate all units within a neural network, although the degree and type of modulation may be determined locally (as it is in biology). However, some discussion of emotions (in Fellous' terms as an attractor within hormone space) may result.

In many of the models described in this chapter (including our own UESMANN model) the neuromodulator/hormone affects a single aspect of the artificial neuron's function: the synaptic strength. It must be borne in mind that this is a gross simplification. In biology, neuromodulators can change the behaviour of almost any aspect of the cells' functions in complex, non-linear ways. This can include the intrinsic nature of how the cell fires and the membrane potential as well as the synaptic strength [42]. There are also many hundreds of substances which have a neuromodulatory function (often in addition to other functions). In most artificial systems there are typically only a few modulators, each with one well-defined function.

While the rest of this section will focus more on systems in which the neuromodulator modulates the synaptic strength, Soltoggio et al. [259] describe a system which evolves a network in which neuromodulators are also evolved which modulate the plasticity of the neurons: the amount by which the weights change according to a version of Hebb's rule. This is inspired by the putative role of neuromodulators such as dopamine as "reward predictors", which respond to the possibility of reward by increasing plasticity [249, 69]. Yoder [307] later applied a similar method to a NEAT-like system with elements of GasNets.

2.5.6 The Neal/Timmis artificial endocrine system

The network discussed in this thesis is a simplification of the model used by Neal and Timmis [209], which uses a multiplicative modulation of network weights. The NTS has two distinct parts: the neuromodulatory model, which determines how the neuron units work and how they are modulated by hormone; and the release model, which determines how the hormone's concentration over time varies with release, decay and saturation.

2.5.6.1 Neuromodulatory model

The foundation of the Neal/Timmis model is the MLP node with the logistic sigmoid activation function of Sec. 2.3.6, although back-propagation is often not used:

$$y = \sigma\left(b + \sum_i w_i x_i\right). \tag{2.12}$$

Here, y is the output of a given node, b is a bias, w_i is the weight associated with each input x_i, and $\sigma(x) = \frac{1}{1+e^{-x}}$. In a system with a single hormone, the weights are modulated by the concentration of the hormone, assumed to have a nominal value of 1:[19]

$$y = \sigma\left(b + \sum_i w_i x_i h s_i\right) \tag{2.15}$$

where hormone concentration is h and the sensitivity of each weight to the hormone is s_i. The model can be extended with multiple hormones, where each neuron has a sensitivity to each hormone:

$$y = \sigma\left(b + \sum_i \left(w_i x_i \prod_j s_{ij} h_j\right)\right) \tag{2.16}$$

[19] In this model, the modulation is multiplicative. It is also possible to model the modulation in other ways, such as additively:

$$y = \sigma\left(b + \sum_i (h s_i + w_i) x_i\right). \tag{2.13}$$

but the amount of modulation will decrease as the weight itself increases. Another model, used by Sauzé and Neal [243] and by the author later in this thesis, is

$$y = \sigma\left(b + \sum_i (1 + hs) w_i x_i\right) \tag{2.14}$$

in which zero is the base level, with negative values for inhibition and positive values for excitation.

The original model applied modulation to the hidden layer only, and is shown in Fig. 2.5. Unmodulated nodes are straightforward MLP nodes. Later work by Sauzé and Neal [243] demonstrated that this arrangement behaves unpredictably under modulation, and switched modulation to the output layer. This is likely to be due to "competing conventions", a common issue when combining two disparate systems in a piece-wise manner, as described in Sec. 2.3.9.2. This issue will be examined in a little more detail in Sec. 4.5.6.4.

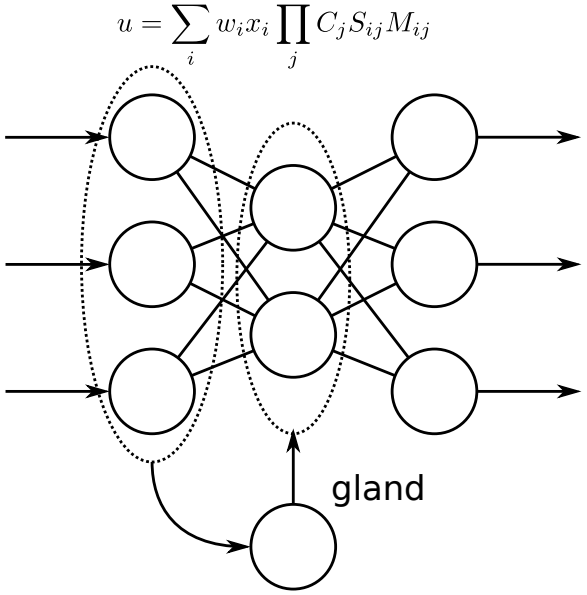


Figure 2.5: The Neal/Timmis AES model with the original formula from [209], which contains a receptor match term M_{ij} in addition to the sensitivity term S_{ij}, and defines C_j as the concentration of hormone j. Note that in this figure the hormone is modulating only the hidden layer — in [243] and much other work it is the output layer which is modulated — and that the stimulus comes directly from the environment (i.e. the input layer) rather than via any further processing.

2.5.6.2 Release model

In [209], the hormones are released by a notional "gland" at some release rate r, and decay geometrically. For a single hormone:

$$h_{t+1} = \tau h_t + \Delta t \cdot r_t \sum_i x_{it} \tag{2.17}$$

where h_t is the hormone level at time t, τ is a decay constant, Δt is the tick length, r_t is the release rate at time t, and x_{it} are the inputs at time t.

Timmis, Neal and Thorniley [279] and Timmis, Murray and Neal [278] later introduce a term to model hormone saturation. This gives release/decay

$$h_{t+1} = \tau h_t + \Delta t \cdot \frac{r_t}{1 + h_t} \sum_i x_{it}. \tag{2.18}$$

Another saturation model in unpublished work by Neal and the author has

$$h_{t+1} = \tau h_t + \Delta t \cdot (k - h_t) r_t \sum_i x_{it} \tag{2.19}$$

where k is the saturation level, typically 0.95. This is shown diagrammatically in Fig. 2.6. These saturation models work if the hormone is positive-valued only,

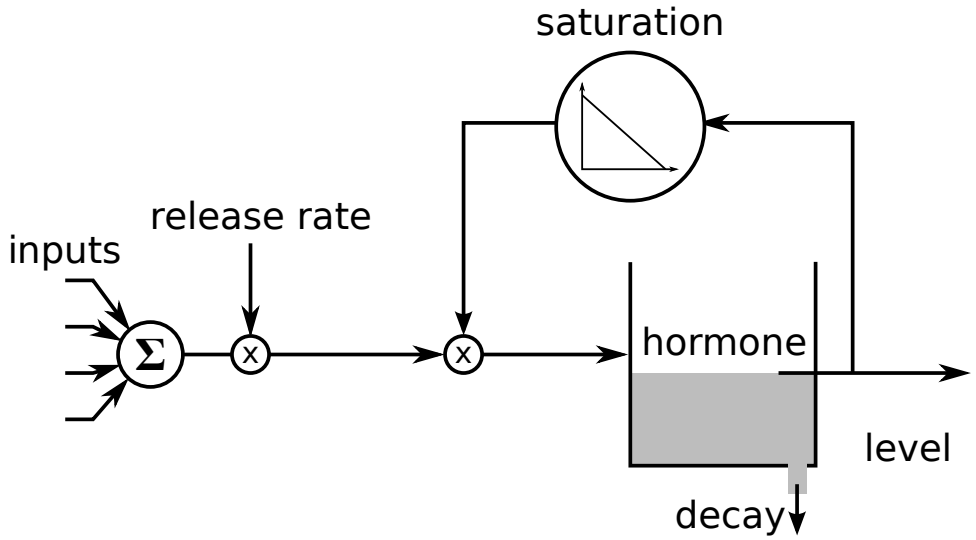


Figure 2.6: A diagram of the NTS hormone release model, showing how a "gland" node works. The effective output is the hormone level.

which may not be the case — Sauzé and Neal [243] discuss hormones which may be excitatory or inhibitory, depending on the sign.

2.5.6.3 Endocrine homeostasis in power management

One important possible application for an AES is power management, which requires homeostatic control. Consider a robotic system which must be deployed for a substantial amount of time, often with long intervals between recharge or refuelling. Such a system must make the most efficient usage of its available power resources. Good examples are planetary rovers and other exploratory/monitoring vehicles.

Such a system could benefit from a homeostatic model in which lower power levels lead to behaviour which would stop them falling any further (and ideally allow them to be recharged), while higher power levels could lead to more work

being done, making the most of the increased capability of the robot and reducing the possibility of battery damage from keeping them fully charged.

In work by Sauzé and Neal [243] a "battery level" hormone was used to modulate only the output layer of a neural network trained to sail a boat (modulating all the layers caused a nonlinear response between hormone concentration and actuator change). The sensitivities and weights were all set by hand. An increase in traversed distance was seen, as the system used the actuators less as the power decreased. At high power levels, the boat sailed more accurately by using the actuators more. In further simulated experiments, a solar panel was added to recharge the battery, and a "sunlight" hormone (an analogue to melatonin) was added. The simulated boat was now able to sail indefinitely, achieving a stable homeostasis, albeit by suppressing almost all activity for a large part of the day.

2.5.6.4 The Timmis-Neal-Thorniley adaptive AES

The work in [279] is an attempt to construct an AES which learns online using Hebbian techniques. The system consists of a robot with collision sensors, proximity sensors, and motors. The collision sensors always lead to avoidant behaviour, and the goal is to associate collision signals with proximity signals, so that proximity also leads to avoidance.

The system consists of two hand-built ANNs, one for "wandering" (the motor outputs simply drive the robot forwards, there are no inputs); and one for "avoiding" (the motor outputs avoid objects according to the proximity inputs — *not* the collision sensors). These ANN outputs are summed and clamped by a single layer perceptron before being sent to the motor, so the motor output is a blend of the two.

Each ANN is sensitive to one of two hormones, produced by two NTS glands of the form described in Figures 2.5 and 2.6. These AES do not have the receptor match term, so break down to

$$y = \sigma\left(\sum_i \left(w_i x_i \prod_j s_{ij} h_j\right)\right). \tag{2.20}$$

Note that there is no explicit bias term — bias is encoded as a weight for an input which is always 1. The release of the "wander" hormone is static, but the "avoid" hormone release is rather more complex. It is determined by two values, formed by weighted sums of each input:

$$A_p(t) = \bar{w_p}(t) \cdot \bar{x_p}(t) \tag{2.21}$$

$$A_c(t) = \bar{w_c}(t) \cdot \bar{x_c}(t) \tag{2.22}$$

where $A_p(t)$ is the activation of the hormone release for proximity at time t, $\bar{w_p}(t)$ is the vector of weights for proximity at time t, $\bar{x_p}(t)$ is the vector of proximity inputs at time t, and $A_c(t)$, $\bar{w_c}(t)$ and $\bar{x_c}(t)$ are similar values for collision activation, weights and inputs respectively. In the case of the collision sensors, the weight vector is set to unity so this becomes

$$A_p(t) = \bar{w_p}(t) \cdot \bar{x_p}(t) \tag{2.23}$$

$$A_c(t) = \sum_i \bar{x_{ci}}(t) \tag{2.24}$$

A hormone is released proportionally to these values, with release rate $R_g(t)$ and a constant stimulation rate α_g:

$$R_g(t) = \alpha_g(A_p(t) + A_c(t)) \tag{2.25}$$

The aim is for the network to learn an association between proximity and collision. To achieve this, the weights associated with the proximity sensor in releasing the hormone are modified by adding a delta each tick:

$$\bar{w_p}(t+1) = \bar{w_p}(t) + \Delta\bar{w_p}(t) \tag{2.26}$$

Several learning rules to produce $\Delta\bar{w_p}(t)$ were developed empirically and compared.

2.5.6.5 The neuroendocrine hexapod

Henley and Barnes [118] applied the NTS to a hexapod walker robot to modulate leg lift height for obstacle navigation. This was achieved by performing forward kinematics to determine the current foot position and feeding this into a NTS along with the desired leg height. The network was trained with back-propagation to learn the inverse kinematics required to produce the required joint angles, and was modulated by a hormone derived from sensor data, as shown in Fig. 2.7. They note

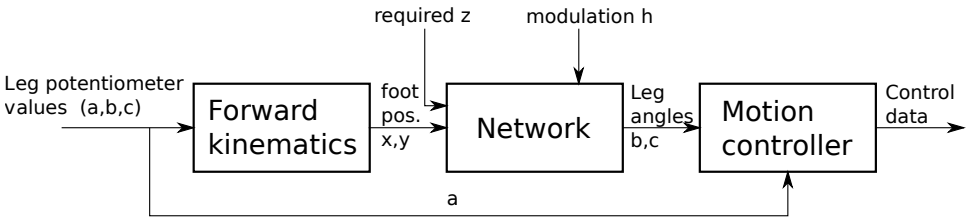


Figure 2.7: Henley and Barnes' neurokinetic hexapod limb controller, after Henley and Barnes [118]. Note that the network input is reduced to two dimensions: one angle is carried through directly to the output.

that as the hormone value increased, the non-linearity in the response due to the total modulation of the network led to undesirable transformations. This was resolved by selecting hormone sensitivities for particular hormones by hand, thus selectively modulating only part of the network.

2.5.6.6 Multi robot systems

Endocrine models have been applied to swarm robotics, such as in [203] (see also Sec. 2.5.7.1). Earlier work includes that of Walker and Wilson [295, 296], who evolved a endocrine-based (without an ANN) system to perform task selection among heterogeneous robots, which also employed an Evolution Strategy [25] for on-line learning while the system was running. This was based on work by Mendao [194], which used a similar system to perform action selection in a single robot. Each hormone in the system represents a task, and the hormone with the highest value determines which task is performed. Other behaviours are suppressed.

A key feature of this system is that the hormone is pooled in the notional "gland" before release, leading to a delay between its initial release and its influence on the system. Pooling permits modifications to the hormone before it actually becomes active, such as reducing the pooled quantity when another robot indicates it is performing a task. In Mendao's original system, in which the hormone modulated an ANN in a similar way to the NTS, pooling led to smoother behaviour.

In work by Timmis, Murray and Neal [278], a swarm of robots each employed multiple NTS networks with each network responsible for one of 11 behaviours, in order to perform a rubbish collection and disposal task. Each network was sensitive to a single hormone, which represented the behaviour. The network outputs were summed to arbitrate the behaviours. Weights and sensitivities in the networks were determined by both hand-coding and training with back-propagation. Hormone release values were calculated from sensor data by a programmed function. This system worked well, demonstrating interesting emergent swarm behaviour and benefiting from the temporal dynamics of the AES.

2.5.7 Other systems

2.5.7.1 SYMBRION and REPLICATOR

SYMBRION and REPLICATOR [147] were two EU funded projects in swarm robotics, building "symbiotic" organisms from smaller units using evolutionary methods, inspired by how some fungi and bacteria can sometimes conglomerate into multi-

cellular organisms to increase their survival (such as the dictyostelid slime moulds, which group to form "slugs" and then fruiting bodies when food is scarce).

The projects test a number of different approaches to the problem, including evolvable ANNs and artificial immune systems (AISs), but the primary citation above (which is essentially a preliminary note written before the experiments) describes a Hormone-Driven Robot Controller (HDRC). In this system, a genome contains a set of rules for the secretion, degradation and diffusion to other nodes of hormones which can alter sensor sensitivities, modulate controllers, trigger activities and even activate/deactivate subcontrollers. They note that hormones can generate homeostasis, aid adaptivity, create target-seeking behaviours (in the case of short term, fast-acting hormones), and generate temporal behaviours. Hormones diffuse both within the robots (e.g. between sensors and actuators) and between the robots.

An initial version, dubbed AHHS (Artificial Homeostatic Hormone System) appears in [267], while an enhanced version with the genome modified in an attempt to smooth the adaptive landscape is described in [110, 109, 247]. Each hormone has parameters for diffusion, decay and saturation encoded into one chromosome, with rules for how the hormone influences actuators and is secreted by sensors (and other hormones) in another chromosome. The AHHS2 differs from AHHS1 in that AHHS2 rules are more complex, allowing the combination of several sub-rules in a weighted manner. AHHS systems are not neuromodulatory – the behaviour is generated directly from the actions of hormones on the actuators.

As part of this project, Thenius, Zahadat and Schmickl [277], inspired by Fellous, have constructed EMANN: a complex RNN with hormonal regulation of all node functions and with all nodes capable of releasing hormone, evolved by a genetic algorithm. Later a rather simpler system instantiated in an evolving swarm of Braitenberg vehicles showed better performance than a hormone-free ANN in a feeding and reproduction task [203].

2.5.7.2 Artificial Hormone Network

The AHN of Teerakittikul, Tempesti and Tyrrell [276] uses a subsumption architecture whose locomotor outputs are modulated by a feed-forward network of three artificial hormones. These are implemented with fairly complex gland modules, which can pre-process their input signals in various ways. The robot on which the system is deployed is designed to navigate a complex environment, such as that found in rescue scenarios, and the modulation assists by slowing down the robot through areas where it is at risk of tipping over. The release of the first two hormones is stimulated when complex environments are detected through infrared and accelerometer. One

of the hormones is designed to detect conflicts in pitch data, resolving them in favour of the infrared hormone by changing their relative weights in a summation. In his later PhD thesis, Teerakittikul expands this system to detect wheel faults with more hormones, and ultimately uses Cartesian genetic programming to evolve an AHN to perform fault detection and amelioration [275].

2.5.7.3 Digital hormone model

The "digital hormone model" of Shen, Chuong and Will [256] is a cellular automaton in which the cells modify their behaviour based on hormones emitted according to Turing's reaction/diffusion equations [283], which permit hormones to activate and suppress other hormones. Shen's model is stochastic in that cells select their actions using a function based on connection data, sensor data, hormone concentrations and local state in a probabilistic manner. In a swarm robotics setting, the hormones diffuse through the continuous 2D space within which the robots move, with concentrations calculated and modified by each robot [255]. With this system, the robots can collectively find and surround an object, evenly cover an area and surmount obstacles. It has been used to construct a search map for multiple UAVs in a wide area search task [219].

2.6 Artificial immune systems

Artificial immune systems (AISs) [61] are based on models of the multiple mechanisms of the biological immune system, which are are a particularly rich seam for biologically inspired adaptivity. The immune system defends against pathogens (typically bacteria and viruses) and can learn to attack new pathogens it encounters. Behaviour of this type is useful in adaptivity — it permits the detection of novelty and in a wider sense is useful in pattern detection and recognition [89].

The immune system can be divided into two groups of components: those in the innate immune system, which all organisms have and which defend from a set of pathogens coded for in DNA; and those of the acquired immune system, which only vertebrates have and which can learn to recognise new pathogens. The innate system is relatively simple: the relevant cells have receptors which recognise known pathogens, and cause them to both attack and recruit other cells. One particular aspect of the innate immune response — cytokine release — has been used by the author to manage motor load across a six-wheeled robot [87]. This shares many features with the endocrine system (discussed in Sec. 2.5): diffusing substances modelled by leaky integrators which modulate behaviour.

The adaptive immune system is far more complex, and still only partially understood with several competing models of how it functions. Essentially, however, the mechanism relies on substances expressed by pathogens called *antigens*, and the constant evolution of *antibodies*, immune system substances which match antigens. AISs are typically based on these models, and there are as many different forms of AIS as there are competing theories.

The antibodies are generated constantly via a process known as *clonal selection* [44]. In "safe" areas of the body, candidate antibody-producing cells reproduce with extremely high mutation rates. Those which produce antibodies which match anything in this safe area (i.e. anything which belongs to the host) are eliminated, along which those which fail to function. The surviving cells circulate through the body. If an antibody-carrying cell binds to an antigen, it will trigger the immune response, proliferate, and become a larger part of the population which will itself undergo the same evolutionary process [89]. These cells can be very long-lived, so that a "memory" of previous antigen encounters is constantly circulating. This is a case in which a phylogenetic system (in which an entire population learns) is running inside an ontogenetic system (in which a single organism learns). Of course, this system is running inside another phylogenetic system: the population of the organisms which have the immune system.

This is very much a broad outline. The immune system is extremely complex and only partially understood, and can be viewed from several different perspectives. For example, the influential *idiotypic network theory* of Jerne [142] describes possible internal interactions of the immune system: binding not only to external antigens but also to itself. While this theory was initially well received and became a dominant paradigm for at least a decade [19], it lost prominence due to a lack of evidence [103] and the rise of competing theories. Certain aspects have been incorporated into the mainstream of immunological thinking, however, and artificial models of immune networks inspired by the theory have been successfully developed. Indeed, the first artificial immune systems were based on a dynamical systems model of idiotypic network theory [83], which is the most widely used artificial immune model in robotics [228]. Similarly influential[3], the danger theory of Matzinger [188] has also become mired in controversy [143, 55].

Aickelin, Dasgupta and Gu [4] provide a useful overview of artificial immune systems and describe two possible applications: intrusion detection and collaborative filtering (such as the recommender systems used by online shopping sites). In robotics, Raza and Fernandez [228] describe three AIS paradigms: clonal selection,

idiotypic networks, and danger theory [188]; and discuss various robotic applications from navigation and goal arbitration to gait acquisition.

Raza and Fernandez [228] note that most AIS applications require auxiliary components from other paradigms (such as reinforcement learning, fuzzy logic and genetic algorithms). This latter they suggest is particularly a problem: some researchers use genetic algorithms to evolve their "antibodies", leaving the question of how "pure" such an AIS might be. Also, most of these systems use the controversial idiotypic network theory: if this theory is invalid, are they "artificial immune systems?" However, as we have seen, so-called "artificial neural networks" are very far removed from how biological neural networks are now thought to function and still produce valid work. Thus most AISs are biologically inspired, but not necessarily bio-mimetic.

2.7 Summary

The viewpoint of this review has been limited largely to sub-symbolic, biologically inspired techniques for achieving adaptivity commonly used in robotic applications, because the artificial neuroendocrine system developed in the thesis is inspired by these varied approaches; also, in the future it may be possible to integrate such a system into an adaptive synthesis such as that suggested by the "Breach" system of Neal and Timmis [208]. There is an additional focus on the field of action selection, because the problem of autonomous behaviour is typically cast as an action selection problem.

There are, of course, a large number of other bio-inspired adaptive algorithms. These include phylogenetic methods, which reduce the mean of some error function across an entire population through a number of generations. Perhaps the most widely known are the evolutionary algorithms, which themselves have many variants from Evolution Strategies [25] to the genetic algorithm of Holland [125]. Other examples of algorithms which involve reducing error over a population include the loosely labelled "swarm intelligence" [28] methods. These are inspired by the collective behaviour of insects, birds and fish, and typically involve a large number of individuals which exchange information as they move through the solution space. Particle swarm optimisation [74] and ant colony optimisation [67] are included in this class. The latter has many variants, but all are based on the idea of "stigmergy" — agents which leave traces in the environment to influence other agents.

The work in this thesis will concentrate on biologically inspired methods for achieving homeostatic adaptation, but it is useful to briefly look outside this narrow

area. For example, the field of reinforcement learning can provide methods which can be used to construct adaptive systems. This typically uses probabilistic and other mathematical methods to learn which of a given set of behaviours lead to "good" outcomes, and to avoid those which lead to "bad" outcomes, and requires solving the "temporal credit assignment" problem [272]: how to determine which action led to a particular outcome when several actions may have intervened. Examples of classical reinforcement learning include the various temporal difference (TD) learning techniques [271] which use dynamic programming and other methods to update the values of actions as the system runs; and direct policy search, which searches the policy space directly for solutions [64]: a "policy" is a method of determining which action to take in which state, and is often implemented as a map giving the probability of each action in each state. As such, direct policy search covers a huge range of techniques in itself, from stochastic techniques such as evolutionary algorithms to gradient-based techniques. Q-learning [297] is a temporal difference method which has been implemented with modifications in a deep convolutional neural network with some success [198].

We have narrowed our study down to three biological subsystems which can be used to build adaptive systems:

- artificial neural networks, which typically work on very short timescales;
- artificial endocrine systems, which provide a medium to long term temporal capacity, and are often loosely coupled with neural networks by some neuromodulatory model;
- and artificial immune systems, which can provide a on-line learning facility for (in particular) negative reinforcement.

Of these three systems, only the first has a well-understood non-stochastic training paradigm (gradient descent through the back-propagation of errors) while the others are typically trained (if trained at all, in the case of AES) through phylogenetic stochastic techniques such as evolutionary algorithms. The primary aim of this thesis is to demonstrate and explore a gradient-based training algorithm for a simple neuromodulatory AES, initially for supervised learning.

Part II

The UESMANN Network

Chapter 3

Introducing UESMANN

The network this thesis will investigate is called UESMANN[1]. At its heart, it is a simple MLP of the kind used for supervised learning for many decades, but with the addition of a single neuromodulator. The starting point for UESMANN is the Neal/Timmis model of neuromodulation — while there are several variations of this model, the one used most in practice for a single hormone is given in Eq. 2.15. This will run into problems when $h = 0$ (i.e. the hormone has completely decayed) because the resulting activation will always be zero. Therefore Sauzé and Neal [242] [244, 243] modify this equation to

$$y = \sigma\left(b + \sum_i w_i x_i (1 + h) s_i\right) \tag{3.1}$$

for a single node, where y is the output or activation, σ is the activation function, b is a bias, w_i are the weights for each input x_i, h is the modulator (hormone) concentration, and s_i is the sensitivity to hormone of each weight. The activation function chosen is the logistic sigmoid,

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.7 revisited}$$

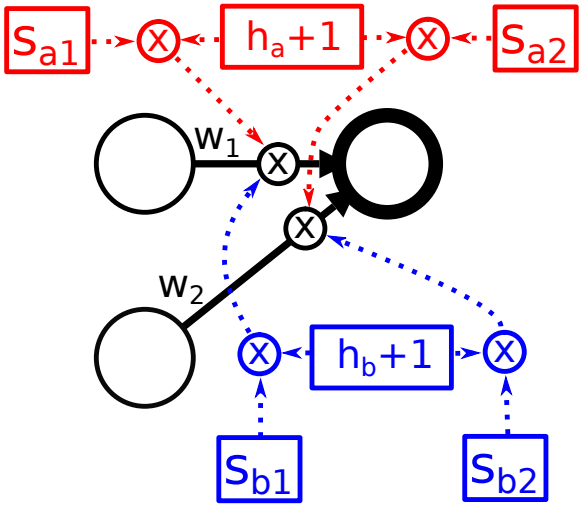
This system is shown in Fig. 3.1a. In the node specified in Eq. 3.1, the weights have their nominal or unmodulated values at $h = 0$, and may be inhibitory or excitatory with positive or negative values of s_i respectively. The goal is to have a uniform excitatory sensitivity to hormone for all weights, so this can be removed giving us

[1] Uniformly Excitatory Switching Modulatory Artificial Neural Network, pronounced "WES-mun."

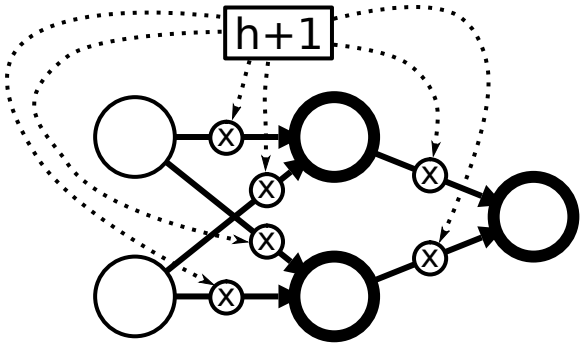
the core UESMANN function

$$y = \sigma\left(b + \sum_i w_i x_i (1 + h)\right), \tag{3.2}$$

as shown in Fig. 3.1b, alongside the Neal/Timmis system. As can be seen, UESMANN omits the hormone sensitivities.



(a) A Neal/Timmis system of the form given Eq. 3.1, with two hormones h_a and h_b. Only two weights are shown, w_1 and w_2. The sensitivities of the weights to h_a are s_{a1} and s_{a2}, while those to h_b are s_{b1} and s_{b2}.



(b) The UESMANN system with a modulator/hormone h, showing the entire network with all weights.

Figure 3.1: The Neal/Timmis AES of Sauzé and Neal [242], compared with the UESMANN system. In the former figure only two weights are shown for simplicity. In both figures, a circle with a thin border represents an input, a circle with a thick border represents a node, a solid edge represents a weight, and ⊗ represents multiplication. Dotted edges represent the influence of the modulator/hormone.

3.1 Can UESMANN represent all boolean pairings?

Much initial work in MLPs looked at whether networks can be trained to perform binary boolean functions, due to their simplicity and the tractability of the resulting networks to analysis. For example, [237] considers the exclusive-OR problem to be the "classic problem requiring hidden units." Additionally, the small number of parameters for networks which implement these functions, as well as the small

domains and ranges of the functions themselves, permits a detailed exploration of the solution space.

Therefore, rather than immediately attempting to solve more complex classification and control problems, it is worth initially evaluating the capabilities of UESMANN in the boolean functions. This is possible without a training algorithm because of the functions' simplicity: Monte Carlo methods can be used instead. This will give an indication of the power of the network architecture.

The key question for boolean nodes is "can a UESMANN network represent any pairing of boolean functions in the same parameter dimensionality as an equivalent MLP?" Here, a "pairing" means two functions f, g such that the network performs $f(x, y)$ when $h = 0$ and $g(x, y)$ when $h = 1$.

To answer this question, we first need to ask which boolean pairings can be represented in a single node. This was done both experimentally with Monte Carlo simulations and analytically.

3.1.1 Monte Carlo simulations of single nodes performing boolean functions

In this experiment, 10^{11} random nodes were generated with weights and biases in the range $[-40, 40]$ (chosen arbitrarily; runs performed with $[-10, 10]$ and $[-2, 2]$ performed similarly). The pseudo-random number generator. (PRNG) used was a 64-bit Mersenne Twister (the C++ Standard Template Library `mt19937_64` random number engine), which has a good compromise between performance, statistical randomness and period [162]. The resulting nodes were analysed to determine which functions they perform at $h = 0$ and $h = 1$, by feeding them all 4 possible inputs $x, y \in \{0, 1\}$ and thresholding the output at 0.5. Thus for inputs $x, y \in \{0, 1\}$ each node's thresholded activation a_{thresh} is given by

$$a_{thresh} = \begin{cases} 0 & \text{if} \quad u(x, y, h) < 0.5 \\ 1 & \text{if} \quad u(x, y, h) \geq 0.5 \end{cases}, \text{where} \tag{3.3}$$

$$u(x, y, h) = \sigma\Big(b + (h + 1)(w_1 x + w_2 y)\Big) \tag{3.4}$$

and thus

$$a_{thresh} = H\Big(b + (h + 1)(w_1 x + w_2 y)\Big), \tag{3.5}$$

where H is the Heaviside step function

$$H(n) = \begin{cases} 0 & \text{if} \quad n < 0 \\ 1 & \text{if} \quad n \geq 0 \end{cases}. \tag{3.6}$$

For all networks, a truth table was generated by inspecting a_{thresh} for all possible inputs at $h = 0$ and $h = 1$, and the generated function pairing obtained. This was used to maintain a count of the function pairings, with the results shown in Fig. 3.2. The white areas without squares show which function pairings are not represented in the output, while the remaining functions fall into several "bands" which vary from 19% of the total count (for the very frequent $T \to T$, $F \to F$ pairings[2]) down to 0.52% for pairings typically involving conjunctions (such as $x \wedge y$ and $\neg(x \wedge y)$). A

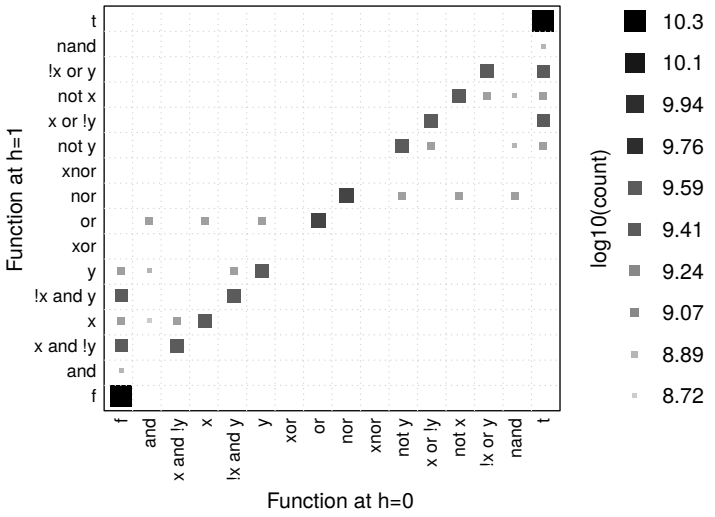


Figure 3.2: log_{10} of the counts of pairings of binary boolean functions produced by 10^{11} random random UESMANN nodes, when the output is thresholded at 0.5. The size and tint of each square gives the $\log_{10}$ of the count. Where there is no square, the count is zero: these are function pairings which never appear. Note that the square size and tint are scaled to the range seen in the data once the zeroes have been removed.

full table of the non-zero pairing frequencies is given in Table 3.1, using the symbol convention on page xxi.

The plot has several notable features. Firstly, the diagonal is strongly represented: it appears easier to transition from a function to itself (in the sense that the solution

[2]The $\to$ indicates a transition under modulator, with the left-hand side referring to the $h = 0$ case and the right-hand side referring to the $h = 1$ case, as described on page xxi.

space for these pairings is larger), although some self-pairings are not represented: $x \wedge y$, $\neg(x \wedge y)$, $x \oplus y$ and $\neg(x \oplus y)$ cannot transition to themselves. The latter pair are unrepresented because they are not linearly separable, and so cannot be performed in a single node. The frequencies of self-pairings are all larger than the frequencies of other pairings.

Secondly, pairings involving T or F as the $h = 0$ function are also strongly represented — but not in the $h = 1$ position; these appear to be impossible except for the self-pairing cases. Thus there is a qualitative difference between the functions which can be learned at $h = 0$ and $h = 1$. This implies that the ordering of the functions is important: it may be possible to represent $f \to g$ but not $g \to f$.

Finally, the plot shows a rotational symmetry. Due to the ordering of the functions along the axes, this shows that the probability of a given function pair is equal to the probability of those functions negated: $P(f, g) = P(\neg f, \neg g)$ where $P(f, g)$ is the probability of the given pairing. This is because negating the weights and biases in any perceptron node with a logistic sigmoid activation, including a UESMANN node, will negate the output of the node's pre-activation function, which will negate the boolean thresholded output (since $\sigma(-x) = 1 - \sigma(x)$).

This plot shows that UESMANN is able to perform only a small subset of the boolean pairings in a single node. Naturally any pairing involving the exclusive-OR function or its negation will be impossible, since these are non-linearly separable, but there are still a large number of pairings unaccounted for. The next two sections will provide algebraic and geometric justifications for these.

Despite the low number of pairings here, a network containing a single hidden layer of UESMANN nodes may be able to combine these to perform most (if not all) boolean pairings: consider that without modulation a hidden layer of two nodes (if we do not permit connections to skip a layer) is required to perform any non-separable binary boolean function[237]. We will investigate networks with hidden layers in Sec. 3.1.4.

Table 3.1: Frequencies of boolean function pairings performed in a Monte Carlo simulation of 10^{11} random UESMANN units inputs x, y and weights and biases in $[-40, 40]$. Rules are given between each band of frequencies.

$u(x, y, 0)$	$u(x, y, 1)$	percentage of total
T	T	19.27 %
F	F	19.27 %
$\neg(x \vee y)$	$\neg(x \vee y)$	4.17 %
$x \vee y$	$x \vee y$	4.17 %
$\neg y$	$\neg y$	3.13 %
$\neg x \wedge y$	$\neg x \wedge y$	3.13 %
x	x	3.13 %
y	y	3.12 %
$x \vee \neg y$	$x \vee \neg y$	3.12 %
$\neg x$	$\neg x$	3.12 %
$\neg x \vee y$	$\neg x \vee y$	3.12 %
$x \wedge \neg y$	$x \wedge \neg y$	3.12 %
T	$\neg x \vee y$	2.60 %
F	$x \wedge \neg y$	2.60 %
T	$x \vee \neg y$	2.60 %
F	$\neg x \wedge y$	2.60 %
$\neg y$	$\neg(x \vee y)$	1.04 %
$x \wedge y$	$x \vee y$	1.04 %
$\neg x \vee y$	$\neg x$	1.04 %
F	y	1.04 %
y	$x \vee y$	1.04 %
$\neg x$	$\neg(x \vee y)$	1.04 %
F	x	1.04 %
x	$x \vee y$	1.04 %
$x \wedge \neg y$	x	1.04 %
$\neg(x \wedge y)$	$\neg(x \vee y)$	1.04 %
$x \vee \neg y$	$\neg y$	1.04 %
$\neg x \wedge y$	y	1.04 %
T	$\neg x$	1.04 %
T	$\neg y$	1.04 %
$x \wedge y$	y	0.52 %
$\neg(x \wedge y)$	$\neg y$	0.52 %
$\neg(x \wedge y)$	$\neg x$	0.52 %
T	$\neg(x \wedge y)$	0.52 %
F	$x \wedge y$	0.52 %
$x \wedge y$	x	0.52 %

3.1.2 Single-node UESMANN as a system of inequalities

We wish to find those pairings of boolean functions which are possible in a single UESMANN node, given a Heaviside step activation function (which is essentially a thresholded form of the logistic sigmoid[3]). Returning to Eq. 3.5 and given that $f(x, y) = u(x, y, 0)$ and $g(x, y) = u(x, y, 1)$, we obtain the two equations

$$f(x, y) = H(b + w_1 x + w_2 y) \tag{3.7}$$
$$g(x, y) = H(b + 2w_1 x + 2w_2 y). \tag{3.8}$$

Using these, we can construct a set of inequalities for each function and their values, given the truth tables for f and g. Consider the case for both inputs zero, i.e. for $f(0, 0)$:

$$f(x, y) = H(b + w_1 x + w_2 y) \qquad \text{(Eq. 3.7)} \tag{3.9}$$
$$f(0, 0) = H(b + w_1 0 + w_2 0) \qquad \text{(subst. } x = 0, y = 0) \tag{3.10}$$
$$= H(b). \tag{3.11}$$

If we substitute $f(0, 0) = 0$ we can find the conditions of weights and bias under which the output is zero when both inputs are zero for function f (i.e. $h = 0$):

$$H(b) = f(0, 0) \qquad \text{(Eq. 3.11)} \tag{3.12}$$
$$H(b) = 0 \qquad \text{(subst. } f(x, y) = 0) \tag{3.13}$$
$$b < 0 \qquad \text{(Heaviside step function, Eq. 3.6).} \tag{3.14}$$

Similarly for $f(0, 0) = 1$:

$$H(b) = f(0, 0) \qquad \text{(Eq. 3.11)} \tag{3.15}$$
$$H(b) = 1 \qquad \text{(subst. } f(x, y) = 1) \tag{3.16}$$
$$b \geq 0 \qquad \text{(Heaviside step function, Eq. 3.6).} \tag{3.17}$$

[3]Note also that $H(x) = \lim_{k\to\infty} \frac{1}{1+e^{kx}}$

Thus we can see that the sign of the bias b determines the value of $f(0,0)$. Similarly for g, we obtain

$$g(x,y) = H(b + 2w_1x + 2w_2x) \qquad \text{(Eq. 3.8)} \tag{3.18}$$
$$g(0,0) = H(b) \qquad \text{(subst. } x = 0, y = 0) \tag{3.19}$$
$$= f(0,0), \tag{3.20}$$

showing that $f(0,0) = g(0,0)$ for all pairings — boolean UESMANN nodes cannot change their output under modulation when both inputs are false (under the chosen representation where 0 represents falsehood). Therefore we can write

$$f(0,0) = 0 \iff g(0,0) = 0 \iff b < 0 \qquad \text{(from Eqs. 3.14 and 3.20)} \tag{3.21}$$
$$f(0,0) = 1 \iff g(0,0) = 1 \iff b \geq 0 \qquad \text{(from Eqs. 3.17 and 3.20).} \tag{3.22}$$

Now consider the case where $f(1,0) = 0$. Here we get

$$f(x,y) = H(b + w_1x + w_2y) \qquad \text{(Eq. 3.7)} \tag{3.23}$$
$$0 = H(b + w_1) \qquad \text{(subst. } x = 1, y = 0, f(x,y) = 0) \tag{3.24}$$
$$H(b + w_1) = 0 \tag{3.25}$$
$$b + w_1 < 0 \qquad \text{(Heaviside step function, Eq. 3.6)} \tag{3.26}$$
$$w_1 < -b. \tag{3.27}$$

Similarly, $f(1,0) = 1$ will give

$$H(b + w_1) = 1 \qquad \text{(Eq. 3.25, changing 0 to 1)} \tag{3.28}$$
$$b + w_1 \geq 0 \tag{3.29}$$
$$w_1 \geq -b. \tag{3.30}$$

Therefore

$$f(1,0) = 1 \iff w_1 \geq -b \tag{3.31}$$

and

$$f(1,0) = 0 \iff w_1 < -b. \tag{3.32}$$

Repeating the argument of Eqs. 3.23 to 3.30 for the $f(0,1)$ case, it is easy to show that

$$f(0,1) = 1 \iff w_2 \geq -b, \tag{3.33}$$
$$f(0,1) = 0 \iff w_2 < -b. \tag{3.34}$$

In the $f(1,1) = 0$ case,

$$f(x,y) = H(b + w_1 x + w_2 y) \quad \text{(Eq. 3.7)} \tag{3.35}$$
$$0 = H(b + w_1 + w_2) \quad \text{(subst.)} \tag{3.36}$$
$$H(b + w_1 + w_2) = 0 \tag{3.37}$$
$$b + w_1 + w_2 < 0 \quad \text{(Heaviside step)} \tag{3.38}$$
$$w_1 + w_2 < -b. \tag{3.39}$$

It follows that for $f(1,1) = 1$,

$$H(b + w_1 + w_2) = 1 \quad \text{(Eq. 3.37, changing 0 to 1)} \tag{3.40}$$
$$b + w_1 + w_2 \geq 0 \quad \text{(Heaviside step)} \tag{3.41}$$
$$w_1 + w_2 \geq -b. \tag{3.42}$$

So

$$f(1,1) = 0 \iff w_1 + w_2 < -b, \tag{3.43}$$
$$f(1,1) = 1 \iff w_1 + w_2 \geq -b. \tag{3.44}$$

We can reiterate these arguments for the g functions. We have already dealt with $g(0,0)$ in Eqs. 3.21 and 3.22 above. For $g(1,0) = 0$,

$$g(x,y) = H(b + 2w_1 x + 2w_2 y) \quad \text{(Eq. 3.8)} \tag{3.45}$$
$$0 = H(b + 2w_1) \quad \text{(subst. } x = 1, y = 0, g(x,y) = 0) \tag{3.46}$$
$$H(b + 2w_1) = 0 \tag{3.47}$$
$$b + 2w_1 < 0 \quad \text{(Heaviside step)} \tag{3.48}$$
$$2w_1 < -b \tag{3.49}$$
$$w_1 < -b/2. \tag{3.50}$$

Similarly, $g(1,0) = 1$ will give

$$H(b + 2w_1) = 1 \quad \text{(Eq. 3.47, changing 0 to 1)} \tag{3.51}$$
$$b + 2w_1 \geq 0 \quad \text{(Heaviside step)} \tag{3.52}$$
$$2w_1 \geq -b \tag{3.53}$$
$$w_1 \geq -b/2 \tag{3.54}$$

which gives us

$$g(1,0) = 0 \iff w_1 < -b/2, \tag{3.55}$$
$$g(1,0) = 1 \iff w_1 \geq -b/2. \tag{3.56}$$

Again we can find the $g(0,1)$ cases with a similar argument, giving

$$g(0,1) = 0 \iff w_2 < -b/2, \tag{3.57}$$
$$g(0,1) = 1 \iff w_2 \geq -b/2. \tag{3.58}$$

Finally, the $g(1,1)$ cases can be dealt with. For $g(1,1) = 0$,

$$g(x,y) = H(b + 2w_1x + 2w_2y) \quad \text{(Eq. 3.8)} \tag{3.59}$$
$$0 = H(b + 2w_1 + 2w_2) \quad \text{(subst.)} \tag{3.60}$$
$$H(b + 2w_1 + 2w_2) = 0 \tag{3.61}$$
$$b + 2w_1 + 2w_2 < 0 \quad \text{(Heaviside step)} \tag{3.62}$$
$$w_1 + w_2 < -b/2, \tag{3.63}$$

and for $g(1,1) = 1$,

$$H(b + 2w_1 + 2w_2) = 1 \quad \text{(Eq. 3.61, changing 0 to 1)} \tag{3.64}$$
$$b + 2w_1 + 2w_2 \geq 1 \quad \text{(Heaviside step)} \tag{3.65}$$
$$w_1 + w_2 \geq -b/2. \tag{3.66}$$

These give us

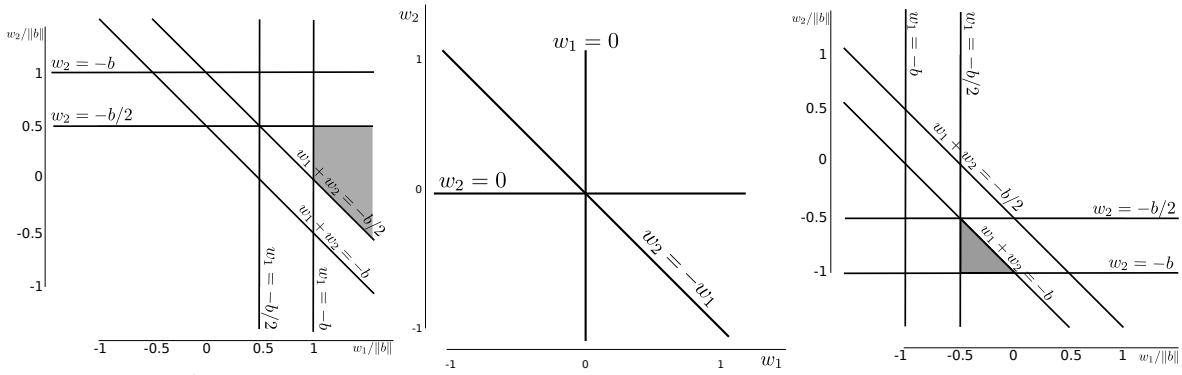
$$g(1,1) = 0 \iff w_1 + w_2 < -b/2, \tag{3.67}$$
$$g(1,1) = 1 \iff w_1 + w_2 \geq -b/2. \tag{3.68}$$

We can now construct a set of inequalities for UESMANN boolean nodes which impose conditions on the values of b, w_1 and w_2 given known values in the truth tables for f and g. These are shown in Table 3.2. We see that although there are 16 inequalities in this set, the inequalities for $f(0,0)$ and $g(0,0)$ are the same, leading to a solution space separated by seven planes. These are shown in Fig. 3.3, plotted as 2D lines by using the ratio of weight to bias magnitude. At $b = 0$, pairs of the planes intersect to give the separating lines $w_1 = 0$, $w_2 = 0$ and $w_2 = -w_1$.

The truth tables for f and g each contain four entries. Each of these can be translated into an inequality using Table 3.2, resulting in eight inequalities (four for

Table 3.2: Function output to inequality mapping for UESMANN, showing how certain outcomes of the f and g functions impose conditions on the bias and weights in a single node, where f is performed at $h = 0$ and g is performed at $h = 1$.

Function output	Inequality	Source in text
$f(0,0) = 0$	$b < 0$	Eq. 3.21
$f(0,0) = 1$	$b \geq 0$	Eq. 3.22
$f(1,0) = 0$	$w_1 < -b$	Eq. 3.32
$f(1,0) = 1$	$w_1 \geq -b$	Eq. 3.31
$f(0,1) = 0$	$w_2 < -b$	Eq. 3.34
$f(0,1) = 1$	$w_2 \geq -b$	Eq. 3.33
$f(1,1) = 0$	$w_1 + w_2 < -b$	Eq. 3.43
$f(1,1) = 1$	$w_1 + w_2 \geq -b$	Eq. 3.44
$g(0,0) = 0$	$b < 0$	Eq. 3.21
$g(0,0) = 1$	$b \geq 0$	Eq. 3.22
$g(1,0) = 0$	$w_1 < -b/2$	Eq. 3.55
$g(1,0) = 1$	$w_1 \geq -b/2$	Eq. 3.56
$g(0,1) = 0$	$w_2 < -b/2$	Eq. 3.57
$g(0,1) = 1$	$w_2 \geq -b/2$	Eq. 3.58
$g(1,1) = 0$	$w_1 + w_2 < -b/2$	Eq. 3.67
$g(1,1) = 1$	$w_1 + w_2 \geq -b/2$	Eq. 3.68



(a) $b < 0$ **(b)** $b = 0$ **(c)** $b \geq 0$

Figure 3.3: The separating planes of the inequalities in Table 3.2, shown as two plots of weight against bias magnitude, separately for $b < 0$, $b = 0$ and $b \geq 0$. The seventh plane is $b = 0$. In Fig 3.3c the example region discussed in the text for the pairing $\neg(x \wedge y) \rightarrow \neg y$ is highlighted in grey, while Fig 3.3a highlights the example region discussed for the pairing $x \rightarrow x$.

f and four for g). Thus any putative pairing $f \rightarrow g$ can be stated as a system of eight inequalities. If these cannot be satisfied, a UESMANN node cannot represent the pairing. Consider, for example, the inequalities involving both inputs at zero, Eqs. 3.21 and 3.22. Since $b \geq 0$ and $b < 0$ is a contradiction, a single UESMANN

Table 3.3: Truth table for pairing $\neg(x \wedge y) \rightarrow \neg y$, with the associated inequalities from Table 3.2.

x	y	$f(x,y) = \neg(x \wedge y)$	$g(x,y) = \neg y$	Inequality for f	Inequality for g
0	0	1	1	$b \geq 0$	$b \geq 0$
0	1	1	0	$w_1 \geq -b$	$w_1 \geq -b/2$
1	0	1	1	$w_2 \geq -b$	$w_2 < -b/2$
1	1	0	0	$w_1 + w_2 < -b$	$w_1 + w_2 < -b/2$

node cannot represent a function pairing in which the output changes value while the inputs remain zero, such as $x \rightarrow \neg x$ or $y \rightarrow T$. As another example, the pairing $\neg(x \wedge y) \rightarrow \neg y$ has the truth table given in Table 3.3, shown with the inequalities derived by looking up each entry in the table in Table 3.2. Thus any putative boolean UESMANN node which performs $\neg(x \wedge y) \rightarrow \neg y$ must satisfy the system of simultaneous inequalities given in Eq. 3.69.

$$\begin{cases} b \geq 0 \\ b \geq 0 \\ w_1 \geq -b \\ w_1 \geq -b/2 \\ w_2 \geq -b \\ w_2 < -b/2 \\ w_1 + w_2 < -b \\ w_1 + w_2 < -b/2 \end{cases}. \tag{3.69}$$

Since this system contains no contradiction, the pairing will have a solution, which is shown in Fig 3.3c.

Given Table 3.2, several approaches can be taken to find the possible pairings. We can find contradictions within these equations, each of which will eliminate a set of potential outputs. One example has already been given: $(b \geq 0) \wedge (b < 0)$ is a contradiction which eliminates cases where $f(0,0) \neq g(0,0)$. However, given the number of inequalities this will involve a large number of contradictions. Alternatively, we could look at the consequences of conjunctions which are not contradictions: for example,

$$(b \geq 0) \wedge (w_1 \geq -b/2) \implies w_1 \geq -b \tag{3.70}$$

$$(f(0,0) = 1 \vee g(0,0) = 1) \wedge g(1,0) = 1 \implies f(1,0) = 1 \quad \text{(from Table 3.2).} \tag{3.71}$$

While this approach might tell us interesting things about the system, arriving at the set of permitted pairings would be tedious. This single boolean node system itself is not important — what is important is what the node can do, which pairings are permitted. Instead, we shall simply analyse the pairings performed by each region of the solution space when divided by the separating planes shown in Fig. 3.3, which will give an exhaustive list of the permitted pairings. This is straightforward: for each region, write down the inequalities which apply and calculate the truth tables. Consider, for example, the region

$$b < 0 \wedge (w_2 < -b/2) \wedge (w_1 + w_2 \geq -b/2) \wedge (w_1 > -b), \tag{3.72}$$

which is marked on the far right of Fig. 3.3a. To get the full truth tables for f and g, we need the full system of seven inequalities (four from Eq. 3.72, and three less constraining inequalities), which can be found by visual inspection of Fig. 3.3a. There are seven and not eight inequalities because the $b < 0$ or $b \geq 0$ inequality provides two entries in the truth table (Eq. 3.20). With the truth table entries they provide (from Table 3.2) they are:

$$\begin{cases} b < 0 & \implies f(0,0) = 0 \wedge g(0,0) = 0 \\ w_1 \geq -b & \implies f(1,0) = 1 \\ w_1 \geq -b/2 & \implies g(1,0) = 1 \\ w_2 < -b & \implies f(0,1) = 0 \\ w_2 < -b/2 & \implies g(0,1) = 0 \\ w_1 + w_2 \geq -b & \implies f(1,1) = 1 \\ w_1 + w_2 \geq -b/2 & \implies g(1,1) = 1 \end{cases} \tag{3.73}$$

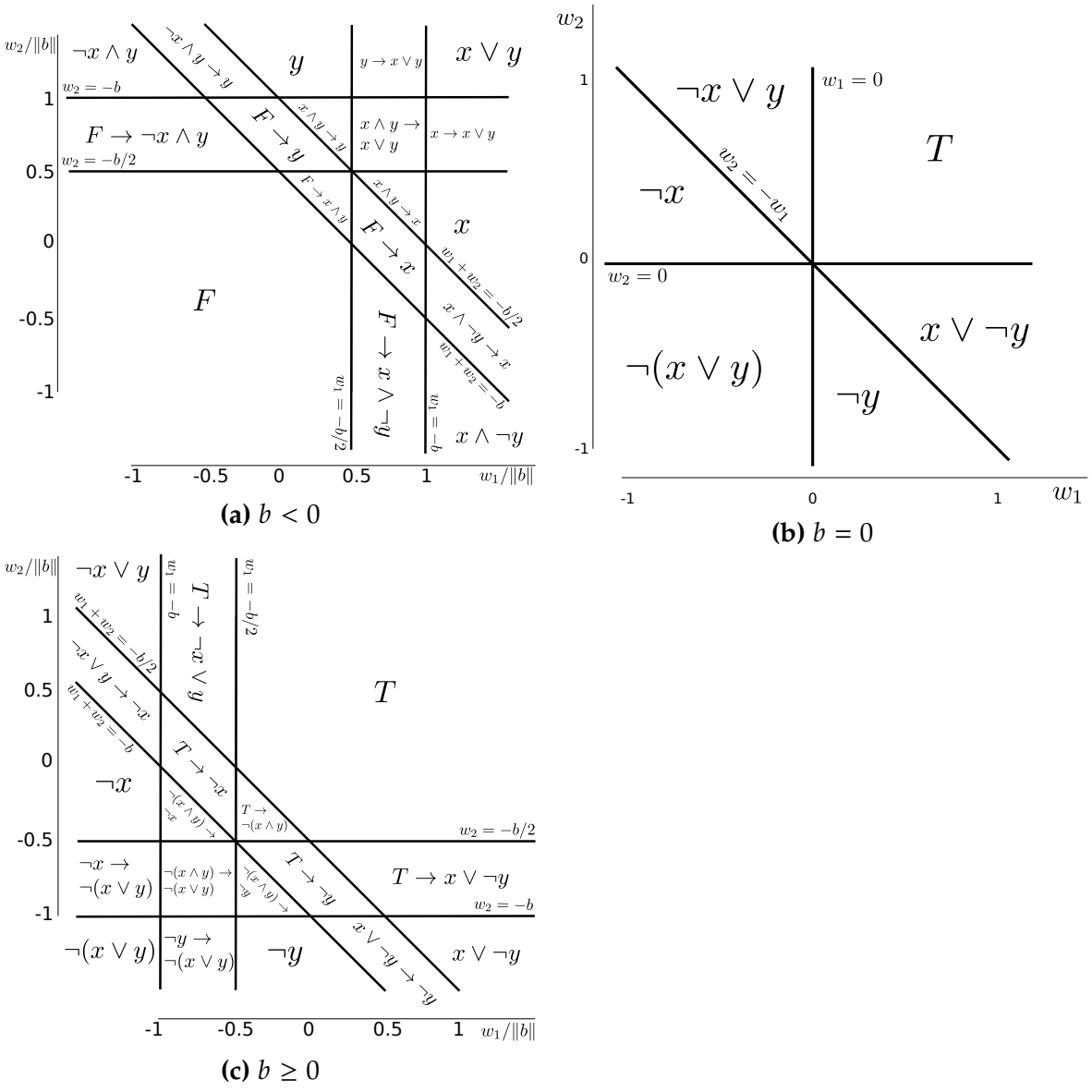
This gives the complete truth table shown in Table 3.4, which shows the pairing $x \to x$. This process yields the map of possible pairings in Fig. 3.4. We can then

Table 3.4: Truth table for the system of simultaneous equations in Eq. 3.73.

x	y	$f(x,y)$	$g(x,y)$
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1

mark the generated functions on a grid similar to that in Fig. 3.2, producing Fig 3.5, which shows an identical pattern to Fig. 3.2 and confirms those results. Additionally,

the areas occupied by the solutions in Fig. 3.4 are consistent with the experimental frequencies in Table 3.1 and Fig. 3.2.



(a) $b < 0$

(b) $b = 0$

(c) $b \geq 0$

Figure 3.4: Plots of function pairing by weight/bias ratio for negative, zero and positive biases.

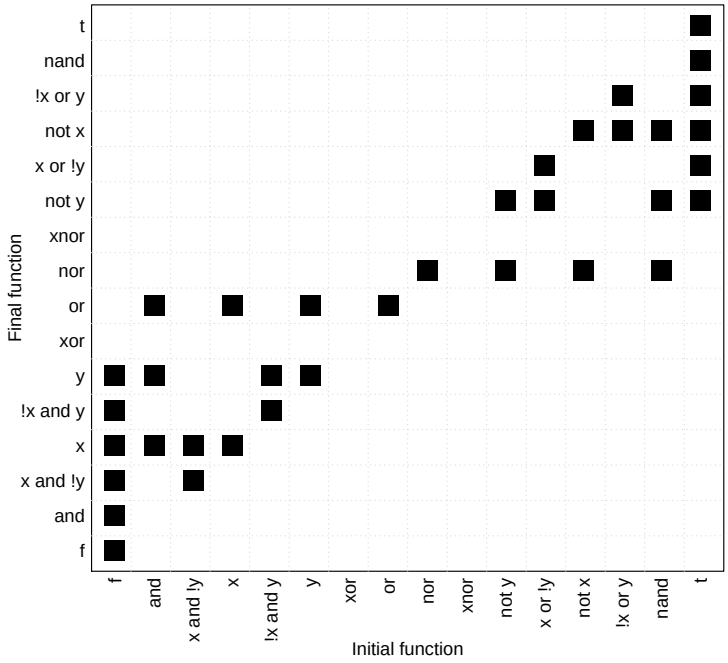


Figure 3.5: Possible UESMANN pairings in a single node, demonstrated by marking those pairings which appear in Fig. 3.4. Compare with Fig. 3.2.

3.1.3 Single-node UESMANN: a geometrical interpretation

It is also possible to view the permitted transitions geometrically, which may provide a more straightforward insight into how the nodes function. Given Eq. 3.5, the boolean threshold for nodes lies at

$$b + (h+1)(w_1 x + w_2 y) = 0 \tag{3.74}$$

$$\frac{b}{h+1} + w_1 x + w_2 y = 0 \qquad \text{(assuming } h > -1\text{)} \tag{3.75}$$

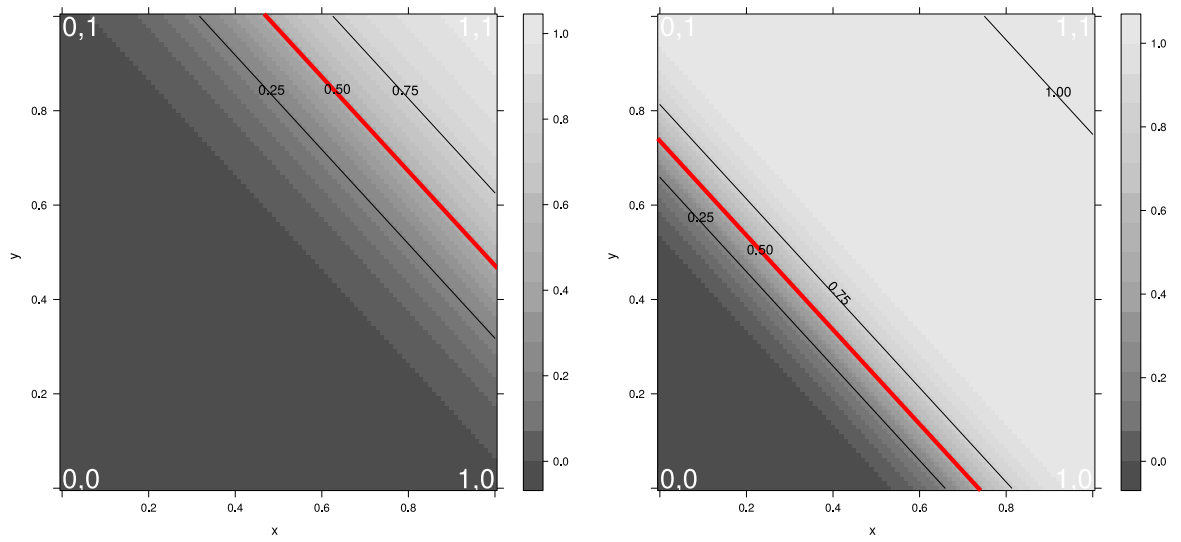
$$w_2 y = -\left(\frac{b}{h+1} + w_1 x\right) \tag{3.76}$$

$$y = -\left(\frac{b}{(h+1)w_2} + \frac{w_1}{w_2} x\right) \tag{3.77}$$

and hence the gradient of the threshold is given by $-\frac{w_1}{w_2}$, while the intercept is given by $-\frac{b}{(h+1)w_2}$. Thus the gradient of the threshold line remains constant under

modulation, while the intercept halves as h moves from 0 to 1. More generally, the modulation halves the bias of the node[4].

As a demonstration, a UESMANN node was extracted from the data generated in Sec. 3.1.1 which performed $x \wedge y \rightarrow x \vee y$ (AND to OR). This node has $b = -10.49, w_1 = 7.13, w_2 = 7.13$. Plotting the two thresholds against the inputs gives Fig. 3.6.



(a) $h = 0$ **(b)** $h = 1$

Figure 3.6: Output of a single UESMANN node, which performs $x \wedge y$ at $h = 0$ and $x \vee y$ at $h = 1$. The threshold (at which the activation function's output is 0.5) is shown in red. The pair of digits at each corner show the inputs at those points.

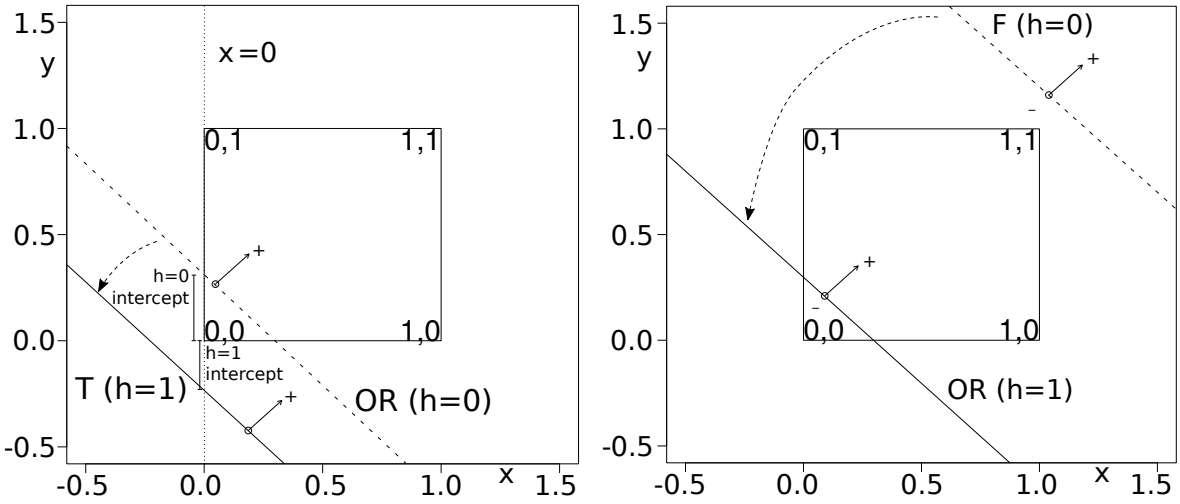
Thus increasing the modulator will always act to decrease the magnitude of the bias (since $h > 0$), moving the threshold towards the origin. We can therefore see why $x \wedge y \rightarrow x \vee y$ is possible, but not $x \vee y \rightarrow x \wedge y$: for the latter to occur, the magnitude of the bias (and the intercept) would have to be increased.

We can now consider the pairings not generated in Table 3.1 and Fig. 3.2 from a geometric standpoint. For example, the pairing $x \vee y \rightarrow T$ is prohibited because it would require the threshold line to change the sign of its intercept, as shown in Fig 3.7a. This is equivalent to our earlier contradiction, $(b < 0) \wedge (b \geq 0)$. Similarly, the false function F cannot transition to $x \vee y$, because that would require the situation

[4]Note that this says nothing about the nature of the "transition region" of the output of a sigmoid node as it rises from 0 to 1; inspection of Eq. 3.2 shows that this region will become narrower and steeper as the modulator increases (see also Fig. 3.12).

in Fig. 3.7b: the solid $h = 1$ line would need to cross the (0,0)-(0,1) and (0,0)-(1,0) line segments to produce $x \vee y$, but the dotted line would need to be entirely outside the box. This is not possible while maintaining the rule that the intercept is halved at $h = 1$. This is equivalent to the contradiction

$$(w_1 \geq -b/2) \wedge (w_2 \geq -b/2) \wedge (w_1 + w_2 < -b). \tag{3.78}$$



(a) $x \vee y \rightarrow T$ **(b)** $F \rightarrow x \vee y$

Figure 3.7: Examples of prohibited pairings shown geometrically. The arrowed dashed lines show the direction of the transition from $h = 0$ to $h = 1$. The other dotted and solid lines show the typical threshold at $h = 0$ and $h = 1$ respectively, with − to + arrows showing the direction of increasing activation (i.e. on the + side the output is true).

While it is possible to find geometric justifications for all impossible pairings, this would be redundant given the findings in Sec. 3.1.2. This brief look at the geometry of the function has, however, shown how UESMANN nodes work. It has also shown that this simple transition can perform a large enough subset of at least the boolean functions to be useful. The next section will extend the analysis to 2-2-1 networks (i.e. networks with 2 hidden nodes).

3.1.4 Networks with a single hidden layer of two nodes

We will now consider whether a UESMANN network with a hidden layer can represent all 256 possible boolean binary function pairings. First we will consider which pairings are possible if the three nodes in the network are all simple boolean nodes, i.e. the outputs of the nodes are thresholded at 0.5. In a true UESMANN

network, each node performs a function of its two inputs and the modulator $\mathbb{R} \times \mathbb{R} \times [0, 1] \to \mathbb{R}$, as given by Eq. 3.4. In Fig. 3.8, the three nodes perform the functions f, g and h, and each performs a single-node pairing as described in the previous section.

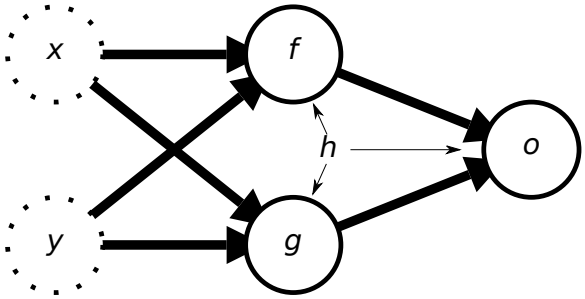


Figure 3.8: A 2-2-1 UESMANN network, i.e. with two inputs x and y, a hidden layer of two nodes f and g and a single output node o. The modulator here is given by h, and its action is shown by the thin arrows.

However, we will consider each node to be completely boolean, $\mathbb{B}^3 \to \mathbb{B}$ where $\mathbb{B} = \{0, 1\}$. This is equivalent to thresholding each node's output in the same way as Eq. 3.5. The output of the entire network is given by

$$u(x, y, m) = o(f(x, y, h), g(x, y, h), h) \tag{3.79}$$

so for any three function pairings f, g, o it is easy to determine the resulting pairing u. The table in Fig. 3.2 gives the "probability" $P(f)$ (0 or 1) of a function pairing $f(x, y, m)$ being representable as a UESMANN node. The probability of a given function pairing $u(x, y, m)$ can then be obtained by

$$P(u) = \sum_{u,f,g,o \in U} P(f)P(g)P(o)[u = o(f, g)] \tag{3.80}$$

where U is the set of all 256 possible functions. The Iverson bracket $[\cdots]$ gives 1 when the predicate inside is true and 0 otherwise [149], and the predicate is the condition in Eq. 3.79. Performing this calculation gives the result in Fig. 3.9. The probabilities are all very low with the exception of the $T \to T$ and $F \to F$ pairings, with many completely unrepresentable. However, it may be that the missing functions may be representable by using the hidden layer nodes without thresholding.

In order to test this, another Monte Carlo simulation was performed. Again, 10^{11} random data are used, but here the data are representations of 2-2-1 networks as shown in Fig 3.8. Thus each datum consists of nine floating point values (b, w_1 and w_2 for each node). The PRNG, floating point ranges and method are otherwise

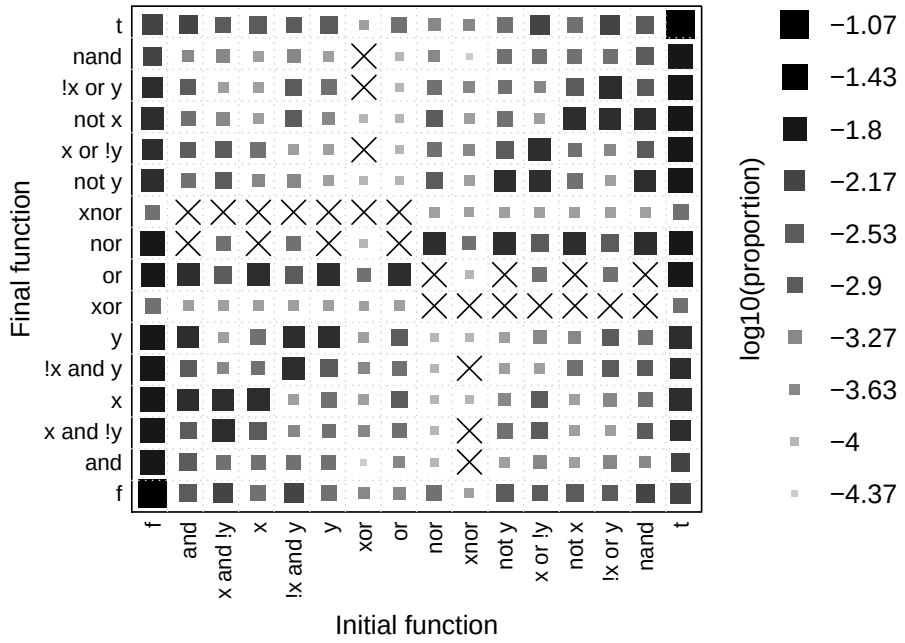


Figure 3.9: The cumulative probabilities for each putative pairing in a 2-2-1 pure boolean UESMANN system, given the data in Fig 3.2 for single nodes. For clarity, crosses mark those pairings for which the probability is zero: these cannot be represented by a UESMANN system with all nodes thresholded.

the same as in Sec. 3.1.1, but note that unlike in the above thresholding only takes place on the output node to determine the final boolean result of the network. The resulting function pairing counts are shown in Fig. 3.10.

Note that there are no crossed cells: UESMANN can represent pairings of every boolean function with 2 hidden nodes, the minimum required to represent any single boolean function in the same topology [7, Theorem 3.9][261][5]. Thus a UESMANN network is able to learn any two boolean functions using the same number of parameters as an MLP which can learn any single boolean. However, note the $\log_{10}$ scale of the plot: many of the pairings found are vanishingly rare, while the majority of the pairings involve the T and F functions. Table 3.5 shows the most common function groups generated: clearly very many networks generate true or false (≥ 0.5 or < 0.5) for all inputs at all hormone levels, accounting for 62.5% of the total. Networks which generate single functions with T or F for the other function

[5]However, it is possible to represent many, if not all, binary boolean functions in a network with a single hidden node if connections are permitted to skip layers [237]. We will not deal with such networks in this thesis.

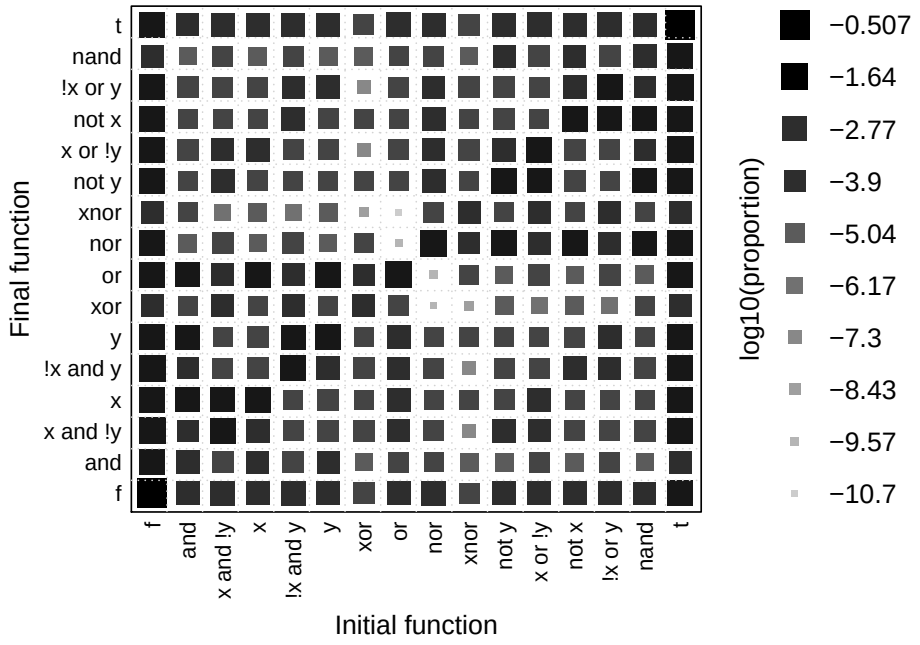


Figure 3.10: log_{10} of the proportion of pairings of binary boolean functions produced by 10^{11} random UESMANN 2-2-1 networks, when the output node is thresholded at 0.5. The size and tint of each square gives the $\log_{10}$ of the count.

are at 22.78% for both orders (i.e. whether the other function is $h = 0$ or $h = 1$ is not relevant). Networks which generate the same function for both modulator levels, excluding T and F networks, are 10.19% of the total.

Table 3.5: Subsets of boolean function pairs and their frequency in 10^{11} random 2-2-1 UESMANN networks.

Type	frequency (as % of total)
$f = F \wedge g = F$	31.14%
$f = T \wedge g = T$	31.14%
$(f = F \vee g = F) \wedge \neg(f = F \wedge g = F)$	11.40%
$(f = T \vee g = T) \wedge \neg(f = T \wedge g = T)$	11.40%
$(f = g) \wedge \neg(f = T \vee f = F)$	10.62%

Table 3.6 shows those pairings which cover a very small region of the weight space — the frequency here is shown as the actual proportion rather than the percentage, which would be hard to read for these small values. Values for which the count is less than 10000 are shown; the next "batch" of networks have counts greater than

40000. A histogram of $\log_{10}(count)$ for all pairings is shown in Fig. 3.11: note the bimodal distribution, with a small group of low-frequency pairings, no pairings with $\log_{10}(count) \approx 2$, and many with $5 < \log_{10}(count) < 9$.

Table 3.6: Rare functions (count less than 10000) in 10^{11} random UESMANN networks, with how often they occur and what fraction of the overall network count they comprise.

Type	count	frequency (as proportion of total)
$\neg(x \oplus y) \rightarrow \neg x \wedge y$	3500	3.5×10^{-8}
$x \oplus y \rightarrow x \vee \neg y$	3462	3.5×10^{-8}
$\neg(x \oplus y) \rightarrow x \wedge \neg y$	3437	3.4×10^{-8}
$x \oplus y \rightarrow \neg x \vee y$	3408	3.4×10^{-8}
$x \oplus y \rightarrow \neg(x \oplus y)$	84	8.4×10^{-10}
$\neg(x \oplus y) \rightarrow x \oplus y$	78	7.8×10^{-10}
$\neg(x \vee y) \rightarrow x \vee y$	17	1.7×10^{-10}
$x \vee y \rightarrow \neg(x \vee y)$	11	1.1×10^{-10}
$\neg(x \vee y) \rightarrow x \oplus y$	4	4.0×10^{-11}
$x \vee y \rightarrow \neg(x \oplus y)$	2	2.0×10^{-11}

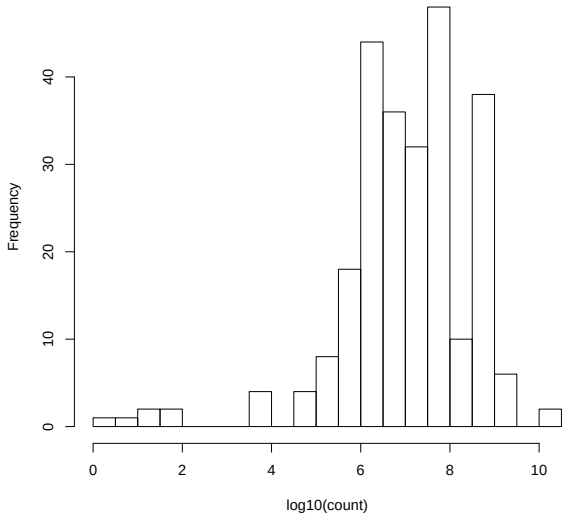


Figure 3.11: Histogram showing the frequencies of different values of $log_{10}(count)$ for different pairings occurring in 10^{11} random 2-2-1 UESMANN networks.

We would expect the uncommon pairings of Table 3.6 to be those which are unrepresentable by the fully-thresholded network, shown in Fig. 3.9, because values which are very large or small will be clipped by the nodes. Functions which require intermediate values, as these must, will have a smaller solution space. Visual

inspection of Figs. 3.9 and 3.10 with Table 3.6 shows that there is some degree of correlation.

We have shown that a 2-2-1 UESMANN network is able to represent any boolean pairing, but that some pairings (such as $x \vee y \rightarrow \neg(x \oplus y)$, OR to XNOR) are vanishingly rare — occupying 2×10^{-11} of the solution space.

3.2 Other forms of modulation

UESMANN uses weight modulation, that is:

$$y = \sigma\left(b + \sum_i w_i x_i (1 + h)\right). \tag{3.2}$$

Other forms of modulation are also possible, and we shall briefly look at two. Firstly, weight and bias modulation:

$$y = \sigma\left((1 + h)(b + \sum_i w_i x_i)\right). \tag{3.81}$$

Here, both the bias and the weight are multiplied by $(1 + h)$. This has the effect of multiplying the entire input to the sigmoid activation function by $1 + h$. Looking at this in terms of the threshold, it is clear that both the slope and intercept of the threshold line will remain unchanged. Therefore, operating as a thresholded boolean node, a single node cannot transition to any other function. However, the output will change shape, as shown in Fig: 3.12. It may be that this change causes hidden nodes to saturate under modulation which would not otherwise, which would lead to different functions being performed in the output node. To test this, once again 10^{11} random networks with the same parameters as those for Fig. 3.10 were generated, this time performing Eq. 3.81. We should expect to see some pairings represented, but without the "boolean mode" of operation seen in Fig. 3.9 they should be far rarer. Fig. 3.13 shows the results — most pairings are represented, but far more are rare. The pairings $x \oplus y \rightarrow \neg(x \oplus y)$ and its negation/inverse are not represented at all. The commonest pairings, as we would expect, are from functions to themselves.

Another form of modulation is to modulate the bias rather than the weight. This gives

$$y = \sigma\left((1 + h)b + \sum_i w_i x_i\right). \tag{3.82}$$

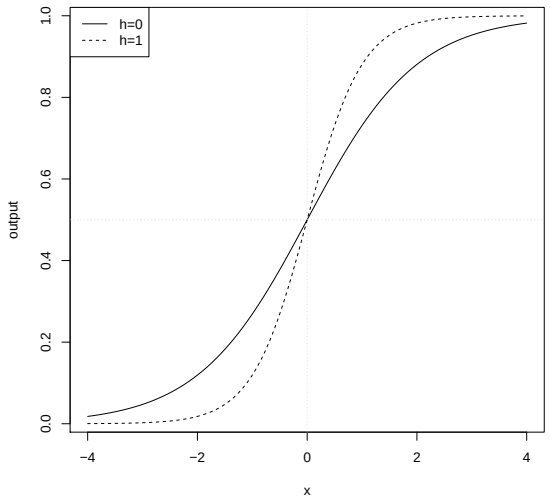


Figure 3.12: Logistic sigmoid $\frac{1+h}{1+e^{x}}$ for $h = 0$ and $h = 1$.

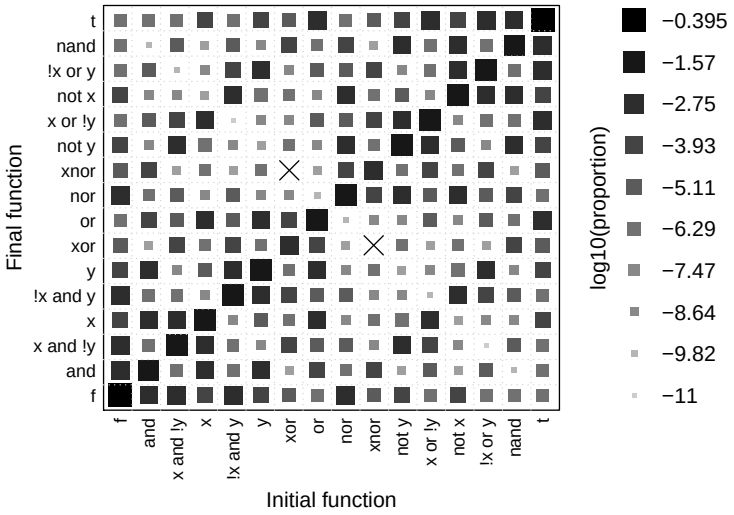


Figure 3.13: log_{10} of the proportion of pairings of binary boolean functions produced by 10^{11} random 2-2-1 networks with weight and bias modulation as in Eq. 3.81, when the output node is thresholded at 0.5. The size and tint of each square gives the $\log_{10}$ of the count. Where there is a cross, the count is zero: these are function pairings which never appear. Note that the square size and tint are scaled to the range seen in the data once the zeroes have been removed.

This has the effect of doubling the intercept under modulation, rather than halving it. We predict this should have roughly the same effect as UESMANN's weight modulation. The results for a 2-2-1 network are shown in Fig. 3.14. This looks rather

similar to the pattern in Fig. 3.9 — to see the difference in distribution more clearly, kernel density estimates of both UESMANN (weight only) and the bias-only method were plotted using R's `density` function, giving the result in Fig. 3.15. The results for modulation of weight and bias are also included.

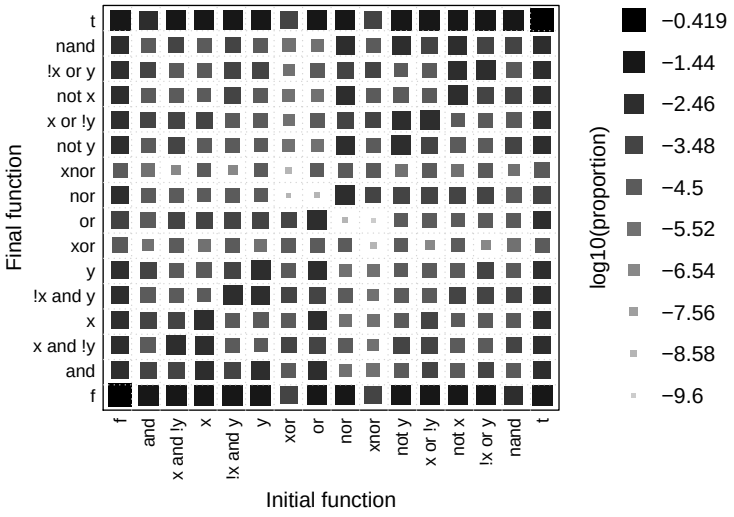


Figure 3.14: log_{10} of the proportion of pairings of binary boolean functions produced by 10^{11} random 2-2-1 networks with bias modulation as in Eq. 3.82, when the output node is thresholded at 0.5. The size and tint of each square gives the $\log_{10}$ of the count.

We would probably prefer a good spread in the distribution with fewer pairings having a low probability. We can see from Fig. 3.15 that if the weight and bias are both modulated, the distribution is wide but weighted towards the low end (and we know that certain pairings are impossible from Fig. 3.13: these are omitted from the density plot because $log_{10}(0)$ is not defined). UESMANN's weight-only modulation appears to give a good distribution, perhaps slightly better than bias-only. Therefore we will continue with this paradigm.

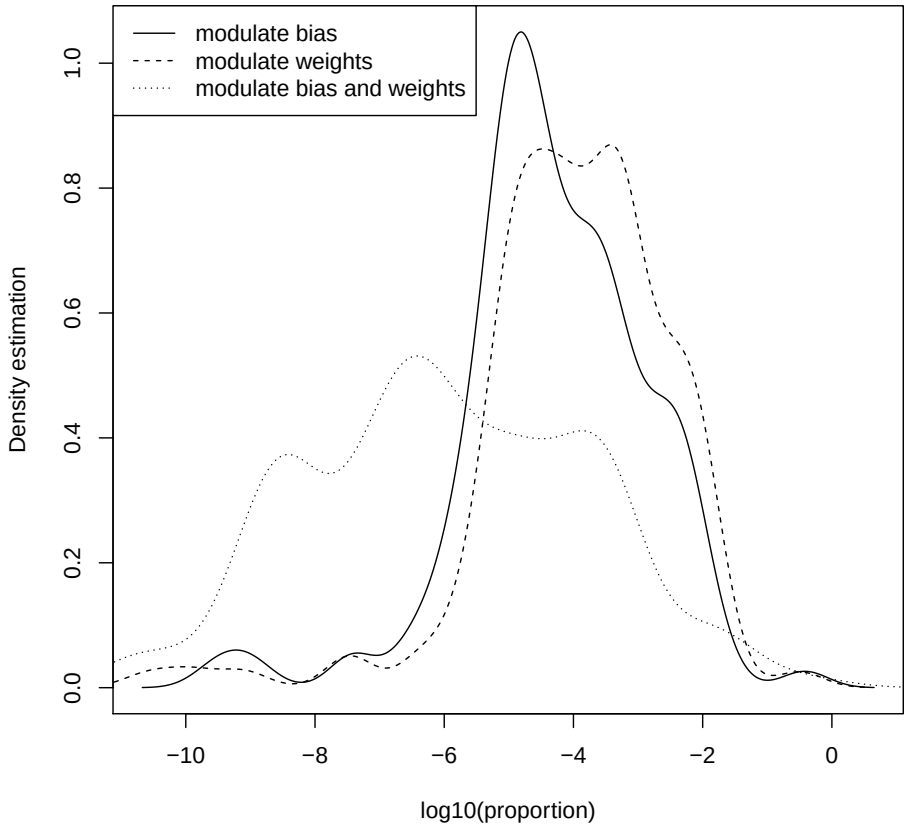


Figure 3.15: Kernel density estimates for log_{10} of the proportion of pairings of binary boolean functions found in 10^{11} random 2-2-1 networks for both weight-only, bias-only, and weight-and-bias modulation.

Chapter 4

Training UESMANN using back-propagation

As we have seen, UESMANN appears to be able to express a large variety of function pairings — indeed, all possible pairings of the binary boolean functions can be represented by a network with the same topology as the minimum strict[1] feed-forward network required to represent any single boolean function. However, many of the boolean function pairings we see in the Monte Carlo simulations of 2-2-1 networks in Fig. 3.10 and Table 3.6 are have vanishingly small counts, with functions such as $x \vee y \rightarrow \neg(x \oplus y)$ taking up a volume of the order of 10^{-11} of the solution space. This suggests that many solutions for other problems will also take up small areas of the solution space (if they exist at all).

It would be useful to find a training method to evaluate UESMANN in more complex problems. There are several possibilities, as discussed in Chapter 2: phylogenetic techniques such as particle swarm optimisation [74] and the genetic algorithm [125] might produce useful results in a reinforcement learning setting without the need to perform complex temporal credit assignment, but this would require embedding in a full artificial endocrine system, and findings may not transfer to other applications. If we look at just the network itself, reinforcement through Hebbian learning might work, but is likely to involve considerable complexity [221].

Instead, it may prove more profitable to study this novel network by following the path taken by early network pioneers: developing a gradient descent technique. Gradient descent methods are easy to understand and implement, and may give insights into the topology of the error surfaces involved, but are a supervised learning

[1] "Strict" is used here to mean "no connections skip the hidden layer": it is possible to perform any function in a network with a single hidden layer, if connections directly from the inputs to the output layer are possible [237].

paradigm. This is not ideal from the point of view of adaptivity, but is still useful for many applications.

4.1 Back-propagation of errors

Our starting point will be the well-known back-propagation method, developed independently by Werbos [298] and later by Rumelhart, Hinton and Williams [237]. This method is in turn a generalisation and extension of the delta rule for updating neurons in a single-layer network, developed by Widrow and Hoff [300]. Given a single example of inputs and outputs, a single back-propagation update modifies the weights and biases in the network to reduce the mean-squared error between the actual outputs and the example outputs $\frac{1}{n}\sum_{i=0}^{n}(y_i - a_i)^2$ (where for the given example inputs a_i is the actual output indexed by i and y_i is the example output indexed by i). Each update involves calculating the gradient of the error with respect to the each weight, and then subtracting some small fraction of that value from the given weight such that it moves "downhill": hence "gradient descent." For the output nodes this is the delta rule, for the hidden nodes the gradients can be derived from the error at the output nodes (hence "back-propagation of errors"). This method has already been covered briefly in Sec. 2.3.6; we now will discuss some of the issues and choices involved in its implementation.

4.1.1 Batching or stochastic training?

In order to train a network to generalise from a set of examples, it needs to be trained on a large number of such examples. Therefore back-propagation needs to be run a large number of times, and the results combined so that the weights converge on a minimum which represents all the examples satisfactorily. This is typically done in one of three ways:

- *Batching*: for each iteration (i.e. each run through the set of examples) the gradients are calculated for all weights, for all examples in the set. The means of the gradients are found and applied at the end of the iteration. This generally gives a smoother approach to a minimum: the true gradient as given by all the examples is being followed.

- *Stochastic*: the gradients are calculated and then immediately applied for each example in the set, with the examples typically being shuffled randomly before each iteration (hence "stochastic"). This results in a "noisy" approach to the

minimum, since each gradient vector produced will be in a different direction, which can sometimes avoid local minima. Stochastic gradient descent (SGD) is typically much faster than batched gradient descent, particularly for large example sets, because the latter must find the mean of all examples' gradients at each step [303]. SGD only needs one gradient to perform a step. SGD is also better at avoiding local minima in non-convex functions [97]. It is, however, sensitive to feature scaling: ideally, inputs and outputs should be normalised to [0,1] (for the logistic sigmoid we are using). Additionally, SGD is prone to overfitting in "normal" neural networks, and there is no reason to think this would not also apply in UESMANN. To avoid this, some form of regularisation is often performed: an example is L2 regularisation of the cost function (mean squared error), which can be shown to be equivalent to applying a decay term to the weight in each update (penalising large weights)[2]. Several regularisation techniques are explored in [166]. However, none will be used for these initial explorations into UESMANN beyond some normalisation of the example values.

- *Mini-batching* is a commonly used compromise between batching and SGD. The examples are split into a number of equally sized small batches, with the mean gradient being used in each one. This combines a smoother approach to the minimum with a lower computational complexity, and is amenable to parallelisation: batches can be processed in parallel.

Rather than using batching or mini-batching we will be using stochastic gradient descent in UESMANN:

- Some of the more complex problems we are studying may have non-convex error surfaces (i.e. with local minima), which SGD has been shown to be more effective at optimising [97]. This is particularly true with UESMANN, given that we are optimising for two functions (see Eq. Sec. 4.4, p. 94).
- SGD is typically much faster than batching, because of the need in batching to find the mean of the gradient for all examples for each update [303].
- We may find that the "jitter" as UESMANN oscillates between the gradients for the two functions (see Sec. 4.2 below) helps avoid local minima.
- It is the simplest method to implement and understand.

Mini-batching was not used because it is essentially a compromise between batching and SGD which introduces an extra hyperparameter: the batch size.

[2] L2 regularisation can be derived by assuming a Gaussian prior over the network parameters [230].

4.1.2 Hyperparameters

Hyperparameters are those parameters fixed by the experimenter which control learning, rather than the parameters which are learned by the system (in this system the weights and biases). Back-propagation has three main hyperparameters: the number of hidden nodes, the learning rate and the initial values. Until recently, hidden nodes were usually limited to a single hidden layer because of the vanishing gradient problem (see below). This layer learns an intermediate representation of the data presented in the input layer, which is then processed by the output layer. If the number of hidden nodes is too low, it may not be capable of representing the distinct features required to give a solution in the output. If it is too high, the topology of the solution space may become unnecessarily complex and lead to local minima, which manifest as convergence to poor solutions or long training times. Overfitting may also result: the network may work well on the data with which it has been trained, but may not generalise to other data (such as the test data typically "held out" from the training data).

It is impossible to choose a hidden node count without carefully considering the problem: the complexity of the function to be learned, the amount of noise in the data, the number of training cases and so on. Many sources describe "rules of thumb", but these are all nearly worthless [241]. Most practitioners start somewhere between the number of input and output units, depending on the complexity of the problem, and try a number of different values.

The other hyperparameters also require trial and error: lower learning rates take longer to converge to a solution and may get trapped in local minima, while higher learning rates are unlikely to find an accurate solution. The initial values for the weights need to be random, far enough apart that the trajectories of the solutions move away from each other, and yet small enough that sigmoid activation functions do not saturate, leading to small derivatives (see Sec 4.1.4 below). A common rule of thumb, or at least starting point, is Bishop's Rule: weights in the range $[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$ where n is the number of inputs to the node (i.e. the number of nodes in the previous layer) [26]. If the inputs are not normalised to the range [0,1] this may be insufficient to prevent saturation.

4.1.3 Early stopping and restarting

In all cases, the initial starting point is often important: it is likely that the gradient from many regions of the error surface will lead to local minima. For this reason, runs which fail to converge to a sufficiently good solution should be restarted with new

initial values. Convergence here simply means that the error function is no longer decreasing because the algorithm has found a local minimum. It is worth noting that with stochastic gradient descent, the error will not decrease monotonically: the solution is being "pulled" in different directions around the solution space, since each update is coming from a different example, but the net result is a decrease. Therefore measuring when to restart can be a difficult problem: in a complex solution space, the error may remain static (or increase) for many iterations before finding a "valley" in the solution space out of the local minimum. UESMANN may be particularly prone to such problems because its error surfaces are likely to be more complex.

4.1.4 Problems with sigmoid activation functions

UESMANN in its current form uses the logistic sigmoid as its activation function, although the training method is easily adapted to any other (mostly[3]) differentiable activation function. This is to permit comparison with the first neural networks trained by back-propagation, which also used the logistic sigmoid. However, while the "standard" sigmoid activation functions (the logistic sigmoid and the hyperbolic tangent) are convenient mathematically, having simple derivatives, they have a number of known problems:

- If a given neuron has an output which is close to 0 or 1 for the logistic sigmoid (-1 or 1 for the hyperbolic tangent), the derivative of the activation will be close to zero. This causes the error correction term to drop to close to zero. Because of this, the network can find itself in incorrect local minima, or can reach a state of "network paralysis." This can be somewhat counter-intuitive: in human learning, a very large error produces a large correction; in the back-propagation algorithm, a very large error might produce near-zero correction because the output is "saturated."

- Back-propagation works by finding the derivative of the error function with respect to each weight at the output layer, and applying to each weight a correctional proportional to this derivative. This error is then propagated back by applying the chain rule. Given that the range of the activation function is typically (-1,1) or (0,1), the derivatives will be much smaller than 1. This results in the multiplication of small numbers, which propagate back — getting smaller at each layer until they become vanishingly small. The output layers therefore

[3]Some activation functions are not completely differentiable, notably the rectified linear unit (ReLU) which is not differentiable at zero.

train much faster than the inner layers, which is why deep neural networks can be very difficult to train efficiently [21, 22][4].

If alternative activation functions are used in which the derivative can take values greater than 1, the related *exploding gradient problem* may occur. Here, the corrections become larger as the calculations move away from the output layer.

4.1.5 Alternative activation functions

In some deep learning systems, piecewise linear ramp activation functions such as the rectified linear unit (or ReLU) $f(x) = \max(0, x)$ are also used. While not completely differentiable (i.e. at 0) they are extremely efficient and do not saturate [157]. These avoid the vanishing gradient problem, but a ReLU unit may "die" if its parameters are updated such that the summed inputs are always less than zero. In this case, the gradient is zero and no learning can take place — a dead ReLU is unlikely to recover. This can result in networks with many nodes outputting zero. Variants of ReLU exist, such as the "leaky ReLU," in which the below-threshold output still has a small positive gradient [179]; and the "noisy ReLU": $f(x) = \max(0, x + Y)$, where Y is a source of Gaussian noise [205]. ReLU is currently the most popular activation function in deep networks [167].

4.1.6 Other enhancements to back-propagation

A large amount of work has been done on adding features to the basic algorithm to allow it to converge faster and to better minima. Indeed, the seminal paper by Rumelhart, Hinton and Williams [237] features the first of these enhancements: momentum, which permits the algorithm to "roll over" small local minima. Others include an adaptive learning rate [293], which modifies the learning rate according to local error surface topography; and weight decay, which was introduced to aid interpretation of weights post-training [120] but may aid generalisation and reduce the effects of noisy inputs [158]. It is also possible to use the Levenberg-Marquardt algorithm — a numerical optimisation technique for minimising least-squares error — to perform back-propagation[108].

Regularisation has briefly been mentioned above, and is used to avoid overfitting. L1 regularisation adds the absolute magnitude of a parameter to that parameter's

[4]This also applies to recurrent neural networks trained by back-propagation through time (see Sec. 2.3.7.2).

cost function, while L2 regularisation adds the square of the parameter. These both have the effect of reducing weights and biases, leading to a less complex model.

4.2 Back-propagation updates in UESMANN

UESMANN is an unusual network in that we wish to minimise two error functions: the first for $h = 0$ with the weights at their nominal values, and the second for $h = 1$ with the weights doubled (since at $h = 1$ the summand of Eq. 3.2 — the UESMANN node equation — becomes $2w_i x_i$). Thus we need to find the gradients with respect to the weights w for the examples for which $h = 0$, and the gradients wrt. the doubled weights $2w$ for the examples for which $h = 1$, and apply them both. In essence, we are following two error surfaces and attempting to find a compromise between them.

In this section, we will use the convention in Fig. 4.1 for labelling weights, biases, and activations and errors within a UESMANN node. Writing Eq. 3.2 in this form

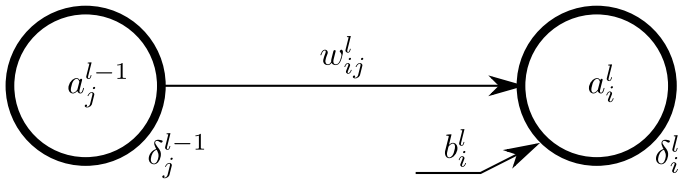


Figure 4.1: Neural network labelling conventions: a_i^l is the activation of node i in layer l, w_{ij}^l is the weight between node j in layer $l-1$ and node i in layer l, b_i^l is the bias on node i in layer l, and δ_i^l is the output error of node i in layer l (i.e. the difference between the example output and the actual output).

we obtain:

$$a_i^l = \sigma\left(b_i^l + \sum_j (h+1) w_{ij}^l a_j^{l-1}\right) \tag{4.1}$$

where σ is the standard logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$. Ideally, we are attempting to find a single set of weights and biases $(\mathbf{w}, \mathbf{b})$ for a network which performs the function pairing $f \rightarrow g$ such that

$$\arg\min_{(\mathbf{w},\mathbf{b})} C_f(\mathbf{w}, \mathbf{b}) = \arg\min_{(\mathbf{w},\mathbf{b})} C_g(2\mathbf{w}, \mathbf{b}), \tag{4.2}$$

where $\arg\min_x f(x)$ ("argument of the minimum of $f(x)$ with respect to x") is the value of x at which $f(x)$ is at its global minimum, $C_f(\mathbf{w}, \mathbf{b})$ is the cost (error) function for the given weights and biases with respect to the function $\mathbf{y} = f(\mathbf{x})$, and C_g is similarly the error function for $g(\mathbf{x})$. These functions are presented to the network

during training as two sets of examples X_f, X_g mapping $\mathbf{x}$ to $\mathbf{y}$. An example of a cost function — and the one used in this work — is the sum of squared errors:

$$C_f(\mathbf{w}, \mathbf{b}) = \sum_{(\mathbf{x},\mathbf{y})\in X_f} \|u(\mathbf{x}, \mathbf{w}, \mathbf{b}) - \mathbf{y}\|^2. \tag{4.3}$$

This is the squared Euclidean distance between the required output vector $\mathbf{y}$ of the network as given in the training examples X_f, and the function actually performed by the network u for the corresponding input vector $\mathbf{x}$, with weights and biases $(\mathbf{w}, \mathbf{b})$.

If we consider the parameter space of the network (i.e. the weight space, if biases are considered special weights), we are trying to find a point in that space which performs one function and which, when doubled in the weight dimensions, performs a second function. Figure 4.2 shows this idea in a 3D form, simplifying the weight space down to two dimensions. There are two distinct error surfaces, one for each function, here shown as heat maps. We are trying to find a point $\mathbf{w}$ which minimises the error on one surface, while the point $2\mathbf{w}$ minimises the error on the other surface. In the figure, the value of $\mathbf{w}$ gives a low error for the $h = 0$ function, but a much higher error for the $h = 1$ function where $2\mathbf{w}$ is used as the weight vector.

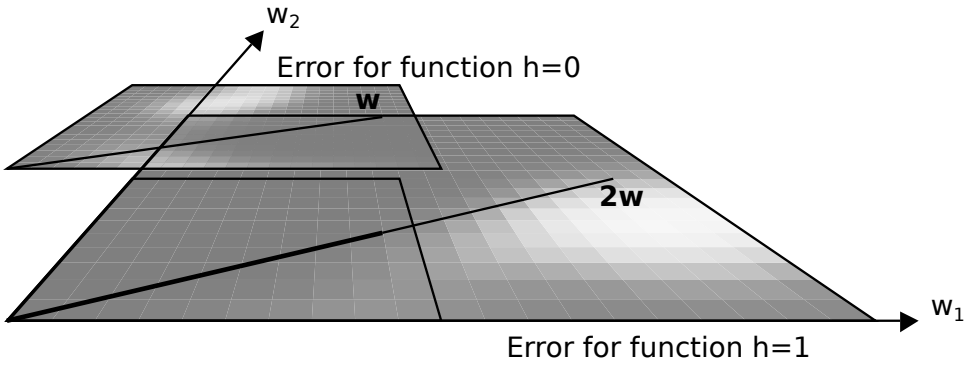


Figure 4.2: Representation of slices through the error/weight spaces for two functions: the functions are merely for illustrative purposes and do not represent actual functions. The brightness indicates the error (bright indicates high error), w_1 and w_2 are two weight axes.

It is clearly unlikely for most function pairs that we will find a true global minimum for both, i.e. a value for $(\mathbf{w}, \mathbf{b})$ which satisfies Eq. 4.2. However, we can at least find local minima for

$$\arg\min_{(\mathbf{w},\mathbf{b})}(C_f(\mathbf{w}, \mathbf{b}) + C_g(2\mathbf{w}, \mathbf{b})) \tag{4.4}$$

4.2.1 The UESMANN equations

In order to train a UESMANN network, the back-propagation update functions must be modified to handle the parameter h. We will not derive the original form of the functions — this can be found in [237], and involves deriving the generalised delta rule for the output layer and applying the chain rule for the hidden layer(s) — but we will state them in Eqs. 4.5-4.8, in scalar rather than vector form:

$$\frac{\partial C}{\partial w_{ij}^l} = a_j^{l-1}\delta_i^l \qquad \text{(error surface gradient wrt. weight)}, \tag{4.5}$$

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \qquad \text{(error surface gradient wrt. bias)}, \tag{4.6}$$

$$\delta_j^L = (a_j^L - y_j) \cdot a_j^L \cdot (1 - a_j^L) \qquad \text{(error in output layer)}, \tag{4.7}$$

$$\delta_j^l = a_j^l(1 - a_j^l) \sum_i w_{ij}^{l+1}\delta_i^{l+1} \qquad \text{(error in hidden layer)}. \tag{4.8}$$

Here, C represents the mean squared error (or cost), w_{ij}^l and b_i^l are the weights and biases as defined in Fig. 4.1, a_j^l and δ_j^l are the activation and error of node j in layer l respectively, and L represents the final output layer. A back-propagation update in the stochastic paradigm involves feeding each example into the network and finding δ_j^l for each node, and using this with the node activations to find $\frac{\partial C}{\partial w_{ij}^l}$ and $\frac{\partial C}{\partial b_i^l}$. These cost gradients are then applied to the weights and biases, so that they move down the gradient towards a (local) minimum in C:

$$w_{ij}^l \leftarrow w_{ij}^l - \eta\frac{\partial C}{\partial w_{ij}^l}, \tag{4.9}$$

$$b_i^l \leftarrow b_i^l - \eta\frac{\partial C}{\partial b_i^l} \tag{4.10}$$

where η is the learning rate: a multiplicative factor applied to the gradient to produce the correction in each update. Note that in Eq. 4.5 the gradient is determined with respect to the weight: for UESMANN, the actual weight is $(1 + h)w_{ij}^l$ (Eq. 3.2). Therefore Eq. 4.5 must be modified:

$$\frac{\partial C}{\partial w_{ij}^l(1 + h)} = a_j^{l-1}\delta_i^l \qquad \text{(Eq.4.5, substituting the modulated weight)}. \tag{4.11}$$

$$\frac{\partial C}{\partial w_{ij}^l} = (1 + h)a_j^{l-1}\delta_i^l \tag{4.12}$$

This gives our final modified form of Eq. 4.5. Similarly for Eq. 4.8:

$$\begin{aligned}\delta_j^l &= a_j^l(1-a_j^l)\sum_i (1+h)w_{ij}^{l+1}\delta_i^{l+1} \\ &= a_j^l(1-a_j^l)(1+h)\sum_i w_{ij}^{l+1}\delta_i^{l+1}.\end{aligned}$$

This gives the full set of equations for updating a UESMANN network for examples at modulator level h:

$$\frac{\partial C}{\partial w_{ij}^l} = (1+h)a_j^{l-1}\delta_i^l \qquad \text{(error surface gradient wrt. weight)}, \quad (4.13)$$

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \qquad \text{(error surface gradient wrt. bias)}, \quad (4.14)$$

$$\delta_j^L = (a_j^L - y_j)\cdot a_j^L \cdot (1-a_j^L) \qquad \text{(error in output layer)}, \quad (4.15)$$

$$\delta_j^l = a_j^l(1-a_j^l)(1+h)\sum_i w_{ij}^{l+1}\delta_i^{l+1} \qquad \text{(error in hidden layer)}. \quad (4.16)$$

4.2.2 Stochastic gradient descent in UESMANN

Given the discussion above of how gradient descent is performed (stochastically, batching, and mini-batching), there are several ways to proceed given that we have two sets of examples, one for each function to be learned. Stochastic gradient descent will be used here because it is faster, conceptually simpler, and the "jitter" inherent in stochastic gradient descent (SGD) may help the system recover from local minima. Bishop's "rule of thumb" for determining the range of the random initial weights will be used [26, p.262], [166]: the weights and bias in each node will be initialised to uniformly distributed random values in the interval $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$ where n is the number of weights feeding into the node.

We assume that examples are provided to the system as tuples of vectors $(\mathbf{x}, \mathbf{y}_0, \mathbf{y}_1)$. Each example consists of a set of inputs and two sets of outputs, one for each modulator level $h = 0$ and $h = 1$. The algorithm will proceed by shuffling the examples, and for each example training with first the $h = 0$ output, and then the $h = 1$ output. Thus the solution will follow the gradients of the two error surfaces alternately. Each example's update will be referred to as a "pair presentation", while each pass through the entire set of examples will be referred to as an "iteration". Algorithms 1 to 5 describe the algorithm in full, in which the following hold:

- Weights, biases and error terms are as in Fig. 4.1.
- $h \in \{0, 1\}$ is a modulator level.

- Layers are indexed from 1 to L. The activations for an entire layer l are notated as the vector $\mathbf{a}^l$.
- a_i^0 holds the inputs indexed by i. The entire layer is notated as $\mathbf{a}^0$.
- The number of nodes in layer l is n_l.
- E is a set of examples each consisting of tuples of vectors $(\mathbf{x}, \mathbf{y}_0, \mathbf{y}_1)$, where $\mathbf{x}$ is the input and $\mathbf{y}_0$ and $\mathbf{y}_1$ are the required outputs at $h = 0$ and $h = 1$ respectively.
- η is the learning rate.

Algorithm 1 UESMANN-backprop algorithm, using Bishop's Rule for the initial weights

$N \leftarrow$ new network containing 1 hidden layer
for all layers l **do**
  for all weights w_{ij}^l entering layer l **do**
    $w_{ij}^l \leftarrow U\left(\frac{-1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right)$ where d is the number of weights entering the node and $U(p, q)$ is a uniformly distributed random number in the range $[p, q]$
  end for
  for all biases b_i^l in layer l **do**
    $b_i \leftarrow U\left(\frac{-1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right)$ (d, U defined as above)
  end for
end for
for $i = 0$ **to** *MaxIterations* **do**
  Shuffle E into a random order
  for all $(\mathbf{x}, \mathbf{y}_0, \mathbf{y}_1) \in E$ **do**
    $h \leftarrow 0$
    present example $(\mathbf{x}, \mathbf{y}_0)$ to Algorithm 2
    $h \leftarrow 1$
    present example $(\mathbf{x}, \mathbf{y}_1)$ to Algorithm 2
    Perform any cross-validation and/or early stopping (see Sec. 4.2.3).
  end for
end for

4.2.3 Cross-validation

It is, of course, important that our algorithm is not tested with the training data (although this is not possible with the boolean functions). In order to achieve this, a number of test examples are held back from training (typically a ratio of 1:5). Cross-validation during training is performed by slicing this test data into a number of equal slices and running the network periodically during training on each example

Algorithm 2 UESMANN-backprop single training iteration

Require: $\mathbf{x}$ is the example input
Require: $\mathbf{y}$ is the example output
Require: h is the modulator level
  $\mathbf{a}^0 \leftarrow \mathbf{x}$ {Set the input layer from the example input}
  $\mathbf{a}^{L-1} \leftarrow \mathbf{y}$ {Set the output layer from the example input}
  Perform Algorithm 3 to run the network forwards
  Perform Algorithm 4 to calculate the errors
  Perform Algorithm 5 to update the weights

Algorithm 3 UESMANN-backprop update. This algorithm is used either to run the network once trained, or produce candidate results during training.

Require: $\mathbf{a}^0$ contains the inputs
Require: h is the modulator
Ensure: a^L contains the outputs
  for $l = 1$ **to** L **do**
    for $i = 1$ **to** n_l **do**
      $a_i^l \leftarrow \sigma\left(b_i^l + (1+h)\sum_j w_{ij}^l a_j^{l-1}\right)$ {Eq. 4.1}
    end for
  end for

in the slice. This slicing is done to ensure the entire test set is used while limiting the size of the test for performance purposes, so that they can be done frequently enough to show convergence without sacrificing too much training performance. We will discuss the actual cross-validation parameters separately in each experiment.

At this point it is worth mentioning that although during training the networks' weights typically converge — asymptotically approach a given point in weight space — in this thesis convergence is evaluated by considering how the error, not the weights, change over time. The one maps to the other: if the weights are not changing,

Algorithm 4 UESMANN-backprop error calculation, performed after Algorithm 3 during training. The vector $\mathbf{y}$ contains the required outputs from the example.

  {Calculate errors, output layer}
  for $i = 0$ **to** n_L **do**
    $\delta_i^L \leftarrow (a_i^L - y_i) \cdot a_i^L \cdot (1 - a_i^L)$ {Eq. 4.15}
  end for
  {Calculate errors, inner layers, propagating the errors back}
  for $l = L - 1$ **to** 0 **do**
    for $j = 1$ **to** n_l **do**
      $\delta_j^l = a_j^l(1 - a_j^l)(1+h)\sum_j w_{ij}^{l+1}\delta_i^{l+1}$ {Eq. 4.16}
    end for
  end for

Algorithm 5 UESMANN-backprop weight/bias update step, performed after Algorithm 4 during training

for $l = 1$ **to** L **do**
  for $i = 1$ **to** n_l **do**
    for $j = 1$ **to** n_{l-1} **do**
      $m_w \leftarrow (1 + h)a_j^{l-1}\delta_i^l$ {Eq. 4.13}
      $w_{ij}^l \leftarrow w_{ij}^l + \eta m_w$
    end for
    $m_b \leftarrow \delta_i^l$ {Eq. 4.14}
    $b_i^l \leftarrow b_i^l + \eta m_b$
  end for
end for

the errors are not changing. However, because of the nature of the dynamical system the weights are following, a small weight change may occasionally cause a large change in error, and this should be borne in mind. In most cases the error is measured as the mean squared error at the output layer, which may itself be averaged over a set of validation examples (as in Chapter 5 and later).

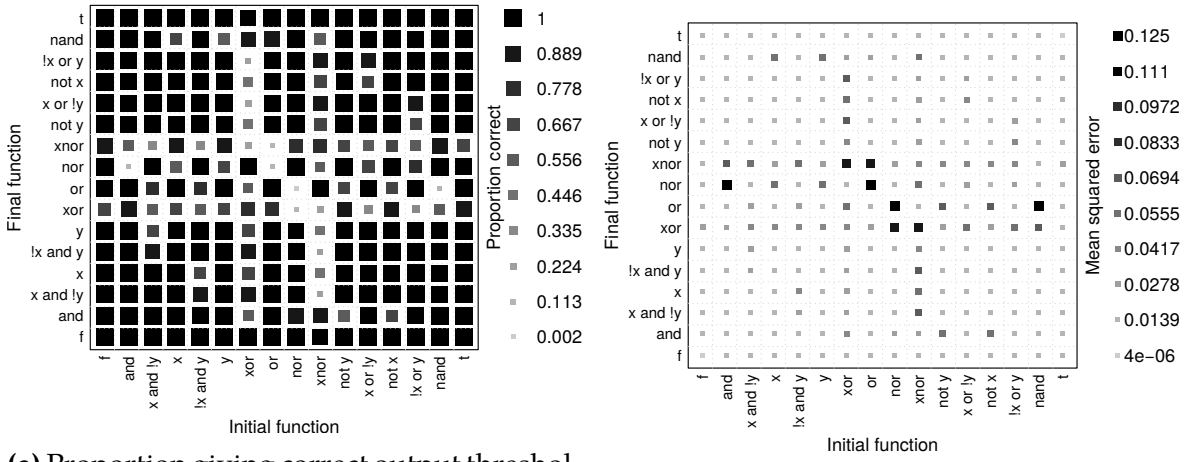
4.3 UESMANN on the boolean pairings

In order to determine the effectiveness of UESMANN on the boolean pairings discussed in previous sections, the algorithm described above was run on each of the 256 possible pairings. The hyperparameters were as follows:

- The learning rate η was set to 0.1, determined by informal experimentation.
- 300000 pair presentations were done (i.e. *MaxIterations* was 75000, with 4 examples presented at $h = 0$ and $h = 1$ in each iteration). This was determined by informal experimentation after a suitable learning rate had been found.
- No early stopping was done — networks which failed to converge to a "good" minimum were left to fail. This avoids the need for extra parameters determining when early stopping should occur.
- For each pairing, 1000 attempts were made from random initial starting points with weights and biases in the range $[-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$ (i.e. Bishop's rule). This was decided after informal experimentation, and was a compromise between the available computational resources and the need for a large sample.

The number of runs which successfully reproduced the pairing under thresholding at 0.5 was recorded for each pairing. No cross-validation could be done: with only 8

examples, which represented the two functions completely, no test set was possible. The results are shown in Fig. 4.3a.



(a) Proportion giving correct output thresholded

(b) Mean squared error

Figure 4.3: Grid of boolean pairings, showing what proportion of 1000 2-2-1 networks trained using UESMANN correctly performed the pairing under thresholding at 0.5 (75000 iterations for each run, 1000 runs, $\eta = 0.1$, initial parameters uniformly distributed in $[-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$), and the mean squared error of the outputs for both functions in those networks.

We can see that the pattern is similar to that of Fig. 3.10 (the Monte Carlo 2-2-1 experiment): pairings which occupy a smaller region of the solution space converge to a solution less often under UESMANN, as we would expect, but there are a few anomalies. Perhaps the most notable is the pairing $x \wedge y \to \neg(x \vee y)$ (AND→NOR) and its negation $\neg(x \wedge y) \to x \vee y$ (NAND→OR), which converges less often than its solution size would suggest. The plot in Fig. 4.4 shows the correlation between $\log_{10}$ of the Monte Carlo count and the proportion of networks which converge to a solution. The correlation is more of a general tendency, with Pearson's $r = 0.749$ and (more importantly given that the log curve may not be appropriate) Spearman's $\rho = 0.812$. The AND→NOR and NAND→OR outliers are marked with squares at the bottom right.

Fig 4.3b shows the mean squared error for all pairings: that is, the mean squared error of all possible boolean values passed to the two functions, over all the networks (whether they produced correct values when thresholded or not):

$$MSE_N = \frac{\sum_{n \in N} \sum_{x \in \{0,1\}, y \in \{0,1\}} \left((f(x,y) - n(x,y,0))^2 + (g(x,y) - n(x,y,1))^2 \right)}{2\,|N|}, \tag{4.17}$$

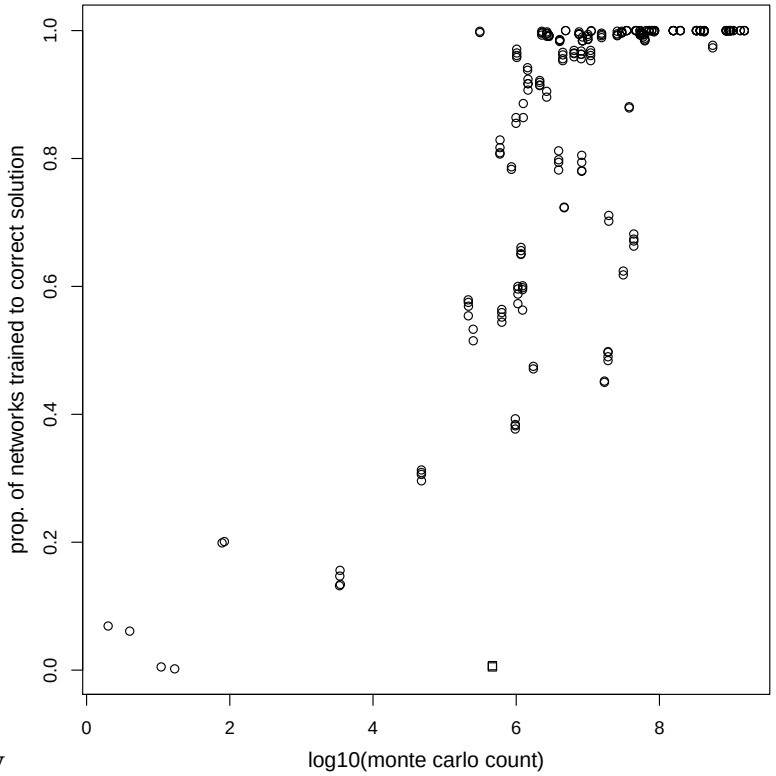


Figure 4.4: Proportion of successful convergences plotted against $\log_{10}$ of the Monte Carlo count for 2-2-1 UESMANN networks learning the boolean pairings at $\eta = 0.1$ with 75000 iterations. Pearson's $r = 0.749$ and Spearman's $\rho = 0.812$. The two very close points marked with a square (at bottom right) indicate the pairings $x \wedge y \longrightarrow \neg(x \vee y)$ and $\neg(x \wedge y) \longrightarrow x \vee y$, which are outliers discussed in the text.

where N is the set of 1000 networks, $n(x, y, h)$ is the function performed by network n, and f and g are the functions for which the network has been trained at $h = 0$ and $h = 1$. The pattern here shows an expected strong negative correlation between error and convergence (Pearson's $\rho = -0.959$): if there is a high enough error, the result will be incorrect. It also shows that the errors (i.e. $\sqrt{MSE_N}$) vary widely, from < 0.1 for most functions up to > 0.35 for the "difficult" functions. This is quite a high error, but it may seem odd that the error is not higher given that many of these networks are not producing the correct functions: consider AND→NOR, where the mean squared error is 0.1234, giving an error magnitude estimate of around 0.35, but with only 0.7% of the networks correct (from Table 4.1). However, in order for the network to be entirely correct, both functions must give the correct answer when thresholded for all possible boolean inputs. It is easy to see that a relatively low mean error could still cause a network to fail.

Table 4.1: Worst-performing boolean pairings, by proportion of networks correct

f	g	rank	both functions correct
$\neg(x \vee y)$	$x \vee y$	1	0.002000
$x \vee y$	$\neg(x \vee y)$	2	0.005000
$\neg(x \wedge y)$	$x \vee y$	3	0.005000
$x \wedge y$	$\neg(x \vee y)$	4	0.007000
$\neg(x \vee y)$	$x \oplus y$	5	0.061000
$x \vee y$	$\neg(x \oplus y)$	6	0.069000
$x \oplus y$	$\neg x \vee y$	7	0.132000
$\neg(x \oplus y)$	$\neg x \wedge y$	8	0.134000
$\neg(x \oplus y)$	$x \wedge \neg y$	9	0.147000
$x \oplus y$	$x \vee \neg y$	10	0.156000
$\neg(x \oplus y)$	$x \oplus y$	11	0.199000
$x \oplus y$	$\neg(x \oplus y)$	12	0.201000
$\neg x \vee y$	$x \oplus y$	13	0.296000
$x \vee \neg y$	$x \oplus y$	14	0.306000
$x \wedge \neg y$	$\neg(x \oplus y)$	15	0.309000
$\neg x \wedge y$	$\neg(x \oplus y)$	16	0.313000
$\neg(x \oplus y)$	y	17	0.377000
$x \oplus y$	$\neg x$	18	0.382000
$\neg(x \oplus y)$	x	19	0.384000
$x \oplus y$	$\neg y$	20	0.393000

4.3.1 Convergence in a single node

We might predict that Algorithm 1 would take a long time to converge compared with the standard back-propagation algorithm: it alternates between two different gradients across the two error surfaces, which may pull it in two different directions. It is fairly clear that "difficult" pairings — those which cover smaller regions of the solution space (Fig. 3.10), show lower convergence success figures, or higher errors (Fig. 4.3) — should take longer to converge and are more likely to do so to local minima (giving the failed convergences in Fig. 4.3a.)

Before looking at the convergence behaviour of back-propagation in networks with a hidden layer, we will investigate that of a single node performing a typical function marked as "possible" in Fig. 3.5. In addition to the convergence curves, we will also investigate the paths of the weights during training through the error volume at the level of an eventual converged bias, and the convergence behaviour at different learning rates.

The test function is $x \rightarrow x \vee y$, i.e. x to OR. This was chosen because it occupies a moderately small region of the solution space (see Table 3.1). This requires four

examples for $h = 0$ and $h = 1$ as shown in Table 4.2. 250000 iterations through this set were made at $\eta = 0.05$, using Algorithm 1. This low learning rate was chosen to expose any difficulties with local minima or flat regions, while the large number of iterations should ensure convergence. To provide a baseline for comparison, simple

Table 4.2: Training examples for single node training of $x \rightarrow x \vee y$

h	x	y	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

back-propagation (effectively the delta rule since we do not have hidden layers) was performed for the function x.

4.3.1.1 Method

Convergence paths: For the UESMANN node performing $x \rightarrow x \vee y$, Algorithm 1 was used with the examples given in Table 4.2. For the "control" learning the function x, the same algorithm was used but without hormonal modulation or training for a function at $h = 1$. We wish to evaluate the paths through the solution space from different starting points, plotting them on a 2D error surface. However, this surface is actually a 3D volume (since there are 3 parameters). As a compromise, we will show the convergence paths on a slice through the error volume at the mean bias at which converging runs end, first checking that these biases are normally distributed or, if not, of small variance, to ensure that the selected bias is representative.

The initial bias for all nodes was set at zero, as an arbitrary value: for the best convergence performance, the parameters should be close to zero. The initial weights are selected from a grid centred around $w_1 = 0, w_2 = 0$. Because some nodes far from the origin may not converge, the final bias at which the error surface is plotted is the mean of a smaller grid $w_1, w_2 \in [-2.5, -2, \ldots, 2.5]$. The initial weights for the nodes whose paths are shown are in $w_1, w_2 \in [-7.5, -4.5, \ldots, 7.5]$. The node was trained for 10^6 examples or pair presentations. This corresponds to 250000 iterations through the training examples (since there are four examples in the x set, and four example pairs in the $x \rightarrow x \vee y$ set). During training the node weights and bias are sampled every 100 pair presentations (every 25 iterations). As stated above, the

learning rate η is 0.05, which was chosen as a small value likely to expose local minima. Other learning rates were tested for individual nodes (see below).

Once generated, the paths from 36 initial weights to convergence were plotted on the mean error surface for the two functions at the mean final bias. The error surface was calculated by generating nodes w_1, w_2, b for w_1, w_2 on a grid of 2500 points over a range larger than the weight range for the initial node settings, to give some sense of the surrounding topography, and generating the mean squared error for all examples (as given by Eq. 4.17). Each node was run with the example set, and the mean squared error of the outputs calculated. Additionally, the weight gradient at each grid point (at the final bias) was calculated directly from Eqs. 4.13 and 4.15, finding the mean for all examples. When interpreting the plots, it should be borne in mind that the paths pass through many parts of the error volume — if a path does not appear to be following the gradient shown, that is likely because the gradients at the path's current bias and the final bias are different.

For some notable nodes, the individual parameters are plotted over training. In these plots, the parameters are approximated with a 100 point spline generated by the R `sm.spline` command from the `pspline` library.

Convergence plots: The mean squared errors over iteration for noteworthy nodes found in the convergence paths plot generated in the previous step were plotted individually. In these plots, the MSE is sampled every 10 iterations (to avoid large image files). However, the data was confirmed to be smooth before so doing by plotting without downsampling.

Learning rate tests: To understand the effect of different learning rates and investigate the topography further, some nodes were also converged at a range $\eta = \frac{4^n}{100}, n \in \{0, 1, 2, 3, 4, 5\}$, i.e. $\eta \in \{0.01, 0.04, 0.16, 0.64, 2.56, 10.24\}$. This was selected as a geometric progression covering a wide range of values. These were again performed for 10^6 iterations. In these plots, the MSE was sampled every 1000 iterations.

4.3.1.2 Results for x

Performing delta rule training for the function x, i.e. the truth table in Table 4.3, we obtain the paths in Fig. 4.5. We can see that all paths converge to solutions within a large flat region. The plot shows the error surface for x with a bias of -4.744, obtained as the mean from the initial small grid run within which $\sigma_{samp}(b) = 0.0009$. Although the distribution was not normal (Shapiro-Wilk $p < 0.05$), the variance is so small ($-4.7466 < b < -4.7243$) that the mean can be seen as representative of the solutions.

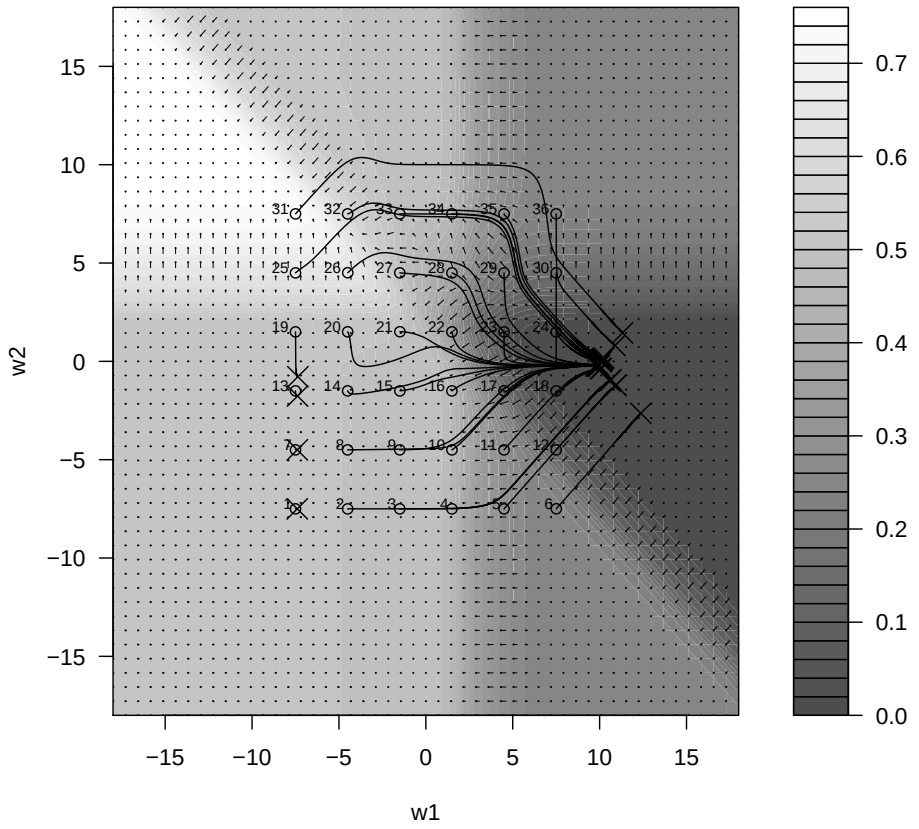
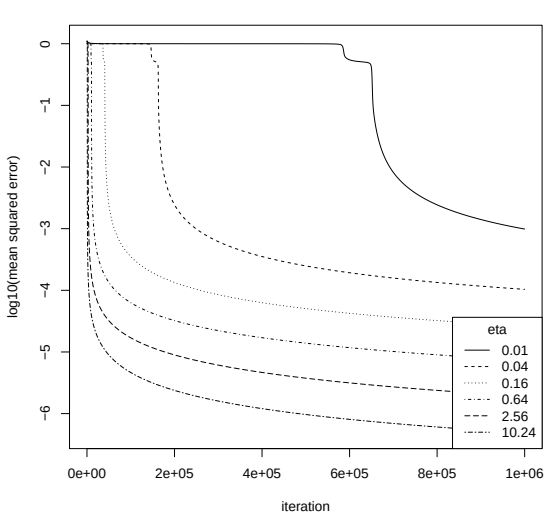


Figure 4.5: Paths of 36 nodes during training for boolean x, overlaid on a plot of the mean error surface at a mean final bias for converged nodes, with arrows showing the vector field for weights trained using back-propagation at the same bias. Weights are sampled every 25 iterations.

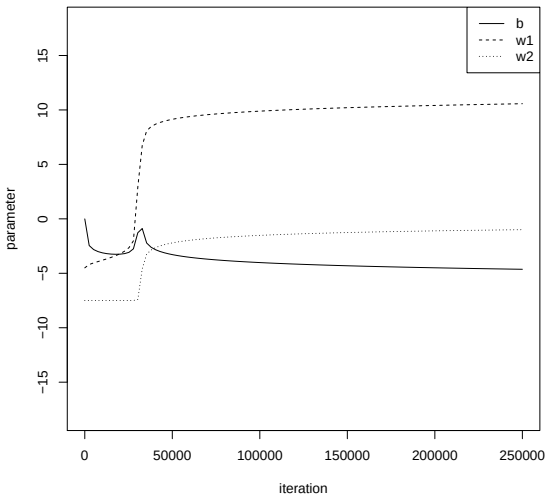
Table 4.3: Training examples for single node training of x

h	x	y	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1

Looking at the convergence of a typical individual node (number 2) in Fig. 4.6, convergence is rapid and smooth after an initial period in a large, flat region — this period would be reduced if the initial weights were closer to the origin. The parameter path shows the bias rapidly decreasing toward the final level with w_1 increasing slowly, until the flat region is escaped. Both weights now increase to their final level, with a small "bump" in bias as w_1 crosses the origin. The parameter and



(a) Convergence of node 2 for x at different η



(b) Paths through weight space at bias=-4.744, $\eta = 0.05$

Figure 4.6: Paths to convergence after 25000 iterations plotted on error surface for plain back-propagation for single node (i.e. delta function) learning the x boolean function. The bias is the mean of those at final iteration from the [-2.5,2.5] grid as described above, giving -4.744 with $\sigma = 0.0008$.

path plots show the node moving through discrete regions of the error volume with quite different gradients.

Fig. 4.6a shows the convergence of node 2 at different learning rates. Using smaller learning rates causes problems in flat regions, as we would expect: the learning takes longer to escape. Even tiny flat regions, such as that roughly between points 5 and 12 on the path plot, cause problems for node 2 where $\eta = 0.01$ — this is the cause of the new flat region near the end of the run.

Again, this is a single node being trained for a simple boolean function by the delta rule: even in this simple case the topology of the error surface is not trivial. We may see considerably more complexity in UESMANN.

4.3.1.3 Training a UESMANN node for $x \rightarrow x \vee y$

The mean bias at convergence was found to be -11.11, with the distribution shown in Fig. 4.7. This is at least an approximately normal distribution (Shapiro-Wilk $p > 0.1$), with narrow variation ($\sigma_{samp} = 0.0009$), so this mean was used as the basis for the error surface plot. The surfaces at this bias for the two functions are shown in Fig. 4.8 as is their mean — the notional surface UESMANN should follow.

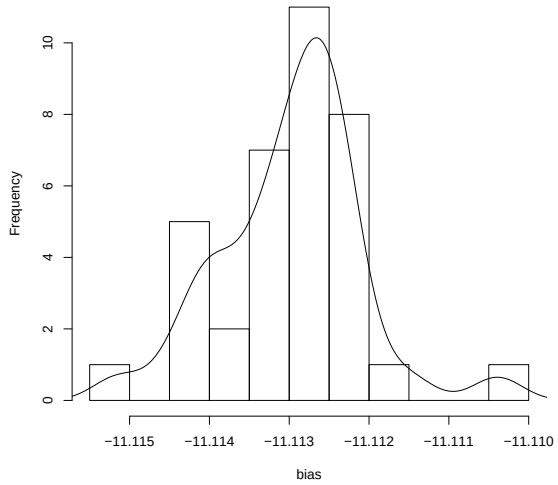
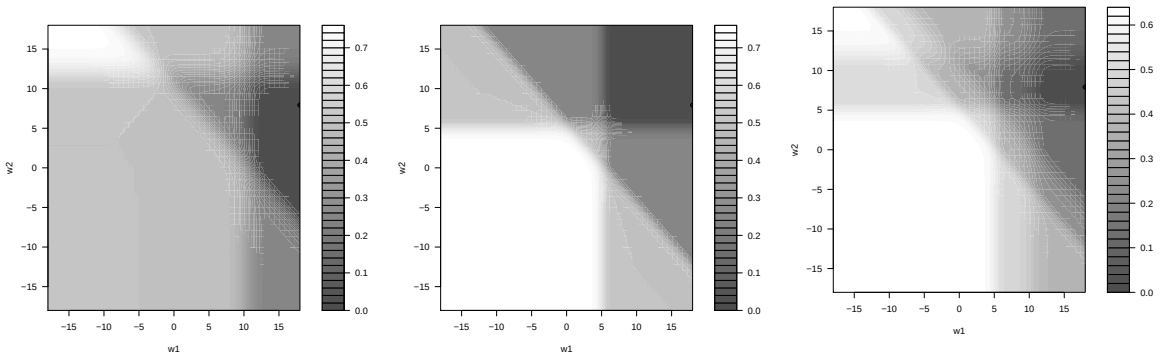


Figure 4.7: Histogram of bias distribution for single nodes trained using Algorithm 1 for $x \longrightarrow x \vee y$



(a) x **(b)** $x \vee y$ **(c)** mean error surface for x and $x \vee y$

Figure 4.8: Error surfaces over w_1, w_2 at $b = 0$ for the functions x and $x \vee y$, found by generating nodes with the given weights and biases and running them over the examples in Table 4.2, finding the mean squared error for all examples.

Once these surfaces were generated, a wider range of initial weights $w_1, w_2 \in [-7.5, 7.5]$ was used for generating nodes whose paths were to be plotted in order to include a few which did not converge to a solution (i.e. the converged node did not perform the required pairing). Convergence curves and parameter plots at $\eta = 0.05$ are shown in Fig. 4.10 for a typical successful node, and for one of the two nodes which did not arrive at a solution. Both nodes show a rapid initial decrease (over the first iteration), after which the unsuccessful node (run 1) is in a flat region. Inspection of the path shows that the weights are hardly changing, while the bias is

decreasing, approaching an asymptote: see Fig. 4.10c. Fig. 4.9 shows the paths of all

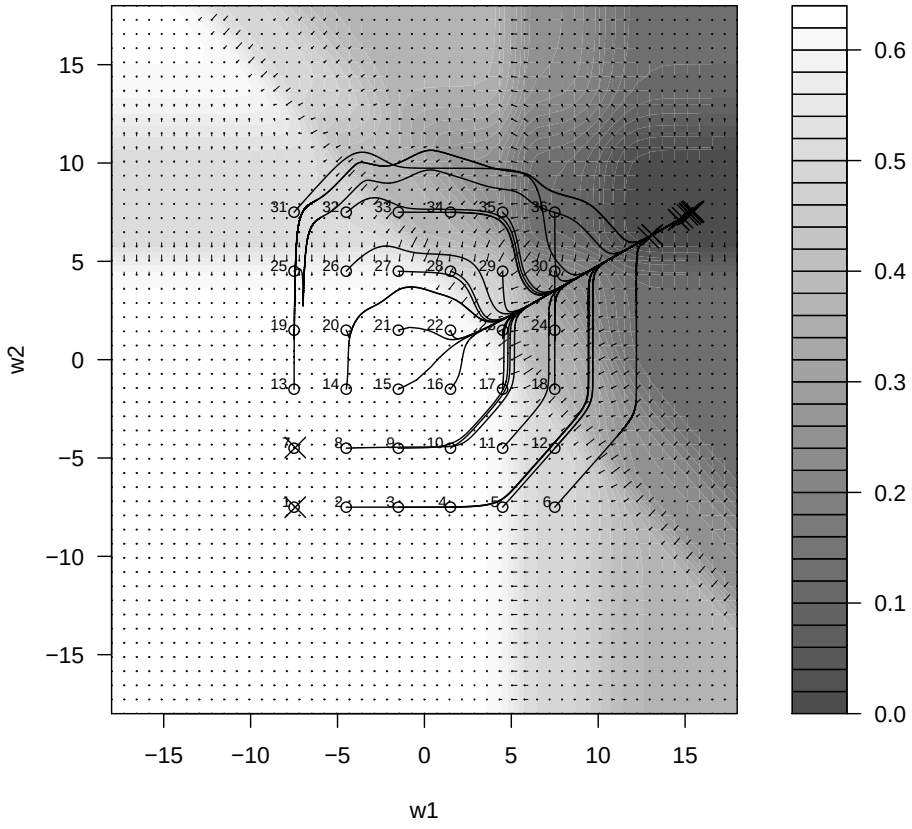
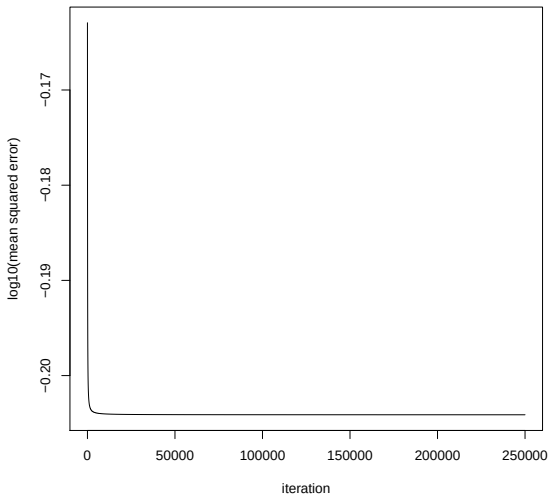


Figure 4.9: Paths of 36 nodes during training for $x \to x \vee y$, overlaid on a plot of the mean error surface for at a mean final bias for converged nodes, with arrows showing the vector field for weights trained using back-propagation at the same bias. Weights are sampled every 25 iterations.

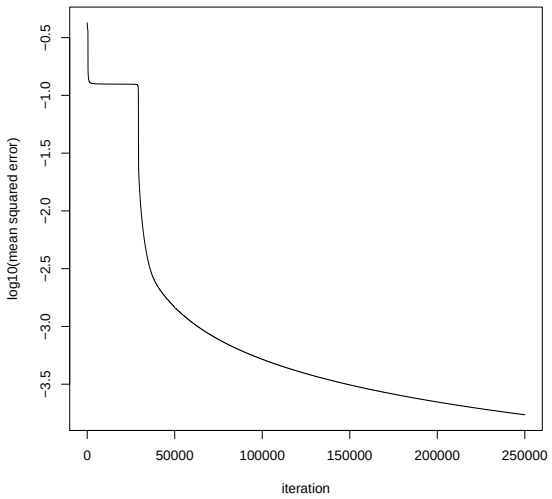
36 nodes overlaid onto the error surface for $b = -11.11$, the mean converged value found above. Also shown is the vector field generated by Eqs. 4.13 and 4.15, used to update the weights. Again, this is the vector field and error surface at the bias of final convergence, so this will not apply for early parts of the paths.

We can see that nodes 1 and 7 are unlikely to converge successfully to a good minimum: these start learning in a region far from the eventual solution, and flat at both the converged bias and at $b = 0$, the starting point. With the exception of these runs, all nodes converge to the same successful minimum. Note, however, the unusual paths taken by some nodes, such as 31 and 32, and particularly node 25, which appears to change direction in w_2 drastically near the beginning before moving rapidly to positive w_1 and converging on the solution as shown in Fig. 4.11. Clearly the error surface is more complex than a simple slice would suggest. It is

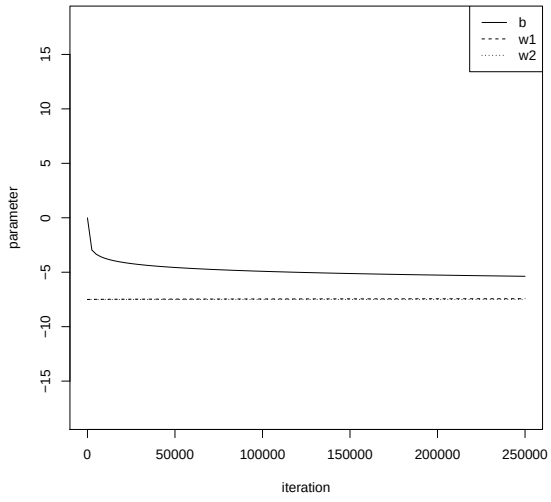
worth noting that the convergence, while complex, is smooth. There is no oscillation, at least at this low η, which also accounts for the long convergence times.



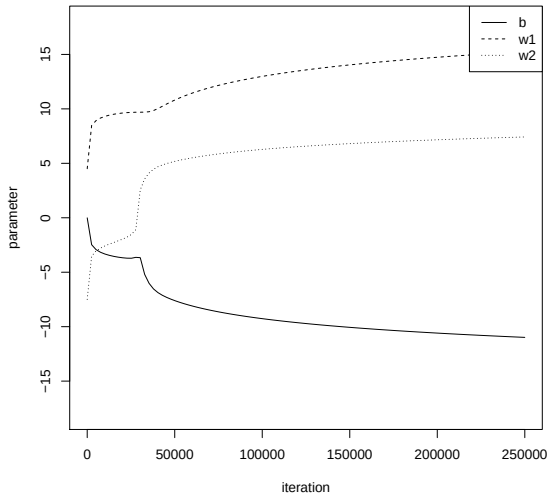
(a) Node 1 convergence, $w_1 = -7.5, w_2 = -7.5$



(b) Node 5 convergence, $w_1 = 4.5, w_2 = -7.5$



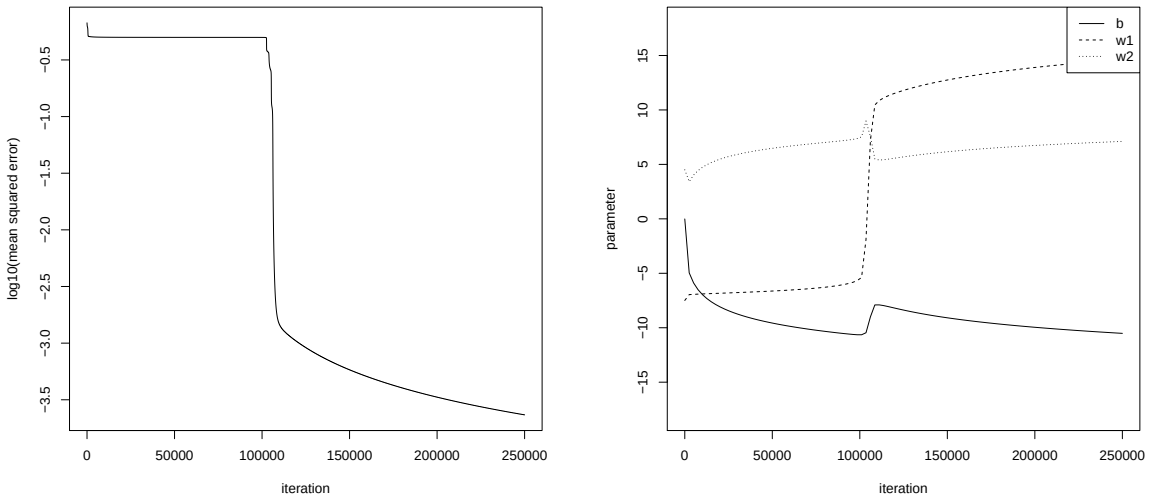
(c) Node 1 parameters, $w_1 = -7.5, w_2 = -7.5$



(d) Node 5 parameters, $w_1 = -7.5, w_2 = -7.5$

Figure 4.10: Convergence curves and parameter plots for nodes 1 and 5 during training for $x \to x \vee y$, one successful and one unsuccessful, sampled after every iteration. The convergence plots are on a log scale and are sampled every iteration, The parameter plots are smoothed using a spline.

The convergence of a more typical successful node (number 5) in Fig. 4.10b also shows a number of distinct phases. The parameter plot in Fig. 4.10d shows



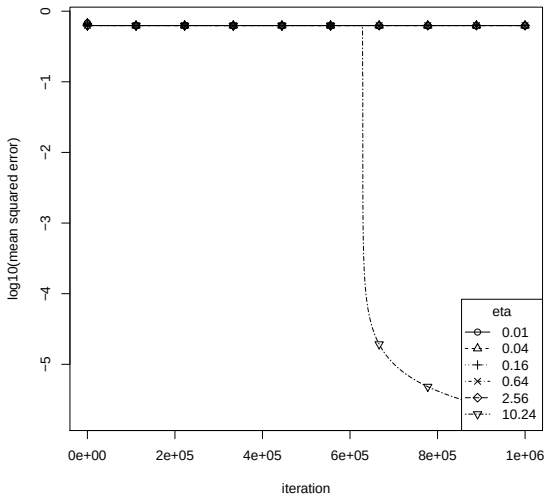
(a) Mean squared error

(b) Parameters

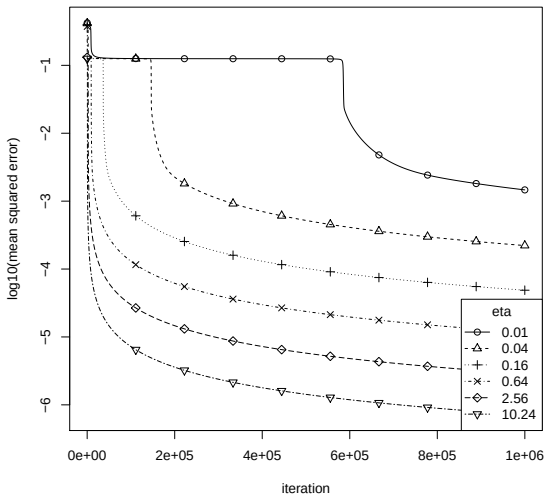
Figure 4.11: Convergence and parameter paths for node 25 (which starts at $w_1 = -7.5, w_2 = 4.5$) during training for $x \longrightarrow x \vee y$, which changes direction abruptly (see Fig. 4.9).

each of these phases corresponds to a change in the gradient being followed, as we might expect: note particularly the rapid drop in error just before iteration 30000, accompanied by a rapid change in the bias and an increase in the rate of change of w_2. It should also be noted that the error is still falling at the final iteration, although this effect is greatly enhanced by the use of a log scale. The minimum found still has a small gradient along which the solution will move, and this gradient probably approaches zero asymptotically as w_1 increases.

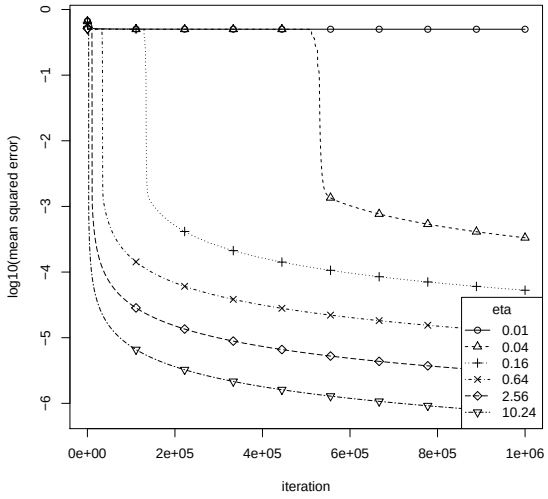
Convergence curves for the three different nodes discussed above at different learning rates are shown in Fig. 4.12. These show a similar behaviour to the plain delta rule, with higher η converging faster and escaping flat regions. Indeed, it seems that $\eta = 0.05$ was rather too conservative, with the very high $\eta = 10.24$ escaping its flat region in the normally non-convergent node 1. In this case, the node must escape a large flat region — increasing η gives the node time to do so.



(a) Node 1 (unsuccessful)



(b) Node 5 (successful)



(c) Node 25 (successful, but unusual)

Figure 4.12: $\log_{10}$ of mean squared error against iteration for nodes 1, 5 and 25 at different learning rates

4.3.1.4 An impossible pairing: $x \vee y \rightarrow x$

Fig. 4.13 shows a similar set of runs for an "impossible" pairing, $x \vee y \rightarrow x$, which does not appear in Fig. 3.5. Note that the error in the surface does not reach zero; no point in the plot is a solution. However, the paths converge to minima, with the majority finding the best (but still inadequate) solution in a long valley $w_1 > 5, w_2 \approx 0.28$.

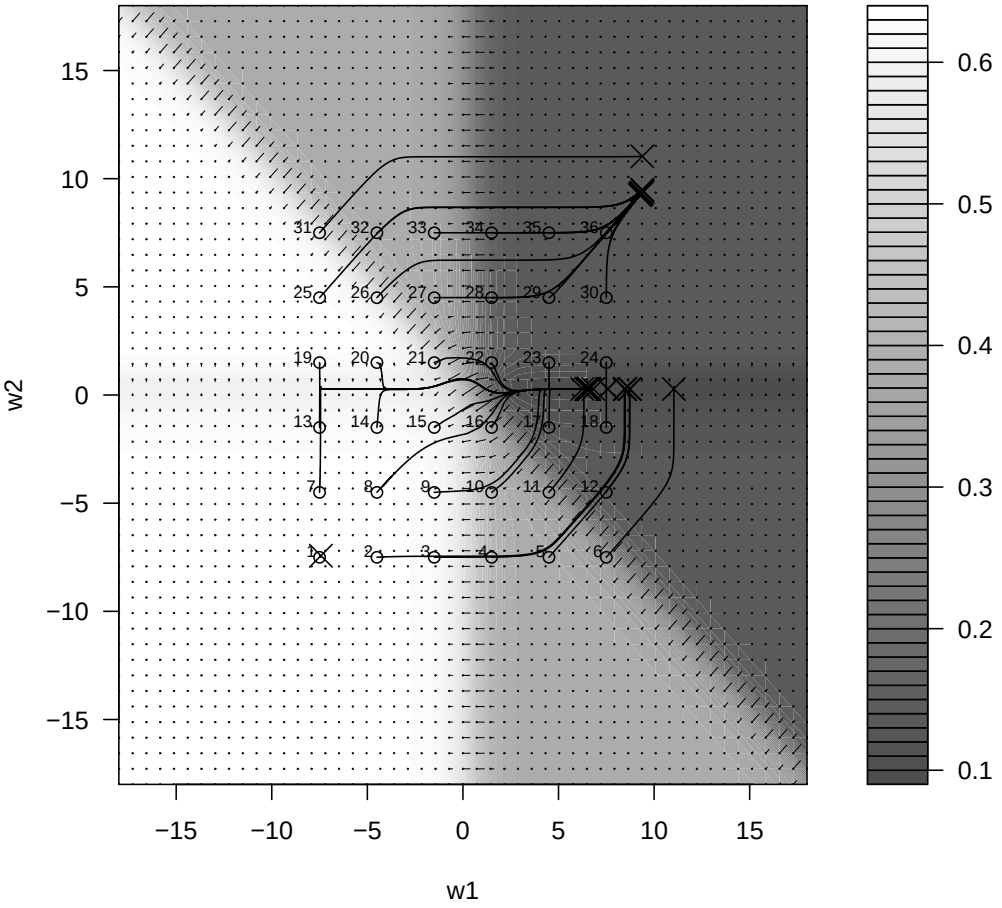


Figure 4.13: Paths of 36 nodes during training for $x \vee y \rightarrow x$, overlaid on a plot of the mean error surface at a mean final bias for converged nodes, with arrows showing the vector field for weights trained using back-propagation at the same bias. Weights are sampled every 25 iterations.

4.3.1.5 Summary

These brief explorations demonstrate that even in a single node, the convergence of UESMANN with back-propagation can be complex due to the nature of the combined error surface which is being traversed — however, the modified back-propagation algorithm (actually the delta rule because we are only looking at a single node) appears to follow the gradient well. High learning rates are advantageous here because of the number of flat regions in the error volume, although this may only apply when initial weights and biases are too large, and only in these boolean problems. We will now look at convergence in a network with hidden nodes.

4.3.2 Convergence in networks with hidden nodes

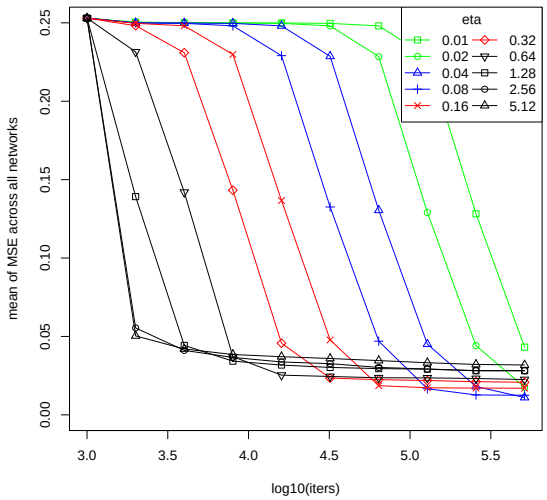
This section examines the convergence behaviour of 2-2-1 networks learning three different boolean pairings, none of which are possible in a single node:

- $x \vee y \rightarrow x \wedge y$ (OR to AND) is an "easy" pairing, occupying 4.8×10^{-5} of the solution space (Fig. 3.10) and converging to a correct solution in all 1000 trials (Fig. 4.3a).
- $x \oplus y \rightarrow x \wedge y$ (XOR to AND) is a "moderate" pairing, occupying 2.5×10^{-6} of the solution space and converging to a correct solution in 62% of 1000 trials.
- $x \wedge y \rightarrow \neg(x \vee y)$ (AND to NOR) is a "difficult" pairing, despite occupying 4.65×10^{-6} of the space, converging to a correct solution in 0.7% of 1000 trials. While not the hardest pairing to train (this honour goes to $x \vee y \rightarrow \neg(x \vee y)$ (OR to NOR) and its negative), it is anomalous in that the Monte Carlo plot in Fig.3.10 suggests it should be easier to train.
- $x \oplus y$ (XOR), trained using plain back-propagation, is used as a basis for comparison.

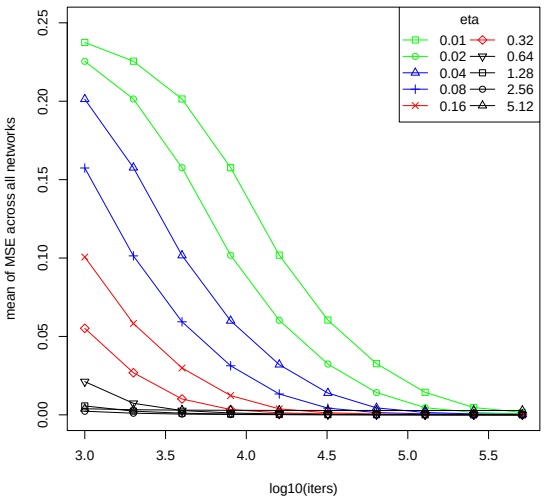
An initial set of runs was made to study the performance at different learning rates. 500 networks with initial random weights and biases in $[-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$ were trained, with the performance being calculated as the MSE across all examples (Eq. 4.17). The networks were tested at a range of different iteration counts, with the mean results in Fig. 4.14. Note that these plots are logarithmic on the iteration count axis. Also note, importantly, that these merely show the *mean performance of the networks.* Rather than showing statistical details, the learning behaviour of some individual networks for each of these problems is examined in the next section.

Firstly, the "control": training a neural network to perform XOR with back-propagation is, perhaps, the first non-separable problem tackled in neural networks, dating back to Rumelhart, Hinton and Williams [237]. Fig. 4.14a shows the networks converge to successful minima with increasing speed as η increases, but that there appears to be a large "flat" region near the starting point for most networks: the mean MSE takes some time to begin falling. The performance of the resulting network is slightly worse (from the point of view of mean squared error) at higher rates, because here the network is oscillating slightly around a solution. However, the solution region is large enough that there is still a good result.

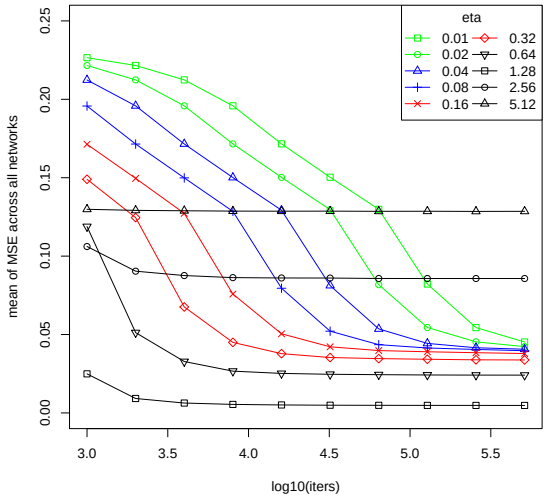
Fig. 4.14b shows the "easy" UESMANN problem of $x \vee y \rightarrow x \wedge y$ (OR to AND) converging to solutions rapidly at all η, typically more quickly than the control. It



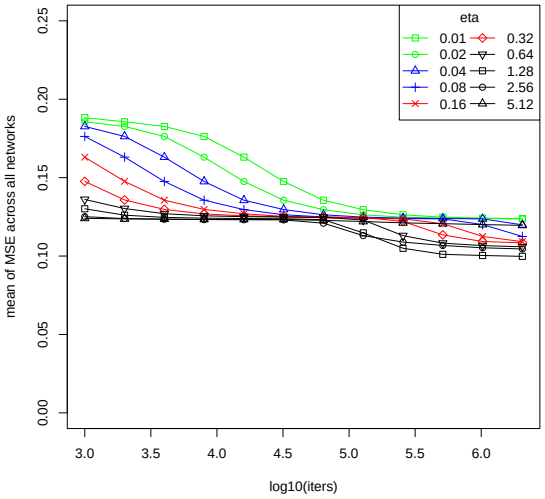
(a) $x \oplus y$ (plain backprop)



(b) $x \vee y \longrightarrow x \wedge y$



(c) $x \oplus y \longrightarrow x \wedge y$



(d) $x \wedge y \longrightarrow \neg(x \vee y)$
(note longer training)

Figure 4.14: Mean squared error against $\log_{10}$ of training pair-presentations (i.e. four times the number of iterations) for three different boolean pairings in 2-2-1 UESMANN networks at different learning rates η. Also shown is the MSE against presentations for plain back-propagation learning $x \oplus y$ (XOR) as a basis for comparison. **Note that the plot begins at 1000 pair-presentations: the initial error is not shown.**

may be that UESMANN avoids flat regions near the start simply because it is working with two gradients: where the surface is flat for one function, it is unlikely to be flat

in the other. We might also expect some speed increase if the gradients of the two functions align: Algorithm 1 performs two updates where normal back-propagation performs one, so if the two updates happen to be along roughly the same gradient, we effectively get "two updates for the price of one."

These systems have a simple error surface with large minima and few problematic local minima — using a high η will allow flat regions to be traversed rapidly, so a solution can be found quickly. The moderately difficult $x \oplus y \longrightarrow x \wedge y$ pairing, shown in Fig. 4.14c converges increasingly quickly and to better solutions as η increases, until $\eta = 1.28$. After this, performance rapidly declines (in the log scale).

The "very difficult" system $x \wedge y \longrightarrow \neg(x \vee y)$, shown in Fig. 4.14d converges slowly and rarely finds an adequate solution — the best mean MSE achieved is 0.1 at $\eta = 1.28$, giving an mean error magnitude of ~ 0.3. However, although the networks at this high η on average perform slightly better from the point of view of mean squared error across all networks, none of the individual networks actually perform the required functions when the output is thresholded. Lower η values have a higher mean MSE but still produce solutions at a rate of around 1 in 100, as can be seen in Fig 4.15 which shows the proportion of "good" networks. The error surface for this pairing appears to have a dominating local minimum for a poor solution, such that a relatively low η and a fortuitous choice of initial weights are required to find it. It is an anomaly, as has already been noted in Sec. 4.3: the proportion of good solutions found by back-propagation is far lower than the size of its solution space would suggest.

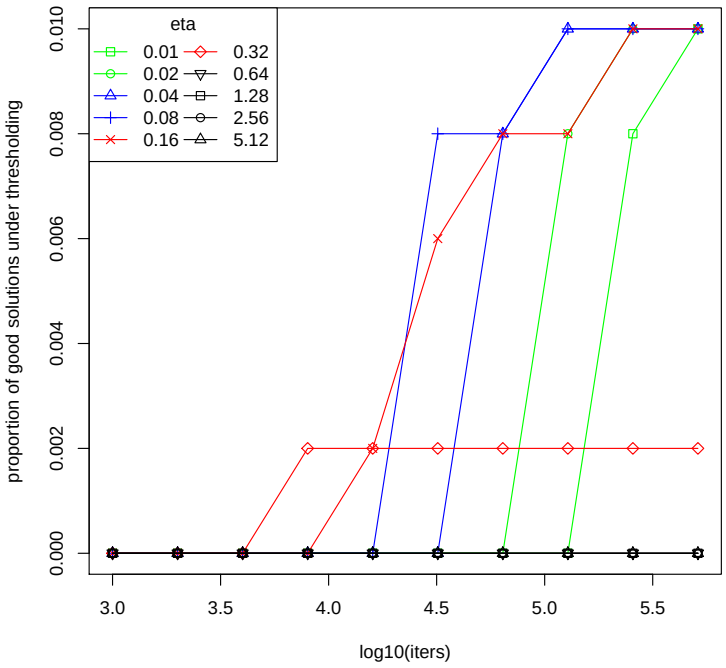


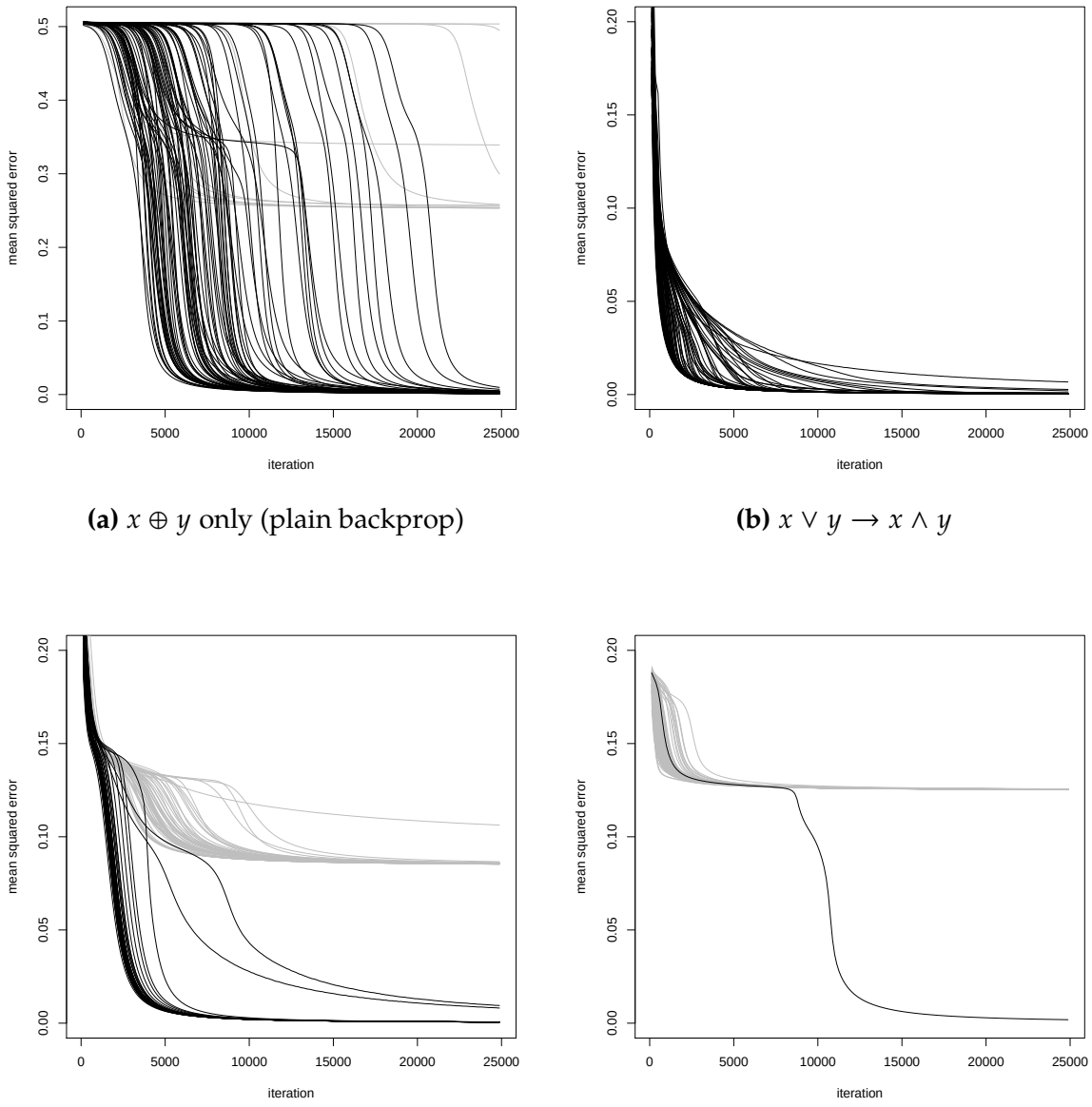
Figure 4.15: The proportion of networks trained for $x \wedge y \longrightarrow \neg(x \vee y)$ which perform those functions when outputs are thresholded at 0.5

4.3.3 Performance studies of individual networks

Informed by the results of the previous experiments, for the following experiments $\eta = 0.1$ was chosen as being sufficiently low to expose differences in the paths followed through convergence, while being high enough for most networks capable of converging to a solution to do so.

For each pairing, 100 attempts were made from different initial random weights, and each attempt was run for 25000 iterations (i.e. four paired examples run through 100000 pair updates: iterations of the inner loop of Algorithm 1). The mean squared error was sampled every 500 pair presentations (125 iterations), giving the results in Fig. 4.16.

Training for $x \vee y \longrightarrow x \wedge y$ shows ready convergence to a wide minimum, or possibly several close minima. $x \oplus y \longrightarrow x \wedge y$ shows a pattern of convergence to a number of minima, two of which are within the success criterion (i.e. produce correct outputs under thresholding for all inputs $x, y \in \{0, 1\}$). Most networks converge to the minimum with the lowest error, although a pair of outliers converge to slightly higher errors. The networks can diverge fairly late, as late as 10000 iterations (or 40000 pair presentations). With $x \wedge y \longrightarrow \neg(x \vee y)$, we see the vast majority of pairings



(a) $x \oplus y$ only (plain backprop)

(b) $x \vee y \to x \wedge y$

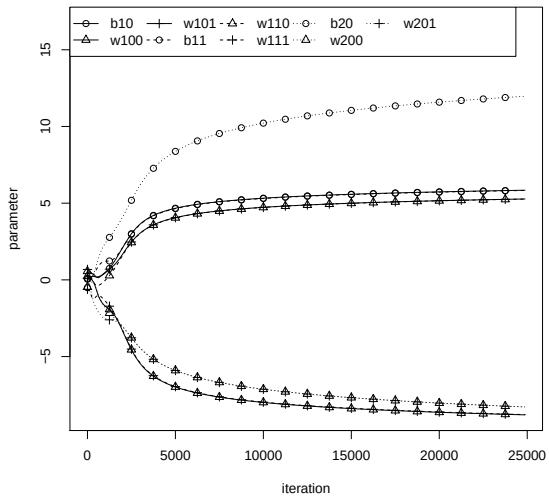
(c) $x \oplus y \to x \wedge y$

(d) $x \wedge y \to \neg(x \vee y)$

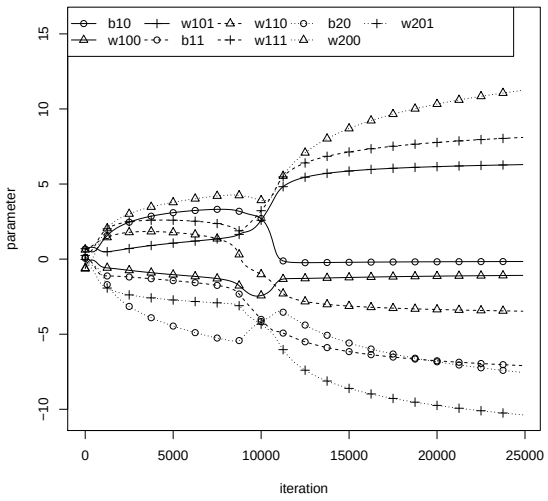
Figure 4.16: Mean squared error (see Eq. 4.17) for 100 attempts of three different pairings (and also XOR only), sampled during training every 500 pair presentations. Networks which converged to a successful solution (when outputs are thresholded at 0.5) are shown in black, unsuccessful networks are shown in grey. Learning rate $\eta = 0.1$.

heading into what appears to be a large, flat local minimum — the networks are a little spread out (reflected by a slight difference in their final values not apparent in the plot). However, one network finds an "escape route" on its way into the minimum.

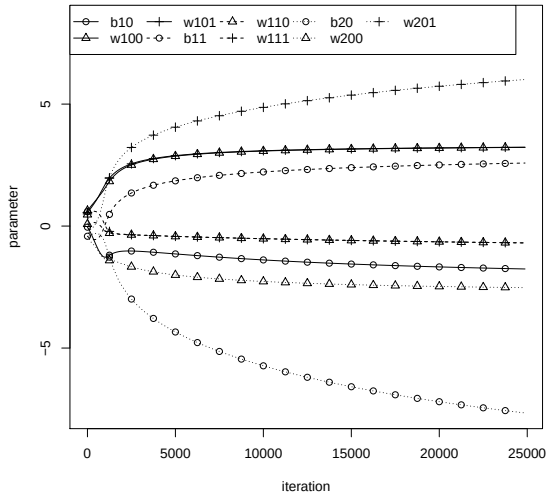
Fig. 4.17 shows the actual parameters of some networks of the two harder pairings during training. $x \oplus y \rightarrow x \wedge y$ shows a quick departure from the initial random values towards the solution, while $x \wedge y \rightarrow \neg(x \vee y)$ has a rather more complex path in the successful run. The failed run path is simple, and the contrast between this and the successful plot, along with the rarity of solutions, suggest that the the successful network converges on an fairly small "escape route" at about 10000 iterations. Since the back-propagation algorithm (in which the weights follow a flow along the an error surface) is a dynamical system, we can perhaps use dynamical systems terminology and describe this point in the training as a bifurcation point: the weights will take one of two paths, dependent on the initial conditions, with the successful path being rare. It is likely that the error surface for this problem has a complex topography, more so than the other problems, with a narrow area of the solution space containing gradients leading into the solution itself.



(a) $x \oplus y \longrightarrow x \wedge y$ (successful)



(b) $x \wedge y \longrightarrow \neg(x \vee y)$ (successful)



(c) $x \wedge y \longrightarrow \neg(x \vee y)$ (failed)

Figure 4.17: The weights and biases of some networks during training: one example of a successful, moderately hard pairing is shown, and examples of successful and failed networks for the most difficult pairing.

4.4 Summary

We have determined that 2-2-1 networks can be trained using Algorithm 1 to perform any pairing of boolean functions, with a success rate roughly correlating with the size of the solution volume as found by Monte Carlo simulation, with the exception of a notable pair of outliers: $x \wedge y \longrightarrow \neg(x \vee y)$ and $\neg(x \wedge y) \longrightarrow x \vee y$.

Studies were done to examine the convergence behaviour of some of these pairings, showing that most pairings perform well at a wide range of learning rates. The performance of some pairings is lower at low learning rates, with more networks becoming trapped in local minima or not converging given the number of training iterations. However performance in some networks also appears to degrade sharply as the learning rate increases: this may be because such pairings have a more complex error space, which requires the network to converge through a fairly narrow volume in the space before it can arrive at a solution.

The most difficult pairing converges extremely slowly, with a narrow range of learning rates being required for success. With too slow a learning rate the network either fails to converge or gets trapped in a local minimum which dominates the solution space, while with a high learning rate the network misses the small "pass" in the error space through which the solution lies.

Studies of individual networks during training bear out these conclusions: the error space for UESMANN networks training more complex boolean functions appears to be complex, with several incorrect local minima (in the sense that they do not produce the required results). In the extreme case of $x \wedge y \rightarrow \neg(x \vee y)$ and its pairwise negation, the space is dominated by a single minimum which is rarely escaped. However, solutions are possible provided enough attempts are made from different initial values.

Future work could examine the effectiveness of batch methods and momentum in escaping these minima, although it is unlikely they would help in this case: batching may actually increase the chance of being caught in a minimum, since the stochastic method applies "jitter" to the networks' paths through the solution space making fortuitous escapes more likely; while momentum may simply speed the dive into a poor minimum, particularly where that minimum dominates large regions.

However, the results show that it is possible to train a 2-2-1 network to perform any given pairing of binary boolean functions under thresholding at 0.5. This is remarkable, since this is the minimum strict feed-forward network topology required for any single boolean, and we are using no extra parameters. We are simply doubling the weights of each node under modulation.

We will now move on to examining the actual networks generated, investigating how many solutions there are and how they cluster, and studying their behaviour under transition.

4.5 The nature of 2-2-1 boolean UESMANN networks

In this section we will examine some solutions discovered by UESMANN to determine how they function. This will be done by selecting a number of pairings based on how difficult they are to train, and training a number of both UESMANN networks (using Algorithm 1) and three types of network of similar complexity, described below in Sec. 4.5.1. This will provide information about how well UESMANN converges to solutions compared with other network types, and about the nature of the error surface.

This done, we will concentrate briefly on the UESMANN networks. The results of the previous section (see Fig. 4.16) show that UESMANN pairings tend to generate solutions which are close to one of several final mean squared errors, which would indicate that they are approximately the same solution. This suggests that the weights and biases of the networks themselves fall into tight clusters.

We will use a clustering technique to establish how many clusters there are, and find their centroids. If the clusters are tight enough, this will give us a reasonable approximation of the successful minima of the network, which we can then study.

For each minimum (each solution cluster's centroid), we will consider the action of the modulator as it moves from 0 to 1 and how it changes the behaviour of the nodes in the network, and thus the output. This will improve our understanding of how UESMANN nodes function in ways which can be generalised to more complex problems.

In the light of this analysis, we will also look at the transition behaviour of a number of UESMANN networks trained on the problem, and compare it with networks trained using the other techniques discussed below. We would expect that output blending will always produce a crisp transition at 0.5, because (a) we are linearly interpolating between two network outputs and (b) the individual functions are easy to train so the endpoints will always be close to 0 or 1. Weight blending is likely to produce complex transitions which will be different for every network, because there are a large number of possible solutions to each function and we are interpolating the weights, producing unpredictable networks due to the different "competing conventions" of the two parents. The h-as-input method should produce complex non-linear transitions, involving some intermediary boolean functions under thresholding. They may or may not be consistent (i.e. the same for different initial weights) depending on how many solutions there are to the problem.

UESMANN itself, we predict, will produce non-linear transitions which will be wide, i.e. the network will transition to an intermediary boolean as it moves from 0

to 1. It should be consistent in behaviour, because the solution will fall into a number of clusters, most of which are linked by some simple symmetry rules and so perform the same behaviour.

4.5.1 Alternative modulatory methods

Three other modulatory methods were selected as a basis for comparison. These were chosen primarily on the basis of being obvious and naïve ways of producing neural network architectures trainable by back-propagation which can interpolate between two functions — other reasons are given under the relevant method. The methods are:

- **Output blending**: Two topologically identical networks are trained separately to perform the $h = 0$ and $h = 1$ functions. At run time both networks are run separately and the output nodes are linearly interpolated with h as the parameter. We define the two networks' nodes by

$$a_i^l = \sigma\left(b_i^l + \sum_j w_{ij}^l a_j^{l-1}\right) \tag{4.18}$$

and

$$z_i^l = \sigma\left(c_i^l + \sum_j v_{ij}^l z_j^{l-1}\right), \tag{4.19}$$

using the conventions of Fig. 4.1 (p. 93) with a, w, b representing the activations, weights and biases of the first network and z, v, c representing those of the second network. The function $\sigma(x)$ is the activation function: the standard logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$. Thus the output layer is given by

$$o_i = (1-h)a_i^L + hz_i^L. \tag{4.20}$$

This method has no direct biological inspiration, and was was chosen because it is perhaps the most obvious method. Arguably, it corresponds to a simple action selection system: the modulator selects which network's output should be used, but in a continuous manner.

- **Weight blending**: As with output blending, two networks are trained separately to perform the two functions. At run time, the weights and biases are linearly interpolated between using h as the parameter to provide the final network to which the inputs are passed. Thus again we have two networks

with the same topologies, as given in Eqs. 4.18 and 4.19. We can denote our final network thus:

$$p_i^l = \sigma\left(r_i^l + \sum_j q_{ij}^l p_j^{l-1}\right), \tag{4.21}$$

obtaining the weights and biases represented by q and r with

$$r_i^l = (1-h)b_i^l + hc_i^l \tag{4.22}$$
$$q_{ij}^l = (1-h)w_{ij}^l + hv_{ij}^l. \tag{4.23}$$

The output of the network is then given by Eq. 4.21 for the output layer. It is also noteworthy that if we make the following definitions:

$$s_i^l = c_i^l - b_i^l \tag{4.24}$$
$$t_{ij}^l = v_{ij}^l - w_{ij}^l, \tag{4.25}$$

then

$$p_i^l = \sigma\left(b_i^l + hs_i^l + \sum_j (ht_{ij}^l + w_{ij}^l)p_j^{l-1}\right). \tag{4.26}$$

This is the similar to the alternative additive form of the Neal/Timmis system given in footnote 19 on p. 47; however, here the bias is also subject to modulation. This method was chosen because of this correspondence to the Neal/Timmis system, in the belief that exploring it would reveal interesting properties of that system and how it relates to UESMANN.

- h**-as-input**: The modulator is provided as an additional input to a standard multi-layer perceptron of the form described by Eq. 2.9, and the training examples for the two functions are both passed to the network with the extra input set to 0 or 1. This is analogous to a biological network with an input from a receptor neuron for a certain chemical, and therefore is not strictly "neuromodulation." It was chosen because it is architecturally identical to an unmodified multilayer perceptron, with only the semantics of the inputs changed.

4.5.2 Expected cluster symmetry

We have said that UESMANN networks are expected to fall into clusters, and given the topology and function of a UESMANN network, we can describe some symmetry properties of these clusters. The following discussions will use the weight and bias labelling given in Fig. 4.18, which itself is based on the convention in Fig. 4.1, page 93.

There are certain symmetries we can expect to find in 2-2-1 networks trained for

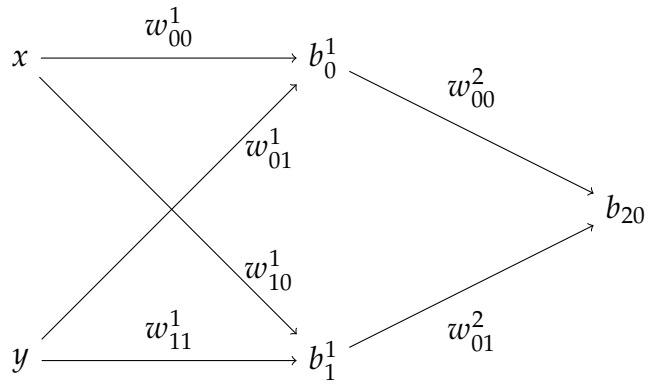
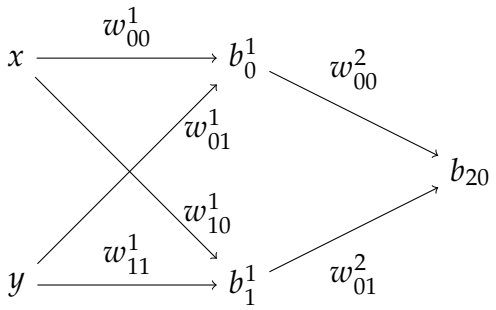
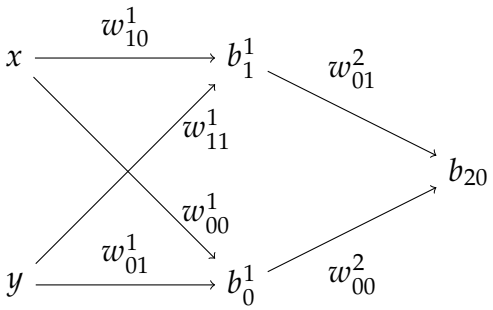


Figure 4.18: Labelling of weights and biases in networks – biases are shown in the nodes themselves. Bias labels are layer/node, weight labels are to-layer/to-neuron/from-neuron.

boolean pairings, since certain transformations of networks will function identically. Consider the transformation in Fig. 4.19. With the hidden nodes and their weights into the output node swapped, the networks N and $h(N)$ will function identically. This will hold for all pairings. This can be written as



(a) N : unswapped



(b) $h(N)$: hidden nodes swapped

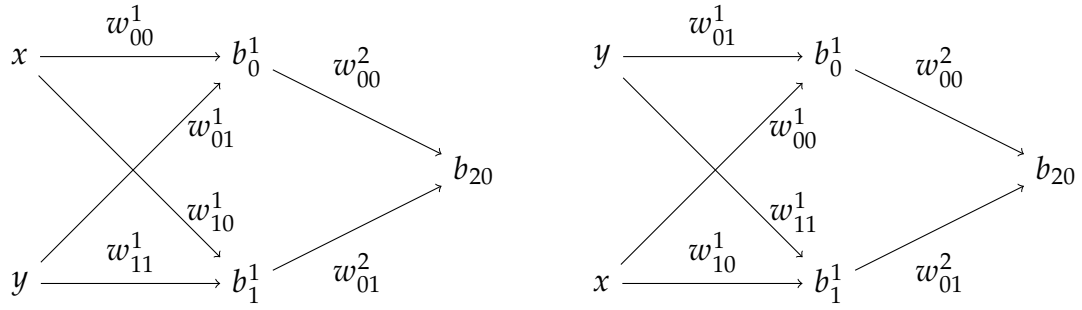
Figure 4.19: Hidden node swap transformation of a network N

$$h(N) = N : \{w^1_{00} \leftrightarrow w^1_{10}, \quad w^1_{01} \leftrightarrow w^1_{11}, \quad w^2_{00} \leftrightarrow w^2_{01}, \quad b^1_0 \leftrightarrow b^1_1\}, \tag{4.27}$$

where $N : \{\cdots\}$ represents N after a series of exchanges on its weights and biases, each described by $a \leftrightarrow b$, where the parameters to be exchanged are given using the notation in Fig. 4.18.

Now consider the situation in Fig. 4.20, where the inputs are swapped along with their weights into the hidden layer. This exchange will result in the same function being performed, but only if the function passed in is commutative. In a UESMANN

setting, both functions in the pairing must be commutative for this transformation to be valid.



(a) N : unswapped **(b)** $i(N)$: inputs swapped

Figure 4.20: Input swap transformation of a network N

This transformation can be written as

$$i(N) = N : \{w^1_{00} \leftrightarrow w^1_{01}, \quad w^1_{10} \leftrightarrow w^1_{11}\}, \tag{4.28}$$

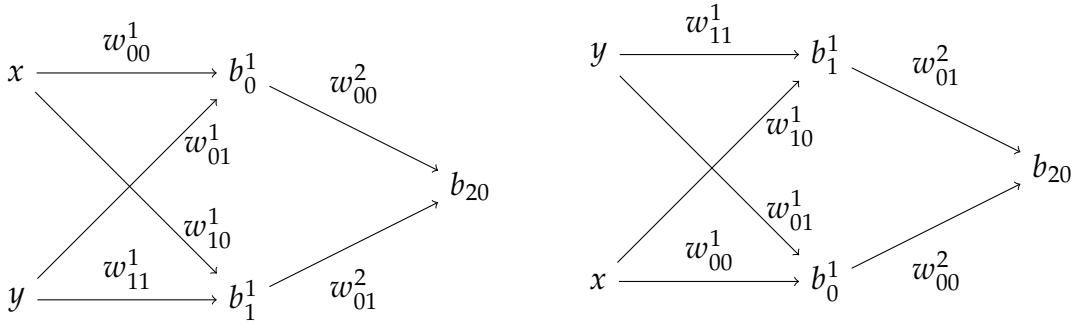
following the convention set earlier.

These transformations can themselves be composed into the transformations $h(i(N))$ and $i(h(N))$, which are identical, both consisting of the exchanges

$$i(h(N)) = h(i(N)) = N : \{w^1_{00} \leftrightarrow w^1_{11}, \quad w^1_{10} \leftrightarrow w^1_{01}, \quad w^2_{00} \leftrightarrow w^2_{01}, \quad b^1_0 \leftrightarrow b^1_1\}. \tag{4.29}$$

This transformation is shown in Fig. 4.21, and again is only valid if the functions the network is trained for are commutative. Although the transformation results in the network being apparently drawn "upside-down", the inputs and weights are exchanged: input x in the old network is now input y in the new, and weight w^1_{00} takes the place of weight w^1_{11} (and so on).

Thus for any network N which performs a pairing $f \rightarrow g$, there will also be a network $h(N)$ which also performs that pairing. If both functions are commutative, there will be two more networks $i(N)$ and $h(i(N))$ which perform the pairing. These solutions will occupy the same amount of solution space, but may not appear with equal frequency in the trained networks due to the complexity of the error surface. We will investigate this possibility for each network.



(a) N : unswapped

(b) $i(h(N)) = h(i(N))$: inputs and hidden nodes swapped

Figure 4.21: Input and hidden node swap transformation of a network N

4.5.3 Clustering method

We wish to find clusters of similar networks, i.e. networks which have similar weights and biases. The method chosen for finding clusters is agglomerative hierarchical clustering[78]:

- Each item is placed into a cluster by itself.
- The distances between all pairs of clusters are found. There are several methods for this. We have chosen the "average linkage distance": the mean Euclidean distance between all pairs of elements in each cluster. Two other methods commonly used are *complete linkage distance*, where the distance is the maximum distance between two points (one from each cluster), and *single linkage distance* which is the minimum such distance (all using the Euclidean distance between individual items). Complete linkage clustering is sensitive to outliers, and single linkage clustering can tend to form long chains which do not correspond to our intuitive idea of clusters as compact, spherical objects. The average linkage distance is a compromise between these two[174, 251].
- The two closest clusters are merged into a single cluster.
- The previous two steps are repeated until there are two clusters; however, all intermediate cluster merges are kept and examined by the experimenter.

Typically, the entire clustering process is visualised as a dendrogram — a tree diagram showing the process of agglomeration — where the height of the tree corresponds to the distance between the clusters. The decision of where to "cut" the tree, of which clustering to accept, is ultimately subjective; but typically the clustering which has the largest vertical space in the dendrogram is used. This is the clustering

for which the greatest change in distance is required before another agglomeration occurs. See Fig. 4.22 for an example.

4.5.4 Clustering solutions for $x \oplus y$ using plain back-propagation

To both provide a baseline for comparison and demonstrate the clustering technique, networks were trained for 75000 iterations with $\eta = 2$ to perform $x \oplus y$ using back-propagation with no modifications, as in Fig. 4.16a. New networks were generated until 500 successful instances were found, requiring 649 networks and giving a success rate of 77%. The high η was used here to match the result in the subsequent experiment, since we will compare this plain back-propagation $x \oplus y$ clustering with the clusters for $x \oplus y \longrightarrow x \wedge y$, to determine if any of the latter clusters (which perform XOR at $h = 0$) match clusters for networks trained to perform XOR alone. Additionally, we have already seen in Fig. 4.16a that this function is better trained at high η, possibly because of a large flat region in the error surface around the origin.

The results of the clustering operation are shown in Fig. 4.22. The dendrogram discussed above is shown on the left: at the bottom are the individual clusters at the start of clustering, each one containing a single network run. These are gradually agglomerated into clusters as the tree height increases, until we arrive at a single split. We have chosen a cut at height 20, where there are 10 clusters. We could have perhaps chosen 4 clusters, to get a coarser view with many outliers folded into clusters. The right hand figure shows the network parameters themselves, with each network shown as a polygonal chain coloured according to the cluster.

This is a complex clustering, and indicates that there are many solutions to the XOR problem, although two solutions (clusters 2 and 6) dominate with more than 200 individual networks each. Other solutions are outliers. The two dominant clusters are effectively the same clusters with the hidden nodes exchanged: recall Eq. 4.27 in Sec. 4.5.2. Similar relationships exist between the outliers. All the relationships and cluster sizes are shown in Fig. 4.23.

There is a large body of literature on the solutions for this problem [173, 29, 261, 111][5], so we will not analyse the clusters any further, having found tight clusters with which to compare UESMANN networks and demonstrated the clustering technique. We should note, however, that there may be many other solutions which it possible to generate using back-propagation, but they may be vanishingly rare.

[5]Note the that [173, 29] and [111] disagree on the existence of local minima in the XOR problem: this demonstrates the difficulty of analysing error surfaces in multilayer perceptrons!

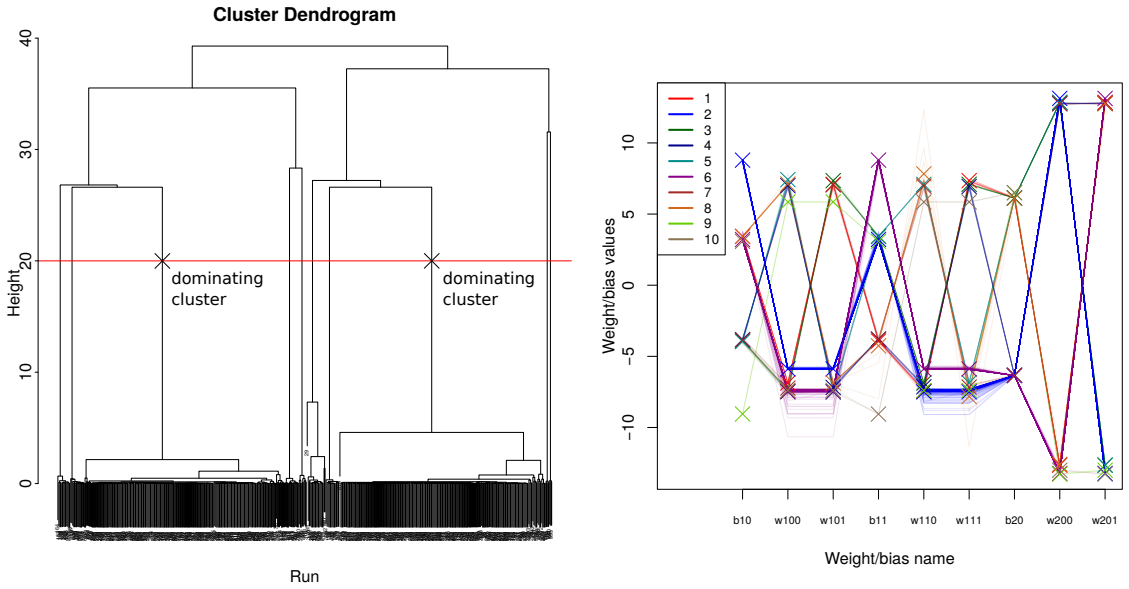


Figure 4.22: Hierarchical clustering of 500 networks successfully trained to perform $x \oplus y$, 75000 iterations at $\eta = 2$, Bishop's rule for initial weights, with accompanying dendrogram. The selected cut point is shown in the dendrogram as a red horizontal line, giving 10 clusters: the two dominating clusters are marked with a cross where they intersect the cut line. Within the cluster diagram itself, each individual network is shown as a faint polygonal chain of the cluster colour. There is more explanation of the dendrogram in the text.

4.5.5 Pairings tested

The two pairings tested were

- $x \oplus y \longrightarrow x \wedge y$ (XOR to AND)
- $x \wedge y \longrightarrow \neg(x \vee y)$ (AND to NOR)

The first, $x \oplus y \longrightarrow x \wedge y$, was selected as a "moderately difficult" pairing. It has a fairly small solution volume (20th smallest according to the Monte Carlo simulations performed earlier), and under the training regime specified for Fig. 4.3 converges to a solution for about half the networks (0.515) making it the 29th most difficult pairing — although this may vary for different learning rates and iteration counts.

Importantly, it is also represented in Fig. 3.9: it is implementable in a 2-2-1 UESMANN network in which each node performs a boolean function, although the solutions are extremely rare at 10^{-4} of the solution space for such networks. Because boolean nodes — nodes which perform boolean functions when thresholded at 0.5 — still perform their functions when saturated, there should be a larger number of solutions which use them. It is likely that the error surface will lead the algorithm to such a solution.

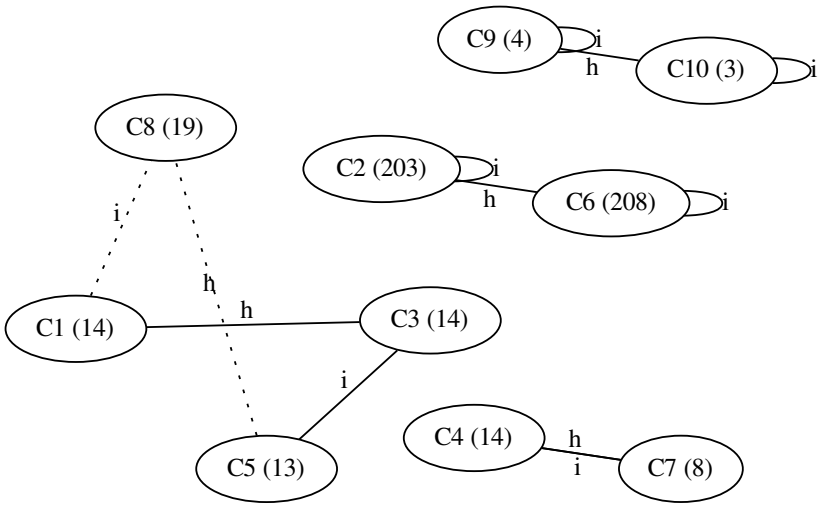


Figure 4.23: Relationships between the clusters in 500 solutions found for $x \oplus y$ at $\eta = 2$ with 75000 iterations, in terms of the symmetries discussed in Sec. 4.5.2. The number of nodes in each cluster is shown in brackets, with the relationship type h (hidden node swap) or i (input node swap) shown as a label on the relationship edge. If the relationship is weak (mean distance greater than 0.1) the edge is dotted.

The second network, $x \wedge y \rightarrow \neg(x \vee y)$, is an anomaly: Fig. 4.4 shows that it is extremely difficult to train with UESMANN despite having a reasonably large solution space: it is only the 24th rarest pairing in the Monte Carlo experiments. Sec. 4.3.2 (p. 113), shows that this pairing seems to converge to a good solution only through a very narrow path in the error surface, with networks appearing to fall into local minima which dominate the space. It is not represented in Fig. 3.9, and so cannot be implemented in a 2-2-1 network consisting of purely boolean UESMANN nodes. This may be a partial clue to its difficulty in training: it must require a hidden node which is operating in an unsaturated manner, which will require a narrower range of weights.

4.5.6 Analysis of $x \oplus y \rightarrow x \wedge y$ (XOR to AND)

4.5.6.1 Comparison of convergence of different techniques for $x \oplus y \rightarrow x \wedge y$

Fig. 4.24 shows the performance at different learning rates of UESMANN and the alternative modulation methods described in Sec. 4.5.1 (p. 122) for $x \oplus y \rightarrow x \wedge y$,

given 125000 iterations. The learning rates chosen are in the range

$$\eta = 0.01 \times 2^{0.5x}, x \in \{0 \cdots 19\}, \tag{4.30}$$

chosen to produce a reasonable range of values (given the previous experimental results) over a logarithmic scale. The metric is the proportion of networks which converged to a correct solution (i.e. one which performs the desired function pairing under thresholding) out of 100 trials.

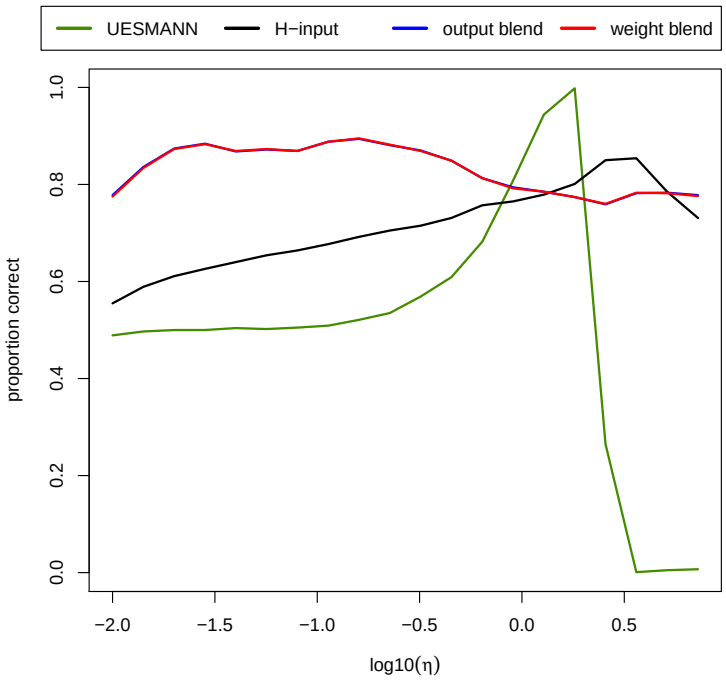


Figure 4.24: Proportion of networks trained for $x \oplus y \rightarrow x \wedge y$ which find the correct solution at different learning rates η after 125000 iterations, using four different types of modulatory network. Bishop's method was used for the initial weights. Note that weight and output blending are the same.

We can see that for this pairing UESMANN finds correct solutions for all but one trial if $\eta = 1.81$. This is a better rate of success than all other methods (note that a more accurate value for the optimum is shown in Fig. 4.28: $\eta = 2, \log_{10} \eta = 0.30$). The plots for output blending and weight blending are identical because in both cases we are generating networks for each function in the pairing and interpolating either their outputs or their weights: given that we are only evaluating the networks at the modulation extrema, weight and output blending are effectively the same operation. The h-as-input method converges to solutions well, but not as well as UESMANN at a well-chosen η. It also requires more parameters: a UESMANN network has

9, while an h-as-input network's extra input increases the count to 11 (two extra weights from the h input to the hidden layer).

On this pairing at least, UESMANN converges to solutions well — better than the other network types at a well-chosen η. For the blended network types this is likely to be because of a large flat area around the origin, which we have already established is likely to exist in the plain back-propagation error surface for $x \oplus y$ (see Fig. 4.16a). This will certainly inhibit the training of the $x \oplus y$ network required for blended networks.

To examine the convergence behaviour more closely, 100 networks were trained at the optimal learning rates found in Fig. 4.24. During training, the mean squared error in the output over the four possible boolean inputs $\{x, y\}; x, y \in \{0, 1\}$ was plotted for all networks separately for each h value $\{0, 1\}$. This was done to see if the networks moved towards solutions which "preferred" one of the functions in the pairing. The results for $x \oplus y \rightarrow x \wedge y$ at $\eta = 1.81$ are shown in Fig. 4.25.

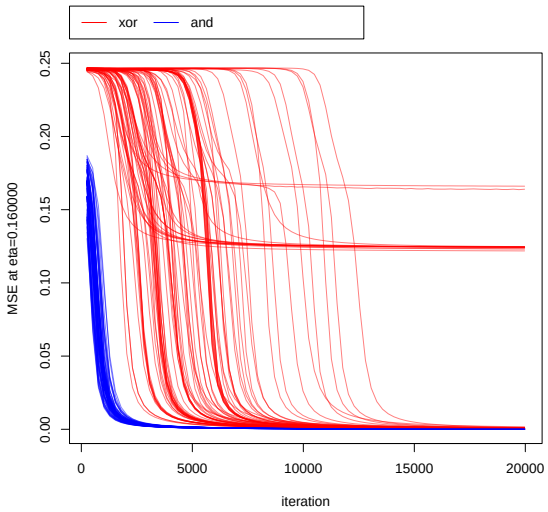
The results show again that output blending and weight blending have the same convergence behaviour when viewed at the modulator extrema. They also show that UESMANN networks converge readily to a solution within a few thousand iterations. Most h-as-input networks converge rapidly on a solution for $x \wedge y$ (although some do not), but have considerably more difficulty finding a solution within the former solution which also performs $x \oplus y$. We have already seen that this function is considerably more difficult to learn, at least by itself by plain back-propagation, and this will carry over into output blending and weight blending networks. In h-as-input the error surface seems more complex: the convergence oscillates until it finds a narrow pathway down the gradient; an oscillation exacerbated by the high learning rate (which is nevertheless the best learning rate of those tested).

4.5.6.2 The relationship between UESMANN and plain back-propagation solutions

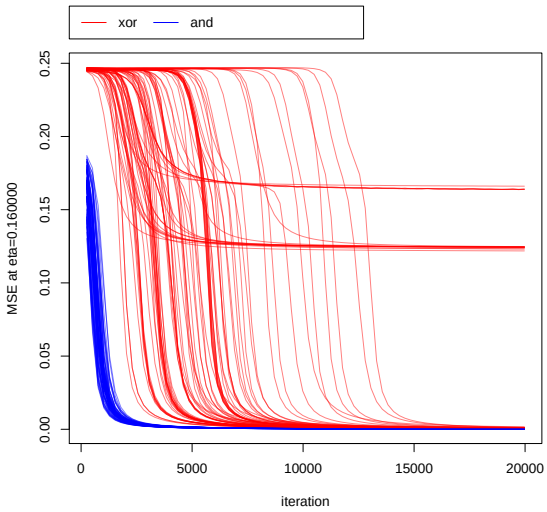
UESMANN networks for $x \oplus y \rightarrow x \wedge y$ may converge to a solution for $x \wedge y$ first, so we might suspect that there exists a solution for $x \wedge y$ which performs $x \oplus y \rightarrow x \wedge y$ when its weights are halved. Fig. 4.26 shows a clustering for plain back-propagation solutions for $x \wedge y$ with the weights halved, with all the UESMANN clusters[6] found for $x \oplus y \rightarrow x \wedge y$ superimposed on it. We can see that there is no UESMANN solution which matches the solutions for $x \wedge y$ with halved weights.

Similarly, no plain back-propagation solution for $x \oplus y$ is a UESMANN solution cluster when its weights are doubled, as Fig. 4.27. Thus the UESMANN solutions

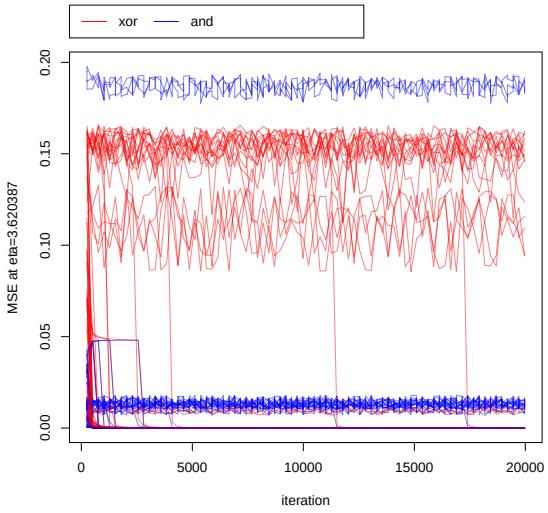
[6]See the following section for how these clusters were derived.



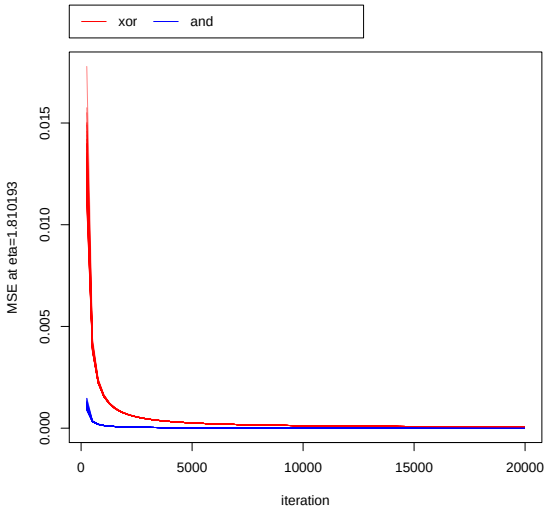
(a) weight blending, $\eta = 0.16$



(b) output blending, $\eta = 0.16$



(c) h-as-input, $\eta = 3.62$



(d) UESMANN, $\eta = 1.81$

Figure 4.25: Convergence behaviour for 100 different networks trained for $x \oplus y \longrightarrow x \wedge y$ at optimal learning rates for four different network types. The plot shows the MSE of the different four different boolean input pairs for both the initial and terminal functions in the pairing separately.

for $x \oplus y \longrightarrow x \wedge y$ are unrelated to the individual back-propagation solutions for $x \oplus y$ and $x \wedge y$.

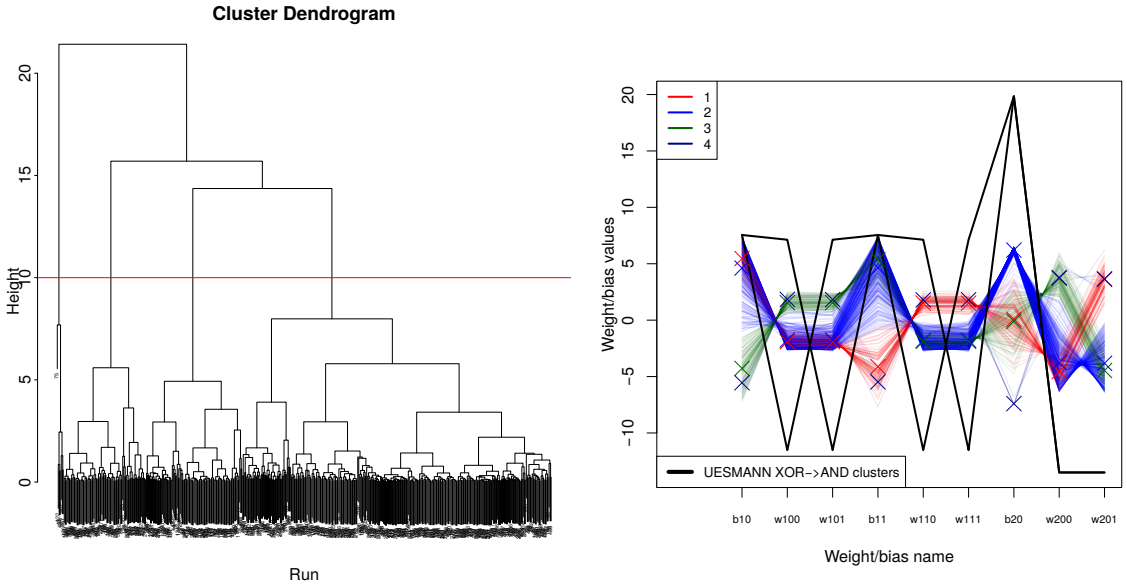


Figure 4.26: Clustering of 500 plain back-propagation solutions for $x \wedge y$ with weights halved, with the UESMANN solutions for $x \oplus y \longrightarrow x \wedge y$ from Fig. 4.29 superimposed over them. The clustering dendrogram and cut point for $x \wedge y$ are shown on the left. Learning rate $\eta = 2$ for all runs, with initial weights from Bishop's Rule. Each network is shown as a polygonal chain linking the values of its parameters. Weights and bias columns are labelled according to the scheme in Fig. 4.18, modified such that `wLIJ` represents w_{ij}^{l}.

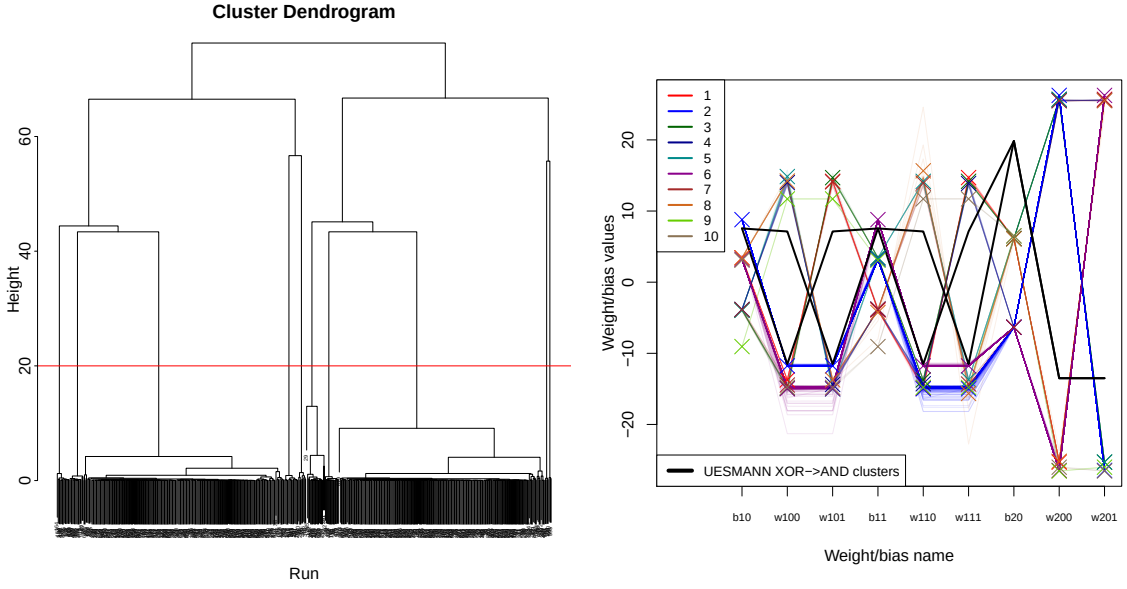


Figure 4.27: Clustering of 500 plain back-propagation solutions for $x \oplus y$ with weights doubled, with the UESMANN solutions for $x \oplus y \longrightarrow x \wedge y$ from Fig. 4.29 superimposed over them. See Fig. 4.26 for more details.

4.5.6.3 Solution clusters for $x \oplus y \rightarrow x \wedge y$

First, we shall establish a more accurate optimal learning rate, and while doing so study the convergence behaviour for this pairing in a little more detail. Fig. 4.24 shows that UESMANN's performance increases with learning rate until $\eta = 1.81$. The next sample, at $\eta = 2.56$, shows a rapid decline. This sudden drop in learning rate suggests that the error surface is moderately complex: as the learning rate becomes too high, the algorithm can no longer find successful minima.

To show this more accurately and find a more accurate optimum learning rate, 500 networks were trained for the pairing at different η, with an increased iteration count of 250000 to hopefully ensure that networks which were converging towards a solution had time to do so. Bishop's rule was once again used for the initial weights, giving the range $[-0.707, 0.707]$. The percentage of networks which converged to a successful minimum was taken, and the results are shown in Fig. 4.28. This shows a rapid drop in performance after $\eta = 2.25$, with the optimum being $\eta = 2$. This is consistent with the plot in Fig. 4.24.

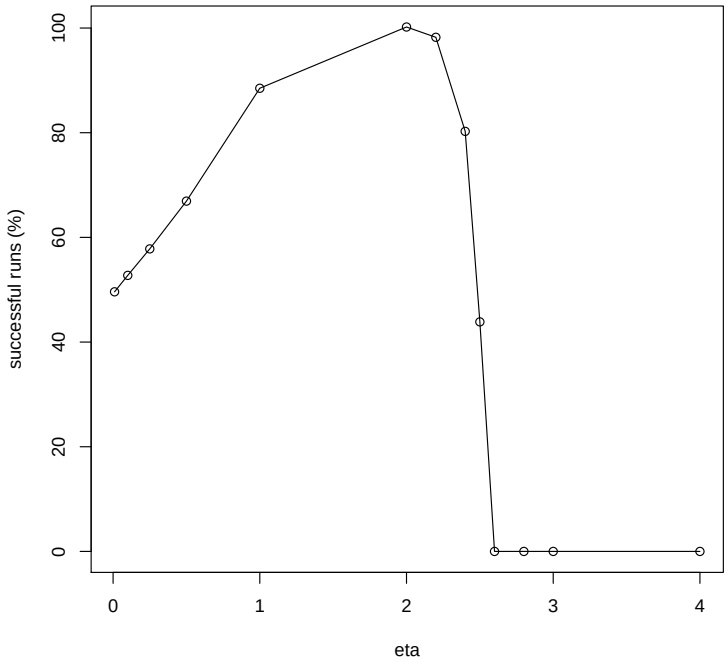


Figure 4.28: Percentage of successful runs (out of 500) for UESMANN $x \oplus y \rightarrow x \wedge y$, given 10^6 training iterations, at different learning rates (η).

With an approximate optimum of $\eta = 2$ established, the same learning rate and initial weight range were used to generate 500 networks, all of which converged to a solution. The weights and biases of the successful networks were clustered

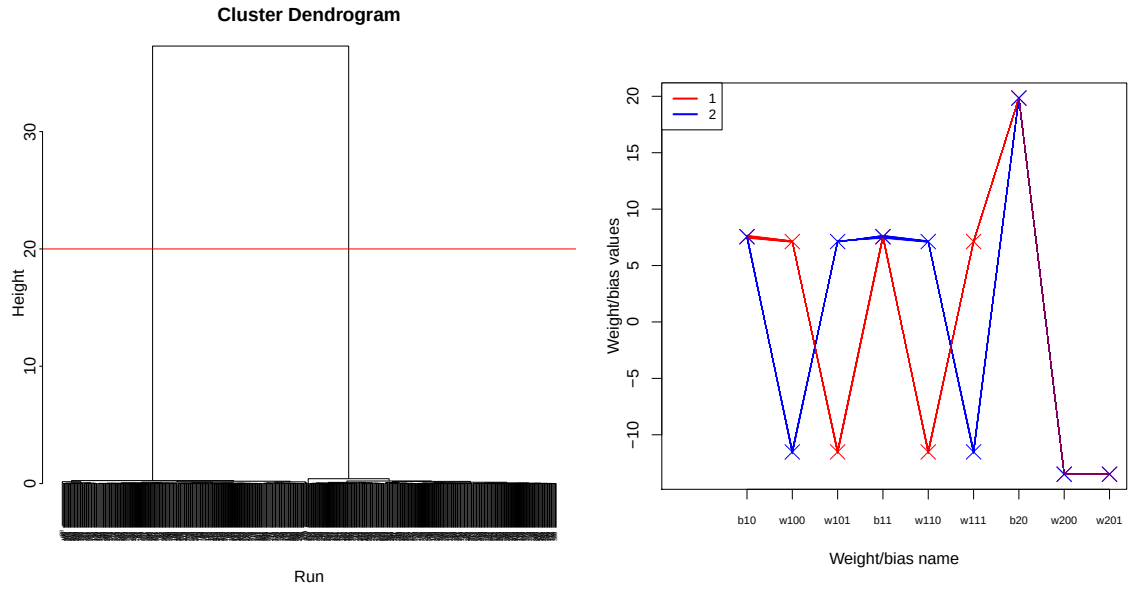


Figure 4.29: Hierarchical clustering of the weights and biases for 500 successfully trained UESMANN networks for $x \oplus y \rightarrow x \wedge y$. Note that the clusters are tight. Each network is shown as a polygonal chain linking the values of its parameters. Weights and bias columns are labelled according to the scheme in Fig. 4.18, modified such that `wLIJ` represents w^l_{ij}.

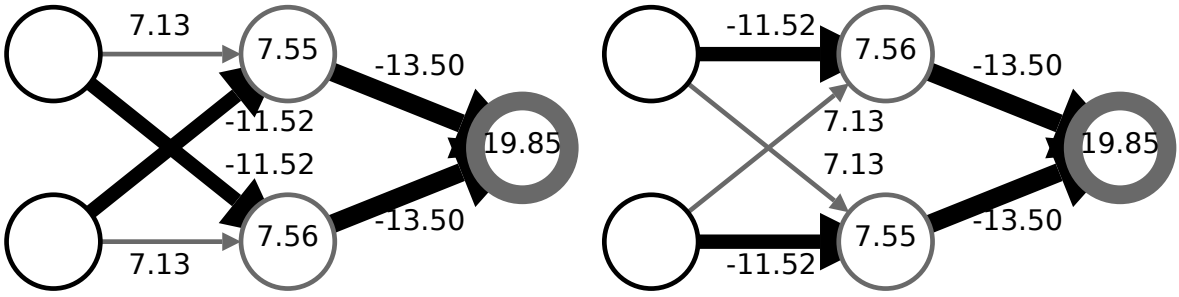
using agglomerative hierarchical clustering, using the average linkage distance as described in Sec. 4.5.3. This produces the clustering in Fig. 4.29, with two clear clusters as indicated in the dendrogram. The values of the parameters at the clusters' centroids are shown in Table 4.4.

Table 4.4: Cluster counts, centroids and standard deviations for centroids for clusters found with hierarchical clustering for $k = 2$, for successful runs of UESMANN $x \oplus y \rightarrow x \wedge y$.

Cluster	count	b^1_0	w^1_{00}	w^1_{01}	b^1_2	w^1_{10}	w^1_{11}	b^2_0	w^2_{00}	w^2_{01}
1	248	7.55	7.13	-11.52	7.56	-11.52	7.13	19.85	-13.50	-13.50
SDs		0.06	0.02	0.07	0.06	0.07	0.02	0.00	0.00	0.00
2	252	7.56	-11.52	7.13	7.55	7.13	-11.52	19.85	-13.50	-13.50
SDs		0.06	0.07	0.02	0.06	0.02	0.07	0.00	0.00	0.00

These clusters are symmetric and tight (note the small standard deviations): in each, both the i and h operations produce the other cluster. The symmetry is evident in Fig. 4.30, which shows both clusters as network diagrams.

It may be that the low number of solution clusters is caused by the limited dimensionality of the solution space — if more weights were provided, for example a network with three hidden nodes, more clusters might result. This is an area



(a) Cluster 1 centroid

(b) Cluster 2 centroid

Figure 4.30: The two types of successful networks produced by UESMANN for XOR→AND with $\eta = 2$, as represented by the centroids of their clusters. Weights and biases tinted and sized by sign and magnitude: red is negative, grey is zero, blue is positive.

where more experiments are required. However, while the clusters are small, they dominate a fairly large area of the solution space as is indicated by the successful training at an appropriate learning rate.

Given the symmetry there is effectively a single solution to the problem, although this results in two minima. These two minima appear with the same frequency (disregarding noise), which means that the same number of networks have fallen into each solution from initial points distributed randomly but evenly around the origin. This suggests that the error surface is symmetric in some plane through the origin.

Because these two minima function identically, we will analyse only cluster 1: cluster 2 will perform in an analogous way. To determine how this network functions — how it performs $x \oplus y$ when $h = 0$ but $x \wedge y$ when $h = 1$, as trained — we will make use of diagrams showing how the activations of the individual nodes in the network vary with the node inputs, and how this changes the function of the entire network[7]. For our network, the set of diagrams for all nodes and the overall network is in Fig. 4.31, and is shown for the modulator at 0, 0.5 and 1. A video of the full transition is available at `https://www.youtube.com/watch?v=ek7u96nhlmE`.

With the modulator low, the hidden layer calculates $x \vee \neg y$ and $\neg x \vee y$ (assuming the two inputs in Fig. 4.30a are x and y reading from top to bottom). Each node in this layer is normally high because of its bias, but the negatively weighted input will pull it low unless the positively-weighted input is also high. The output node also has a large positive bias, and so will be high unless both (negatively weighted)

[7]These diagrams are inspired by Hinton diagrams, originally used by Hinton and Shallice to describe activations in network layers [122]. We use far more points and a much smaller scale to show network behaviour rather than activations.

inputs to it are high, calculating $\neg(h_1 \wedge h_2)$. Thus, the network calculates

$$\overline{(x \vee \neg y) \wedge (\neg x \vee y)} = x \oplus y.$$

We can clearly see these functions in Fig. 4.31, bearing in mind that solid black is 1 and white is 0, while the intersections of the four dotted lines mark the four possible inputs to the boolean system (0,0), (0,1), (1,0) and (1,1).

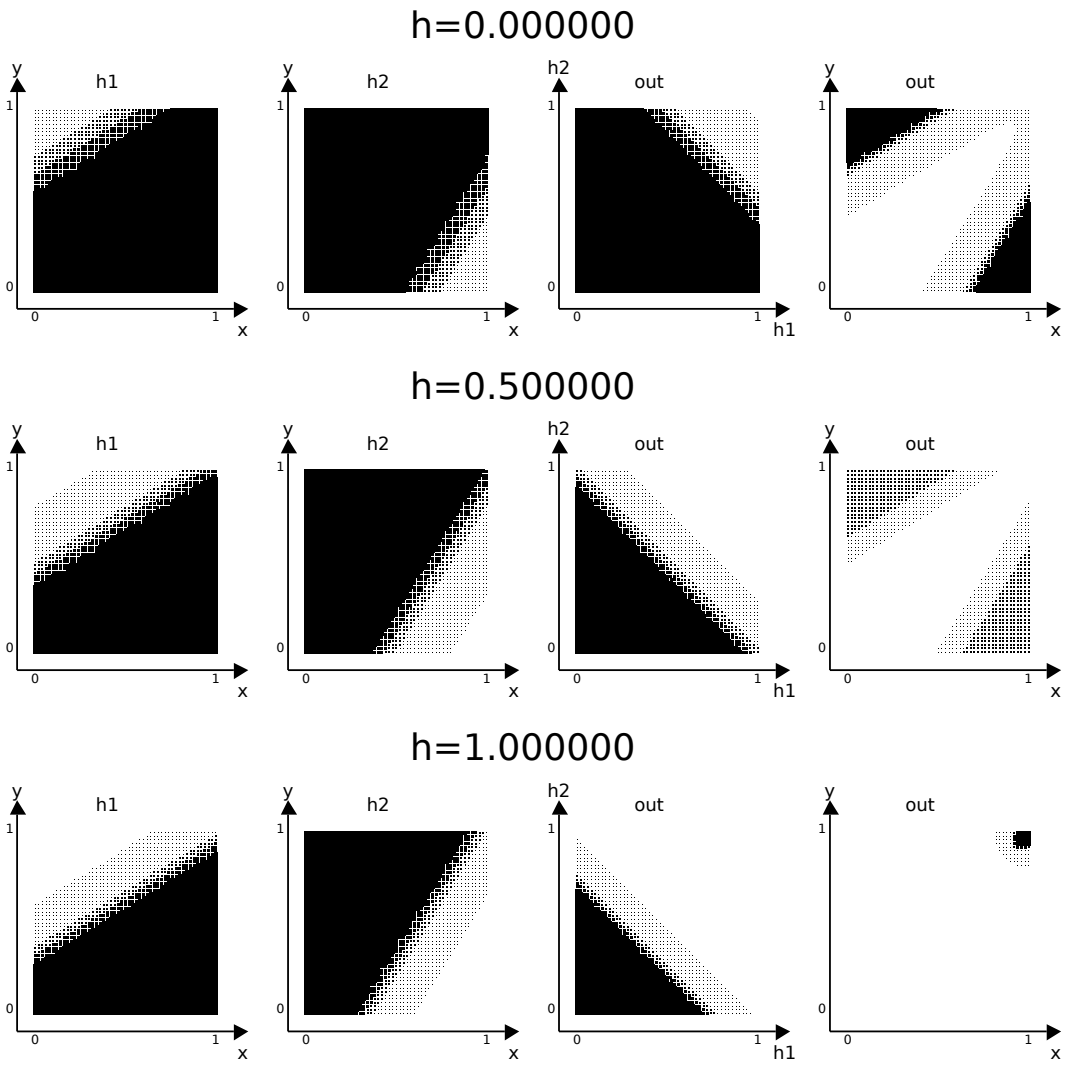


Figure 4.31: Diagrams showing the output of each node of a UESMANN network trained for $x \oplus y \rightarrow x \wedge y$ (at $\eta = 2$) given its inputs, along with the output node given the inputs to the network. The two left-hand plots show the hidden node activations as the inputs x and y change, the third from the left shows the output node as the hidden node outputs change, and the right-hand plot shows the output of the network (i.e. the output node) as the inputs change. The edge length of each square is proportional to the output at that point; white is 0 and solid black is 1. Plots are given for both modulator $h \in \{0, 0.5, 1\}$ to demonstrate how the modulation changes the functions.

We can see that this network is different from most networks generated by plain back-propagation for $x \oplus y$: if we run that algorithm 50000 times with random initial weights each time, this function or its counterpart from the other UESMANN cluster appears only 39 times. It does not appear at all in Fig. 4.29. The most notable point of difference is the high bias on the output node, b_{20}, which does not appear in Fig. 4.29.

With the modulator high, we can see from Fig. 4.31 that the thresholds of the input nodes move closer to zero so that the hidden layer now calculates $\neg y$ and $\neg x$. The output layer's threshold also moves closer to zero, decreasing so that it now calculates $\neg(h_1 \vee h_2)$. Thus the network calculates

$$\overline{\neg x \vee \neg y} = x \wedge y.$$

Of 50000 randomly initialised networks trained for $x \wedge y$ using plain back-propagation, only 2 used this strategy. Thus, the large bias on the output node which was remarked upon earlier is key to the operation of the network. Without the weight modulation, both hidden nodes must be high to overcome this bias and produce a high output. With the weights doubled, only one high hidden node will produce a high output.

Note also how marginal the $h = 1$ solution is: at (1,1), the h_1 node produces an output of ~ 0.2, with the same output being produced by h_2 at (1,1). This is within the unsaturated range of the sigmoid activation function for the $\neg y$ and $\neg x$ functions respectively, and is the result of a small change in bias due to modulation. The resulting overall network behaviour is also marginal: the input (1,1) is only just inside the region where the output is over threshold.

4.5.6.4 Transitional behaviour of $x \oplus y \rightarrow x \wedge y$

Having established that UESMANN converges to solutions more readily than the three other modulatory techniques, and having examined those solutions, we will now compare how the four different network techniques transition: i.e. how their behaviour changes as the modulator h rises from 0 to 1. Until now, we have only considered the behaviour at the extrema. To do this, 12 networks of each type were trained to perform the pairing at the optimal learning rates found above, for 128000 iterations with random initial weights chosen with Bishop's Rule. For these networks, the actual functions evaluated under thresholding at different modulator values were found, and plotted on a number of graphs. The learning rates were determined in previous experiments, and can be found in Fig. 4.25 with the exception of that used for the UESMANN network, which was obtained from Fig. 4.28 ($\eta = 2$). The results are shown in Fig. 4.32.

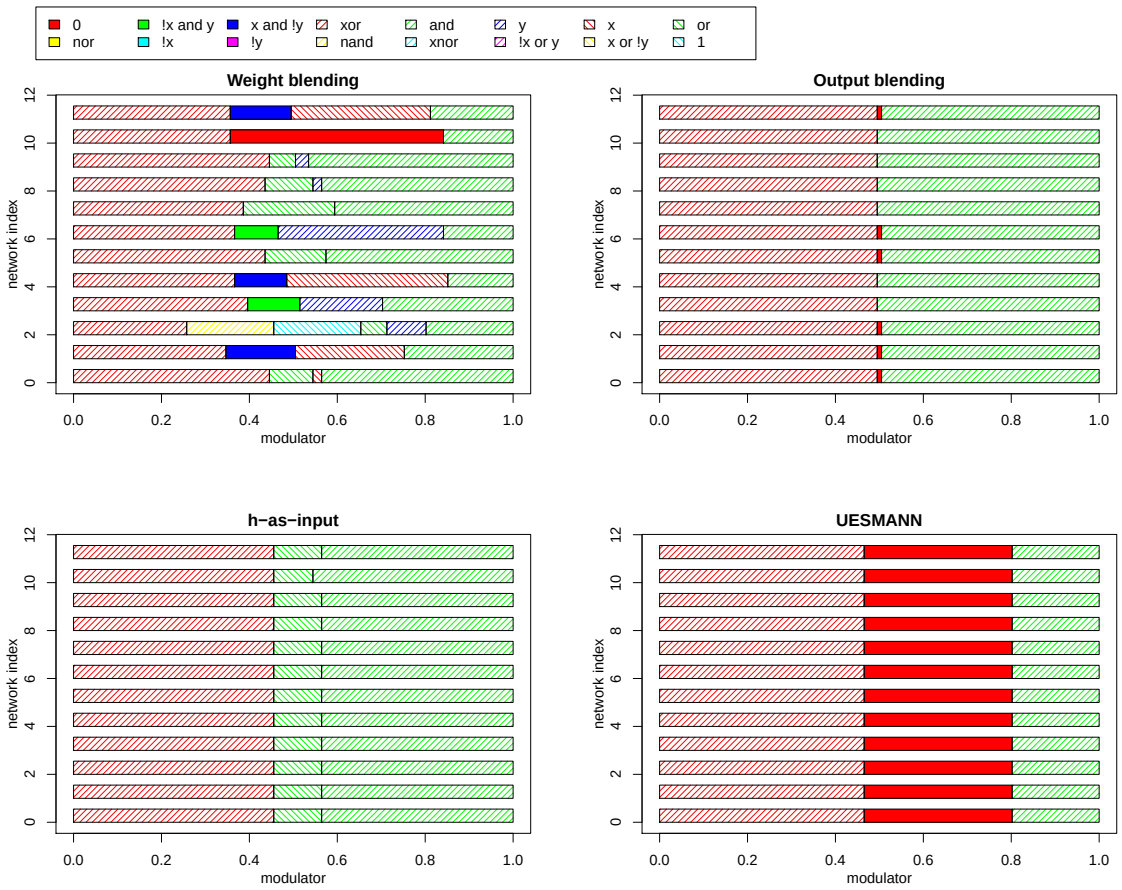
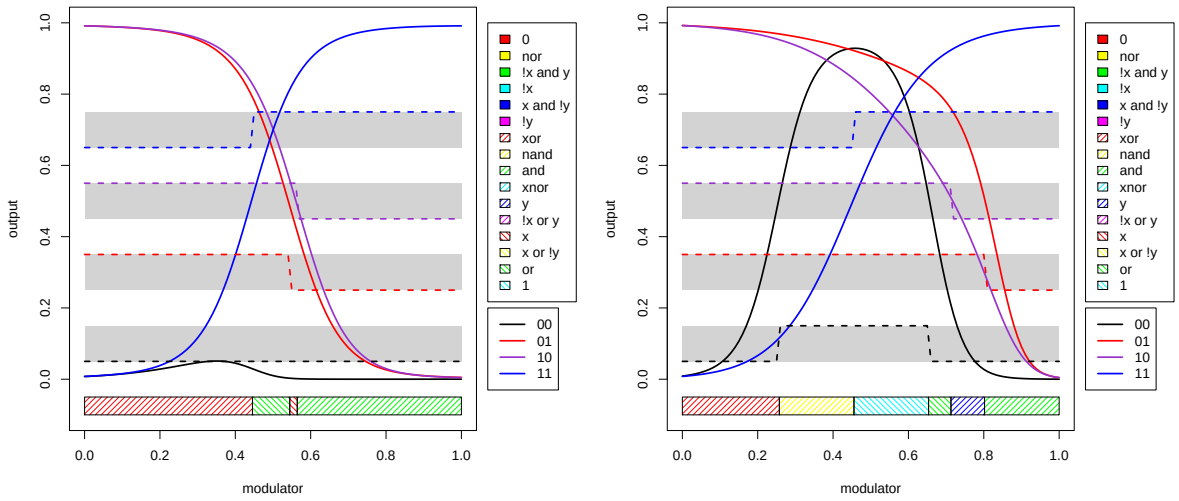


Figure 4.32: Discrete function transition diagrams for different network types trained for $x \oplus y \longrightarrow x \wedge y$ (using the optimal learning rates — see the text). Each diagram shows 12 different networks trained from random initial weights, and how the function it performs under thresholding of the output at 0.5 varies as the modulator changes.

It is immediately obvious that weight blending produces different transitions for each network. For example:

$$\begin{aligned}
&\text{Network 0:} \quad (x \oplus y) \longrightarrow (x \vee y) \longrightarrow x \longrightarrow (x \wedge y) \\
&\text{Network 1:} \quad (x \oplus y) \longrightarrow (x \wedge \neg y) \longrightarrow x \longrightarrow (x \longrightarrow x \wedge y) \\
&\text{Network 2:} \quad (x \oplus y) \longrightarrow \neg(x \wedge y) \longrightarrow T \longrightarrow (x \vee y) \longrightarrow y \longrightarrow (x \wedge y)
\end{aligned}$$

Two of these transitions are shown in more detail in Fig. 4.33 with the actual continuous outputs for each input pairing $x, y \in \{1, 0\}$ also shown, along with their thresholded values. Clearly, these two networks function differently.



(a) Network 0

(b) Network 2

Figure 4.33: Transition behaviour for weight blending networks 0 and 2 from Fig. 4.32. The continuous curves show the output for the different input pairings $x, y \in \{0, 1\}$ as the modulator changes. The dotted curves in grey rectangles show the thresholded output for the input pairing of the same colour — each can be considered to have its own y axis (0,1).

The $x \oplus y$ problem has a number of solutions: Fig. 4.22 shows that these fall into around 10 clusters, dominated by 2. The $x \wedge y$ problem has many solutions which occupy large regions of the solution space in flat regions, as shown by Fig. 4.26. The weight-blending technique trains two networks to perform these functions, and which of the solutions are found depends on the random initial weights. How the system transitions as the modulator changes depends on the inner workings of the two networks of which it is comprised.

This is analogous to the "competing conventions problem" encountered in naïve attempts to train neural networks with evolutionary algorithms [246]. In such systems, attempts to crossover (combine the genomes of) two high-fitness neural networks to produce a good candidate may fail, because the two parent networks may have a fundamentally different functional architecture: they use the nodes at the same positions for different things. Any form of naïve combination of these two networks will result in a network whose behaviour will be hard to predict, and may bear no resemblance to the parent networks' behaviours. Evolutionary algorithms such as NEAT attempt to counter this by tracking each node's history, so that only nodes which are functionally similar are combined (see Sec. 2.3.9.2).

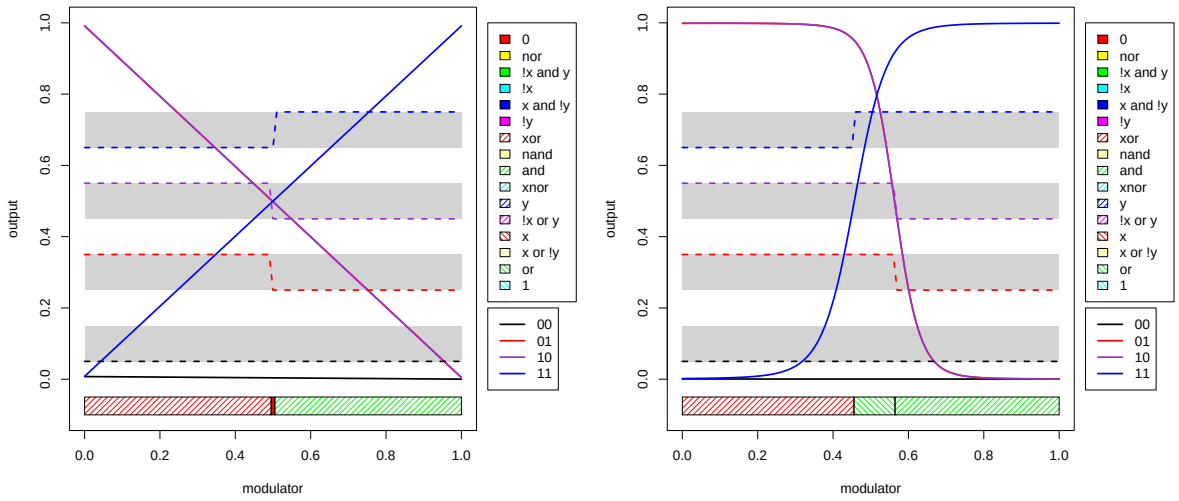
Our weight-blending algorithm is naïve in that we are simply finding a linear interpolation between all the network parameters — finding, for each node, an interpolation of the weights and biases of the two nodes in that position in the parent networks. Thus we may find that the two networks do not combine in a useful way. With boolean problems, the interpolated networks will perform some boolean function, but it will be hard to predict what this function will be. Given that in this particular problem (and probably very many others) the initial networks may be very different for different initial weights, the behaviour of the interpolated networks will also be difficult to predict as, Figs. 4.32 and 4.33 show. This also may explain why the interpolative modulation used by Sauzé and Neal in initial experiments in [243] and in the hexapod of Henley and Barnes [118] produced a "nonlinear response," as discussed in Sec. 2.5.6.

Such a system may arguably have its uses: the unpredictable behaviour of the output network can serve as a source of variety in an evolutionary algorithm, if one were to evolve both networks as part of a single genome (albeit risking the competing conventions problem again when combining genomes for crossover). However, we will discount weight blending as a modulatory method: the intermediate functions are simply too unpredictable.

Moving on to output blending, this works exactly as predicted: the transitions are crisp, and only have a small transition region because the "parent" networks have not converged to a solution which gives exactly $\{0, 1\}$ outputs. This can be seen in Fig. 4.34a, which shows network 0: the outputs for each input pairing transition linearly, and all cross the threshold very close to $h = 0.5$ because the end points are very close to 0 or 1.

In the h-as-input network, a distinct transition region is present roughly symmetrically around $h = 0.5$ which performs $x \vee y$. This is present in all networks but is slightly smaller in network 10, indicating that perhaps there are (at least) two minima, or clusters of identically functioning minima, to which these networks are converging. This is a good transition from $x \oplus y$ to $x \wedge y$, in that each step has only a single change in the truth tables of the performed function. The full transition for a typical network is shown in Fig. 4.34b, which also clearly demonstrates the symmetry around $h = 0.5$. Compare, in particular, with the output blending network in Fig. 4.34a: the non-linearity in the transitions create the transition region.

Finally, the UESMANN network shows a complex but consistent behaviour, which was expected since there are only two solution clusters (from 500 networks), and these are functionally identical. As noted above, this low number of solutions may arise from the low dimensionality of the system. The full transition is shown in



(a) output blending network 0

(b) h-as-input network 0

Figure 4.34: Transitions for output blending and h-as-input networks (both number 0) from Fig. 4.32. See the caption for Fig. 4.33 for explanatory notes. Note, however, that the $\{1, 0\}$ and $\{0, 1\}$ (red and purple) lines are superimposed.

Fig. 4.35. While this transition region is considerably wider than that of h-as-input, it is not as symmetric around $h = 0.5$ and shows a transition to the F function (always false), the truth table of which is a Hamming distance of 2 from the $h = 0$ function[8]. This is in contrast to h-as-input, the transitions of which both have a Hamming distance of 1 (and can therefore be seen as "minimal" transitions), and is symmetric. However, h-as-input requires 2 more weights.

In summary, the UESMANN transition is consistent (in terms of getting the same results for multiple networks trained), and wider than that of output blending and h-as-input. It is, however, markedly asymmetric and produces a somewhat odd choice of transition function due to the asymmetry in the continuous responses (see Fig. 4.25d). Weight blending produces a large set of complex transitions, due to the competing conventions of its "parent" networks being combined in piecewise fashion. Interestingly, UESMANN converges well, until the learning rate increases to the point where the algorithm "skips over" the small solution: although the solutions are small, the combined error surfaces of the two functions sum to a large enough gradient to be followed from the initial point, while $x \oplus y$ has a large, flat

[8]It is worth noting that we have no reason to expect this type of symmetry in UESMANN's transition.

region in the initial weight region which affects the convergence for output (and weight) blending. The h-as-input convergence appears to be affected by a complex error surface.

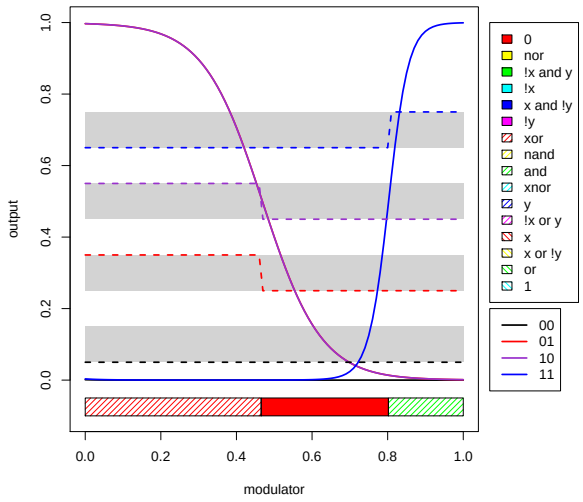


Figure 4.35: Transitions for UESMANN network 0 from Fig. 4.32. See the caption for Fig. 4.33 for explanatory notes. Note, however, that the $\{1, 0\}$ and $\{0, 1\}$ (red and purple) lines are superimposed.

4.5.7 Analysis of $x \wedge y \rightarrow \neg(x \vee y)$ (AND to NOR)

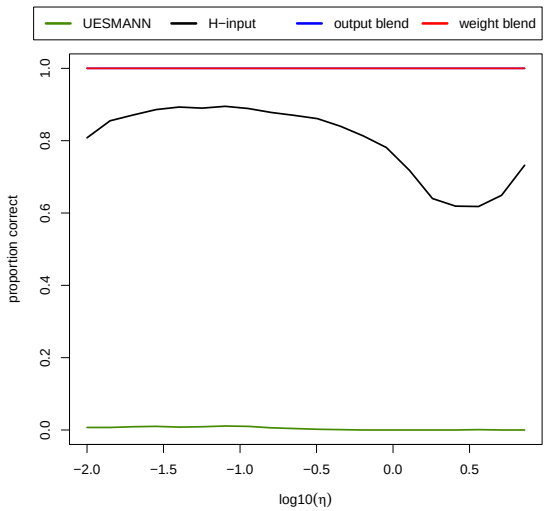
4.5.7.1 Comparison of convergence of different techniques for $x \wedge y \rightarrow \neg(x \vee y)$

We will now examine the convergence and behaviour of this pairing, which is the most difficult pairing for UESMANN to learn despite having a reasonably sized solution in the parameter space (see Fig. 4.4).

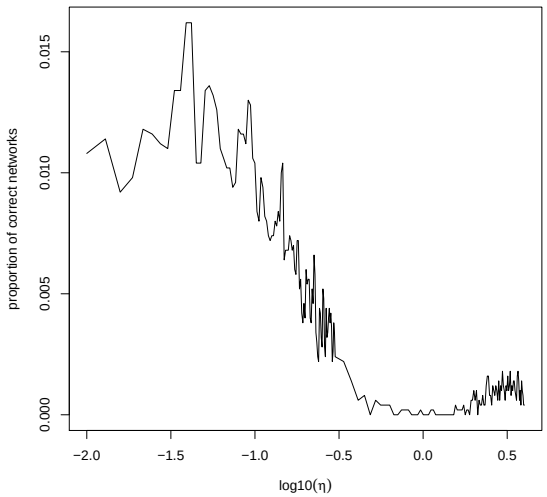
The previous section established that there is no link between the UESMANN solution for $f \rightarrow g$ and the common solutions (by back-propagation) for f and g in the case of $x \oplus y \rightarrow x \wedge y$. That is, there is no solution for $x \oplus y$ which, with weights doubled, gives a solution for $x \oplus y \rightarrow x \wedge y$; and there is no solution for $x \wedge y$ which, with weights halved gives such a solution (see Sec. 4.5.6.2).

To establish a suitable learning rate for the different network types, 100 networks were trained for 125000 iterations at varying η as in the previous experiments, with the proportion of "correct" networks being recorded. The results are shown in Fig. 4.36a. Output blending and weight blending have the same perfect success at all learning rates: individually, these are easy functions for a network to learn. The h-as-input networks have a variable performance, while UESMANN performs poorly.

A more detail learning rate plot was made for UESMANN, using 10^6 iterations for 5000 networks: this is shown in Fig. 4.36b.



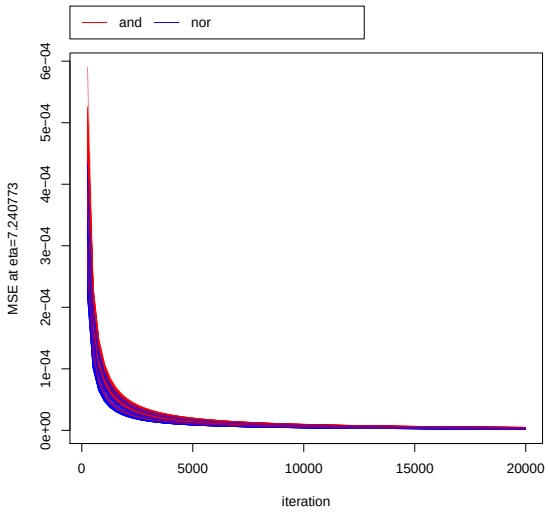
(a) All four methods, 100 networks for 125000 iterations



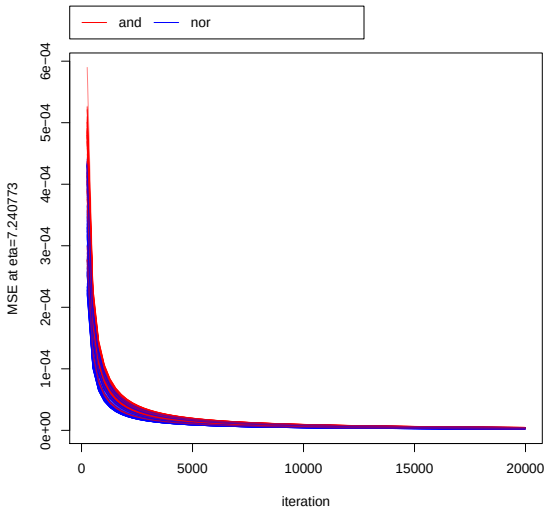
(b) UESMANN, 5000 networks for 10^6 iterations

Figure 4.36: Proportion of networks trained for $x \wedge y \rightarrow \neg(x \vee y)$ which find the correct solution at different learning rates η using four different types of modulatory network. Bishop's method was used for the initial weights. Note that weight and output blending are the same. A separate plot is shown for UESMANN with a different y axis; this uses a higher number of iterations and more individual networks.

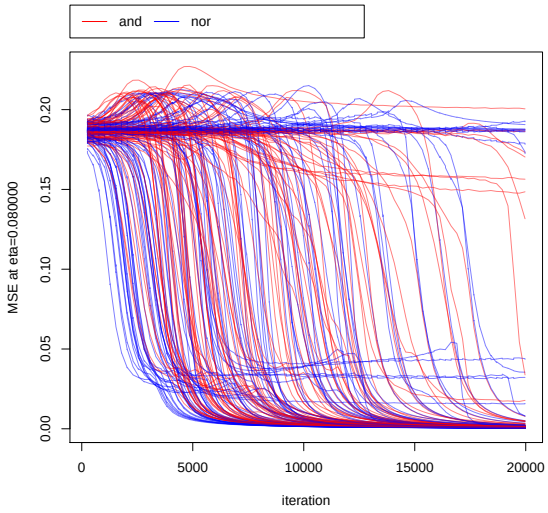
The latter figure shows a large amount of noise, because we are looking at events which occur rarely. The networks succeed less than 1% of the time, with each success dependent on essentially random conditions (the initial weights). If we consider the success or failure of a network as a Bernoulli trial with probability $P = 0.01$, the actual number of successes will follow a binomial distribution with $n = 5000$ and $P = 0.01$. This will give a mean success count of $nP = 50$ with a standard deviation of $\sqrt{nP(1-P)} = 7.036$ [195]. This is a wide distribution, hence the noise. For UESMANN, we will use a learning rate of $\eta = 0.039$, which is a maximum in Fig. 4.36b, with the understanding that this is likely not a truly optimal learning rate due to the noise. For weight and output blending we will use the largest learning rate attempted (since they all work well), and for h-as-input we will use 0.08 from the data behind Fig. 4.36a. The results (and selected learning rates) are shown in Fig. 4.37.



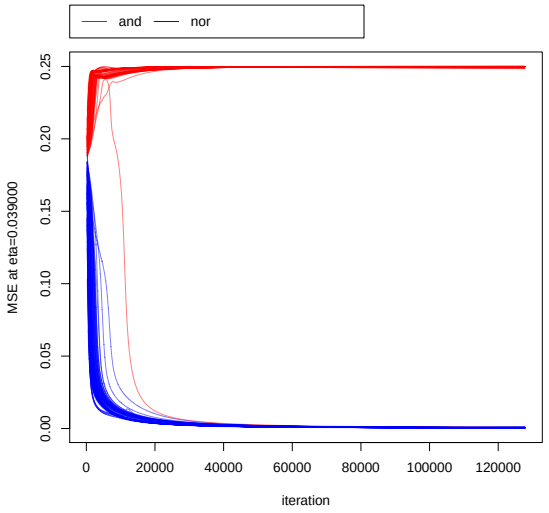
(a) weight blending, $\eta = 7.24$



(b) output blending, $\eta = 7.24$



(c) h-as-input, $\eta = 0.08$



(d) UESMANN, $\eta = 0.039$

Figure 4.37: Convergence behaviour for 100 different networks trained for $x \wedge y \longrightarrow \neg(x \vee y)$ at good learning rates for four different network types. The plot shows the MSE of the different four different boolean input pairs for both the initial and terminal functions in the pairing separately.

As before, the weight blending and output blending plots are identical, because we are looking at the endpoints of the transitions which are trained separately and identically in those two network types. Both converge rapidly to a solution, as should be expected for such simple functions. The h-as-input plot is more complex,

and appears to show a flat region near the origin which takes some time to escape, rather like the earlier $x \oplus y$ plots. Indeed, some networks fail to find a region within which both functions are evaluated correctly, instead finding a minimum in which one functions works but not the other. UESMANN networks rapidly converge on a solution for $\neg(x \vee y)$ but of the 100 runs only one converges on $x \wedge y$; this network has already arrived at a solution for the former function. It may be that "incorrect" local minima which solve for $\neg(x \vee y)$ dominate the solution space, with narrow areas leading to a solution for $x \wedge y$ within that space.

4.5.7.2 Solution clusters for $x \wedge y \rightarrow \neg(x \vee y)$

We now turn to the UESMANN solution clusters for this pairing. We have already established a reasonable learning rate of 0.039, although this is only an approximate optimum. UESMANN networks were trained at this learning rate, using Bishop's rule for the initial weights and biases. For this experiment, 250000 iterations through the training set were used to ensure convergence. To obtain 500 networks, 41659 networks needed to be trained, giving a success rate of around 1.2%: these successful networks were clustered using agglomerative hierarchical clustering with the average linking distance, as described in Sec. 4.5.3.

The solutions do not fall into clusters as neatly as $x \oplus y \rightarrow x \wedge y$, which is interesting in itself: there may more minima available, although the learning algorithm has difficulty finding them, as we have seen. The clusters are also wider than those for $x \oplus y \rightarrow x \wedge y$, and the distributions of the individual parameters tend to be skewed away from zero, as can be seen particularly clearly for w^2_{00} in cluster 2. A cut of 8 clusters was selected from the dendrogram; although 2 gives a larger distance range, this is not enough clusters for a clear view of the solutions. The results are shown in Fig. 4.38, and in tabular form in Table 4.5. Grouping the clusters using the symmetries of Sec. 4.5.2, we obtain Fig. 4.39.

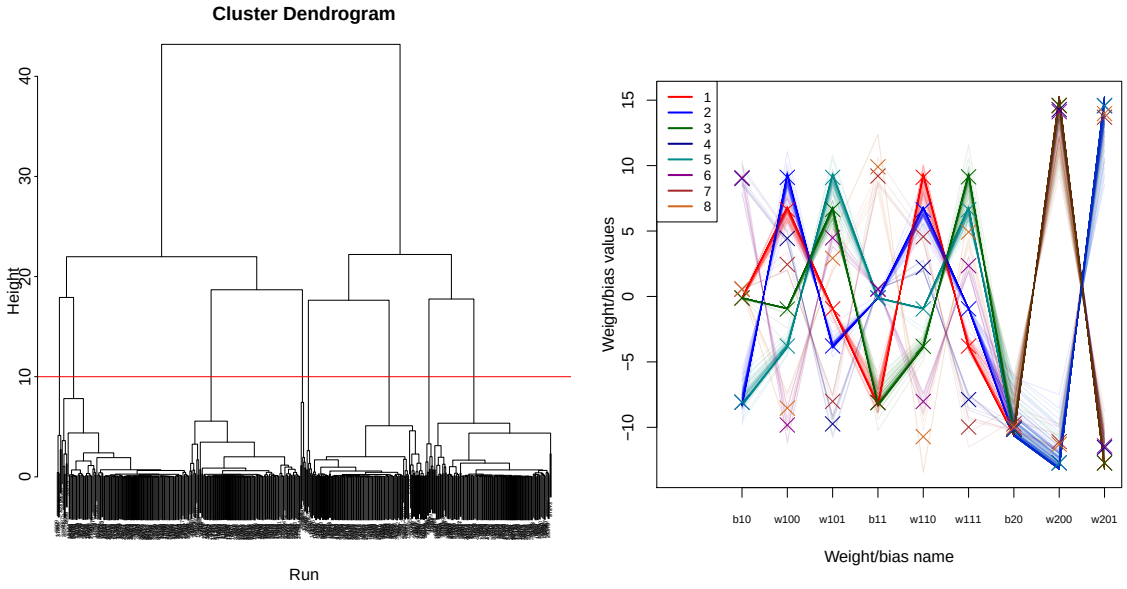


Figure 4.38: Hierarchical clustering of the weights and biases for 500 UESMANN networks successfully trained on $x \wedge y \longrightarrow \neg(x \vee y)$. Each network is shown as a polygonal chain linking the values of its parameters. Weights and bias columns are labelled according to the scheme in Fig. 4.18, modified such that `wLIJ` represents w^l_{ij}.

Table 4.5: Cluster counts, centroids and standard deviations for centroids for clusters found with hierarchical clustering for $k = 2$, for successful runs of UESMANN $x \wedge y \longrightarrow \neg(x \vee y)$.

Cluster	count	b^1_0	w^1_{00}	w^1_{01}	b^1_2	w^1_{10}	w^1_{11}	b^2_0	w^2_{00}	w^2_{01}
1	113	-0.12	6.63	-0.95	-8.09	9.11	-3.79	-10.10	14.57	-12.71
SDs		0.02	0.40	0.13	0.52	0.56	0.21	0.98	1.23	0.84
2	129	-8.08	9.10	-3.79	-0.12	6.62	-0.95	-10.12	-12.71	14.59
SDs		0.53	0.60	0.25	0.03	0.45	0.20	0.95	0.91	1.24
3	119	-0.12	-0.97	6.65	-8.12	-3.81	9.14	-10.14	14.61	-12.74
SDs		0.02	0.28	0.45	0.52	0.22	0.56	0.92	1.17	0.80
4	7	9.01	4.43	-9.74	0.57	2.22	-7.88	-10.16	14.28	-11.55
SDs		0.59	0.35	0.67	0.01	0.37	0.50	0.34	0.52	0.45
5	112	-8.08	-3.79	9.10	-0.12	-0.95	6.63	-10.10	-12.72	14.57
SDs		0.51	0.21	0.54	0.02	0.23	0.44	0.93	0.80	1.17
6	9	9.07	-9.82	4.48	0.57	-8.03	2.35	-10.04	14.12	-11.42
SDs		0.87	0.97	0.51	0.01	0.59	0.45	0.50	0.78	0.66
7	5	0.56	2.44	-8.02	9.20	4.56	-9.98	-9.74	-11.10	13.69
SDs		0.01	0.64	0.68	0.86	0.52	0.98	0.78	0.91	1.12
8	6	0.56	-8.53	2.92	9.90	-10.72	4.92	-9.96	-11.30	13.97
SDs		0.02	1.32	1.33	1.97	2.18	1.11	0.65	0.89	1.05

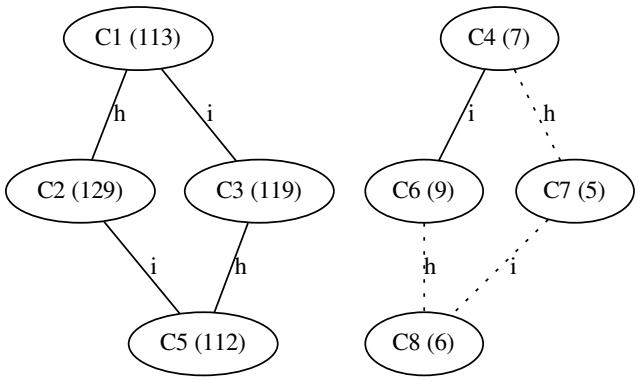


Figure 4.39: Relationships between the clusters in 500 solutions found for $x \wedge y \longrightarrow \neg(x \vee y)$ at $\eta = 0.039$ with 125000 iterations, in terms of the symmetries discussed in Sec. 4.5.2. The number of nodes in each cluster is shown in brackets, with the relationship type h (hidden node swap) or i (input node swap) shown as a label on the relationship edge. If the relationship is weak (mean distance greater than 0.1) the edge is dotted. In this case, this is likely to be due to the low number of samples in the right-hand group.

The clusters fall into two distinct groups: one consisting of clusters 1, 2, 3 and 5; and another consisting of clusters 4, 6, 7 and 8. The latter clusters are an order of magnitude smaller than the former. There are similarities between the two groups in the output: each cluster in the second group is close to a cluster in the first, but with all parameters except b_0^2 negated. However, these distances are larger than Fig. 4.38 suggests. Consider clusters 1 and 8:

Cluster	count	b_0^1	w_{00}^1	w_{01}^1	b_2^1	w_{10}^1	w_{11}^1	b_0^2	w_{00}^2	w_{01}^2
1	113	-0.12	6.63	-0.95	-8.09	9.11	-3.79	-10.10	14.57	-12.71
8	6	0.56	-8.53	2.92	9.90	-10.72	4.92	-9.96	-11.30	13.97

The cluster parameters are clearly quite different in magnitude. Cluster 1 has b_0^1 with a mean of -0.12, and a range of $[-0.204, -0.088]$, but the corresponding bias in cluster 8 is well outside the negation of this range.

Indeed, if we take any cluster centroid (all cluster centroids perform the correct pairing $x \wedge y \rightarrow \neg(x \vee y)$) and generate a new network by negating all parameters but b_0^2, we find that the resulting networks perform $x \wedge y \rightarrow F$. Therefore we will continue to treat these groups of clusters as distinct.

Because these clusters are a little more complex than those for $x \oplus y \rightarrow x \wedge y$ in their operation, and do not consist of entirely boolean nodes, we will show cluster function in more detail. Although it appears from Fig. 4.39 that there are two groups of clusters which may function differently, we will only analyse a single cluster: all clusters within the larger group (1, 2, 3, 5) are equivalent by symmetry; and the clusters of the smaller group probably work in a related fashion. We are only interested in the details of how the clusters behave because it might tell us more about why this particular pairing is so difficult to train. We have chosen cluster 1 as a representative of the larger group of clusters: Fig. 4.40 shows how the cluster behaves at $h = 0$ for every possible boolean input.

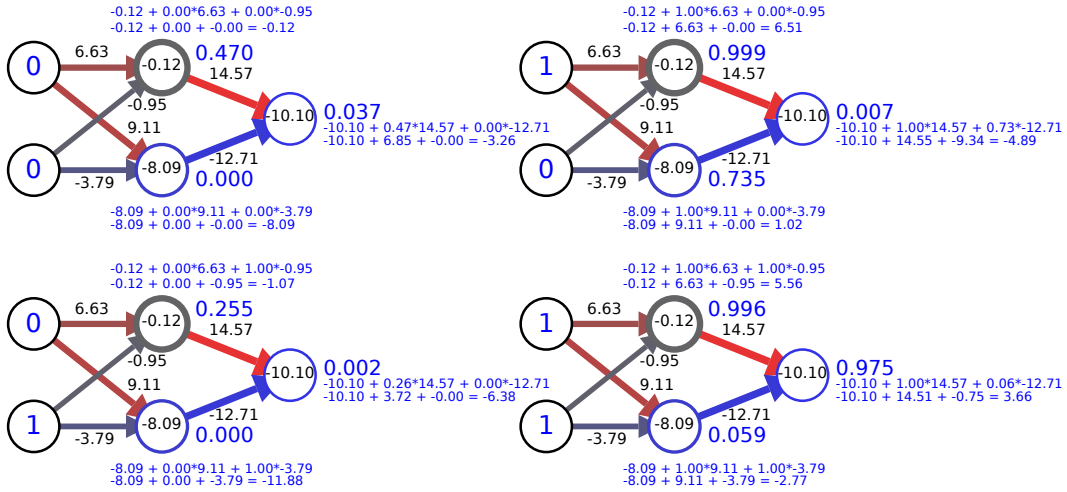


Figure 4.40: Centroid for $x \wedge y \longrightarrow \neg(x \vee y)$ cluster 1 analysed at $h = 0$. The network's function is shown for all possible boolean inputs. The small blue figures show the calculation at each node in two stages, while the large blue figures show the output of the node post-sigmoid. Weights and biases are shown in black.

In order for the output node to be high at $h = 0$, the upper hidden node must be high and the lower hidden node low (since the lower node inhibits the output):

- If both inputs are low, the upper node will have a mid-range output (from its small bias), and the lower node will be pulled low (having a high negative bias). Thus there is no inhibitory effect from the lower node, but the upper node's low activation is insufficient – even with its high weight – to overcome the output's bias.
- If only the upper input is high, both hidden nodes will be driven high because of the high weights from that input, and but now the lower inhibitory node keeps the output low.
- If only the lower input is high, the inhibitory effect of that input's weights will keep both nodes low (although the upper node still outputs a low positive value). The upper node's high weight into the output is still not sufficient to overcome the high negative bias, however.
- If both inputs are high, the upper node is high but the lower node is low – it will no longer inhibit the output, which goes high.

At $h = 1$ the network functions as shown in Fig. 4.41. Here, we see the negation of the previous function – the output is only high if the inputs are both low:

- If both inputs are low, the mid-range output from the upper hidden node is now sufficient to drive the output high (since the weight is doubled).

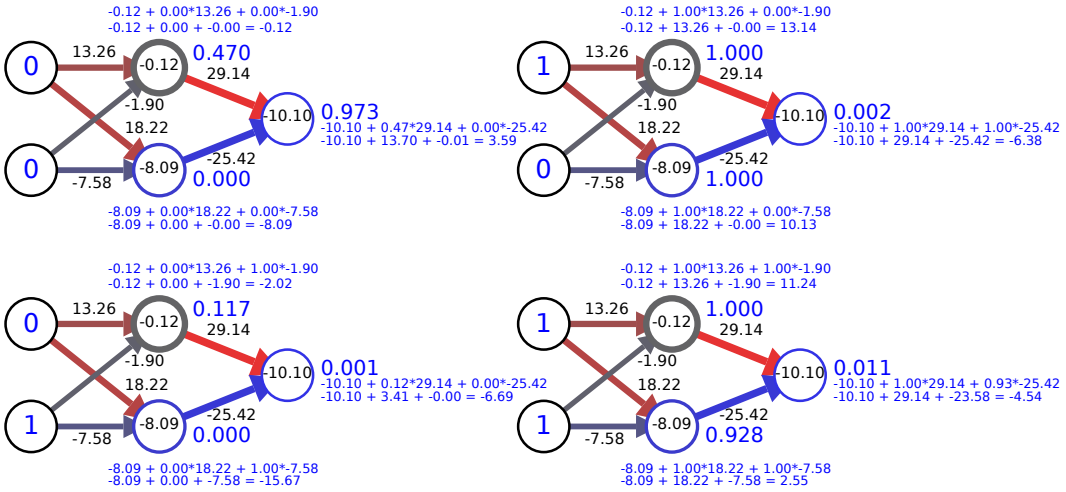


Figure 4.41: Centroid for $x \wedge y \longrightarrow \neg(x \vee y)$ cluster 1 analysed at $h = 1$. The network's function is shown for all possible boolean inputs. The small blue figures show the calculation at each node in two stages, while the large blue figures show the output of the node post-sigmoid. Weights and biases are shown in black.

- If the upper input is high, both the lower and upper hidden nodes are pulled high, but the lower inhibits the output, and the sum remains too low to overcome the bias.
- If the lower input is high, both hidden nodes are pulled low and the output is low.
- If both inputs are high, once again the hidden nodes are both high, and the lower node inhibits the upper so that the sum cannot overcome the bias.

The two h values differ in how they deal with the cases where both inputs are the same. At $h = 0$, if both inputs are high the lower hidden node remains low because of its bias, so it does not inhibit the output being driven high by the upper node. At $h = 1$ the lower node is now driven high by the doubled weight of the upper input, inhibiting the output. In the case where both inputs are low, at both h values the lower hidden node is saturated at zero, so it has no inhibitory effect. When $h = 0$, the upper node's low activation level from its low negative bias is insufficient to drive the output high. When $h = 1$ this low activation now drives the output high due to the doubling of the weight.

Fig. 4.42 shows this transition using the method used for Fig. 4.31. It is clear from this figure that the upper hidden node — hidden node 1 in Fig. 4.42 — hardly changes under modulation due to the wide transition region, while changes in the other hidden node and the output node are much more significant. Indeed, if modulation is performed but weights w^1_{00} and w^1_{01} are left unmodified, the network

still performs the pairing. The effect of the transition as a whole is to shift a ridge of activation towards $x = 0$ while narrowing it (due to increased saturation in all nodes).

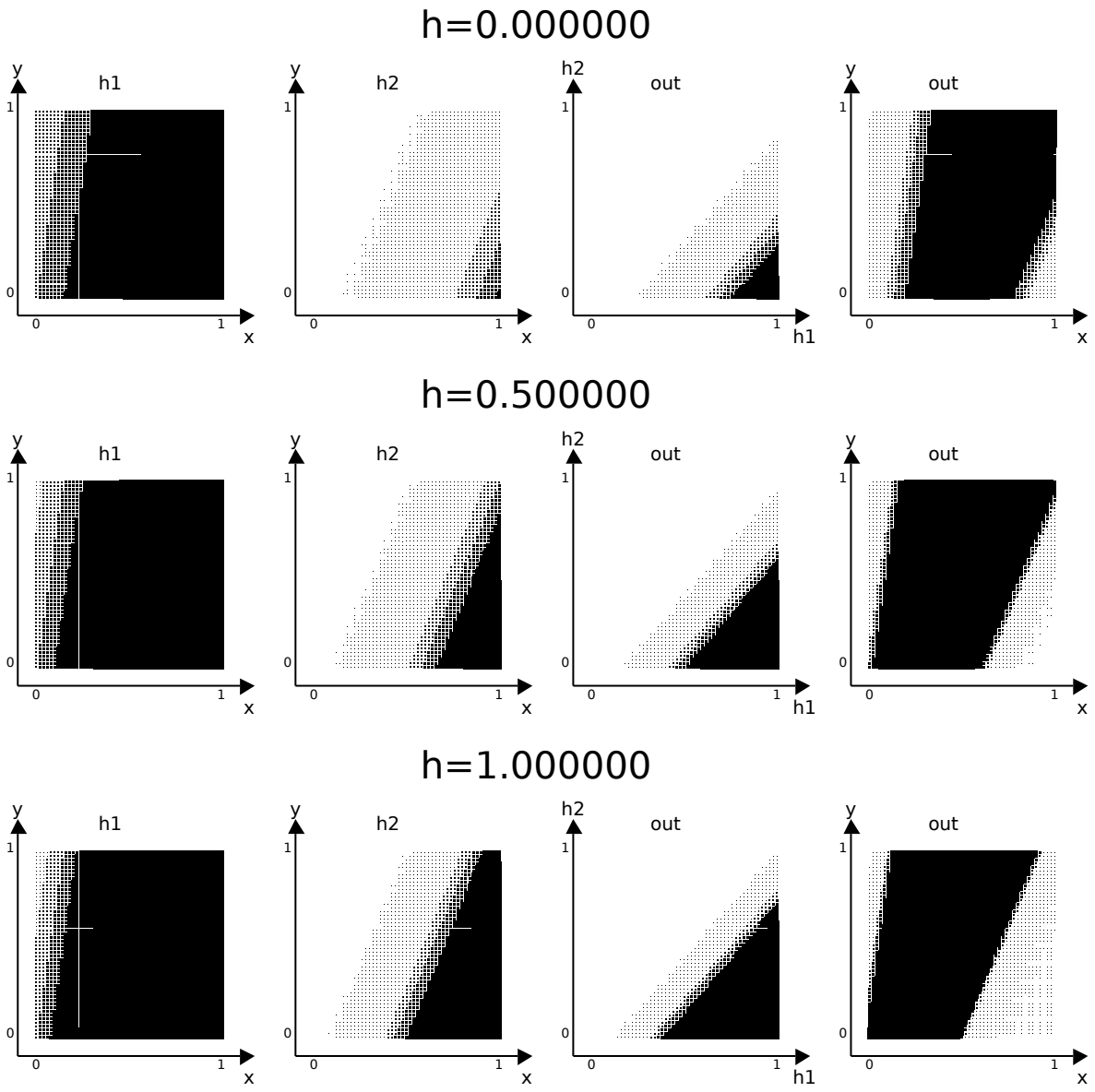


Figure 4.42: Diagrams showing the output of each node of the centroid of cluster 1 of $x \wedge y \rightarrow \neg(x \vee y)$ given its inputs, along with the output node given the inputs to the network. See Fig. 4.31 for explanatory notes.

In order to perform the pairing correctly, this ridge of activation must cover only (1,1) when $h = 0$ and shift to cover only (0,0) when $h = 1$, which places some constraints upon it. However, these constraints are insufficient to explain why the pairing is so hard to train — any constraints here would also limit its frequency in the Monte Carlo simulations, where it is only the 24th rarest pairing.

4.5.7.3 Transitional behaviour for $x \wedge y \rightarrow \neg(x \vee y)$

Before we look further into the problem of why this pairing is so difficult to train, we will analyse the transitions of the different network types. These are shown in Fig. 4.43 using the same conventions as Fig. 4.32. However, the UESMANN networks were selected differently. In the previous pairing, there are only two clusters, and they behave identically (albeit with hidden nodes and inputs swapped). Thus all the networks work the same way, and produce the same transition. This pairing has eight different clusters belonging to two groups. Therefore, rather than selecting 12 random networks to show transitions, we use the centroids of each of the clusters.

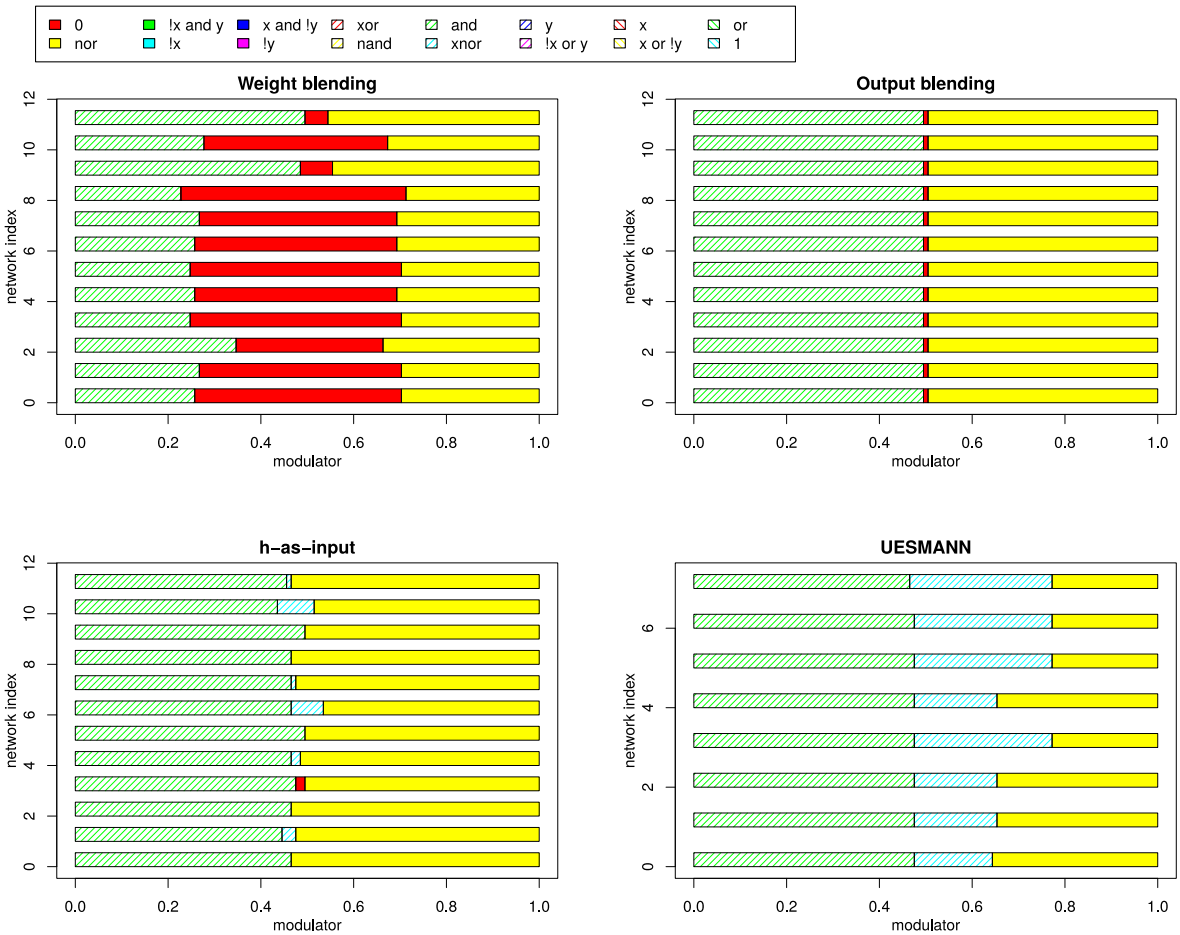


Figure 4.43: Discrete function transition diagrams for different network types trained for $x \wedge y \rightarrow \neg(x \vee y)$ (using the optimal learning rates — see the text). The diagram shows a number of different networks trained from random initial weights, and how the function it performs under thresholding of the output at 0.5 varies as the modulator changes. The UESMANN network plot uses the centroids of the clusters found in the previous section, while the other plots use 12 networks from random initial weights.

We see that both output blending and h-as-input produce narrow transition regions. This is expected for output blending, but in the previous pairing h-as-

input produced a wider transition (see Fig. 4.32). The h-as-input transitions are also inconsistent, suggesting that there may be a large number of solutions to which the system converges: only two solutions appear in the previous pairing. Weight blending does produce functionally consistent solutions, but of different transition widths. Finally, UESMANN is again consistent in the nature of the functions involved in the transitions, transitioning through the XNOR function $\neg(x \oplus y)$. This is a "sensible" transition, in that it is the minimal Hamming distance from the endpoint functions. There is slight inconsistency in the transition widths: the transition width for clusters 1, 2, 3 and 5 is less than that for clusters 4, 6, 7 and 8. This reflects that the former clusters (which belong to the more populated group in Fig. 4.39) may have a qualitatively different solution from those in the smaller cluster group.

It is notable that the UESMANN networks for both this pairing and $x \oplus y \rightarrow x \wedge y$ have a marked "preference" for the $h = 0$ endpoint during the transition. In both cases, the transition does not switch over to the intermediate function until slightly less than 0.5. The reason for this apparent bias in favour of the $h = 1$ function is currently unclear, and requires testing multiple networks for all pairings to establish whether it is a general property for the boolean functions. This is left for future study, particularly if such a bias is evident in more complex problems.

4.5.7.4 Why is this pairing so difficult to train?

We noted above that $x \wedge y \rightarrow \neg(x \vee y)$ is more difficult to train than its frequency in Monte Carlo simulations would suggest, but have yet to find a reason for this. One possible approach is to directly examine the error surface. This is actually a 9-dimensional scalar field, where the dimensions are the weights and biases and the value at each point is the mean squared error of the output of the network at that point. This is the surface down which the back-propagation algorithm should descend. This can be done by viewing 2D slices of this field by varying two of the parameters and holding the rest constant. It should be noted that this can be deceptive: intuitions from single 2D slices in this sort of analysis often do not apply to the entire field because the high dimensionality gives more routes around an apparent local minimum [72].

Slices through the error surface were made by finding a solution and then varying pairs of parameters around that solution while the other parameters were held constant. The solution used was a cluster centroid as found in earlier experiments. For each modified pair of parameter values, the network is run on all four boolean combinations and the mean of the squared errors of the results evaluated. In the case of UESMANN this was done twice, for $h = 0$ and $h = 1$. Because UESMANN

effectively follows the sum of the two error surfaces (in the limit as $\eta \to 0$), we will show the sum of these two processes.

A large number of plots at various slices through the error surface were made for both $x \wedge y \to \neg(x \vee y)$ and the easier $x \oplus y \to x \wedge y$, shown in Figs. 4.44 and Figs. 4.45 respectively. These were inconclusive: the output layers for both seem to have smoother gradients, but the hidden layers are made up of plateaux intercut with trenches; there is no obvious difference between the natures of the topography in each pairing. However, there is a ridge between the start region (marked by a square) and the solution (the origin) in the plot for w^1_{10} against w^1_{00} for $x \wedge y \to \neg(x \vee y)$ (the top-right plot of Fig. 4.44) which may contribute to its difficulty, although it is quite possible that there is a way around this ridge which is not apparent from the limited view provided by the 2D slices.

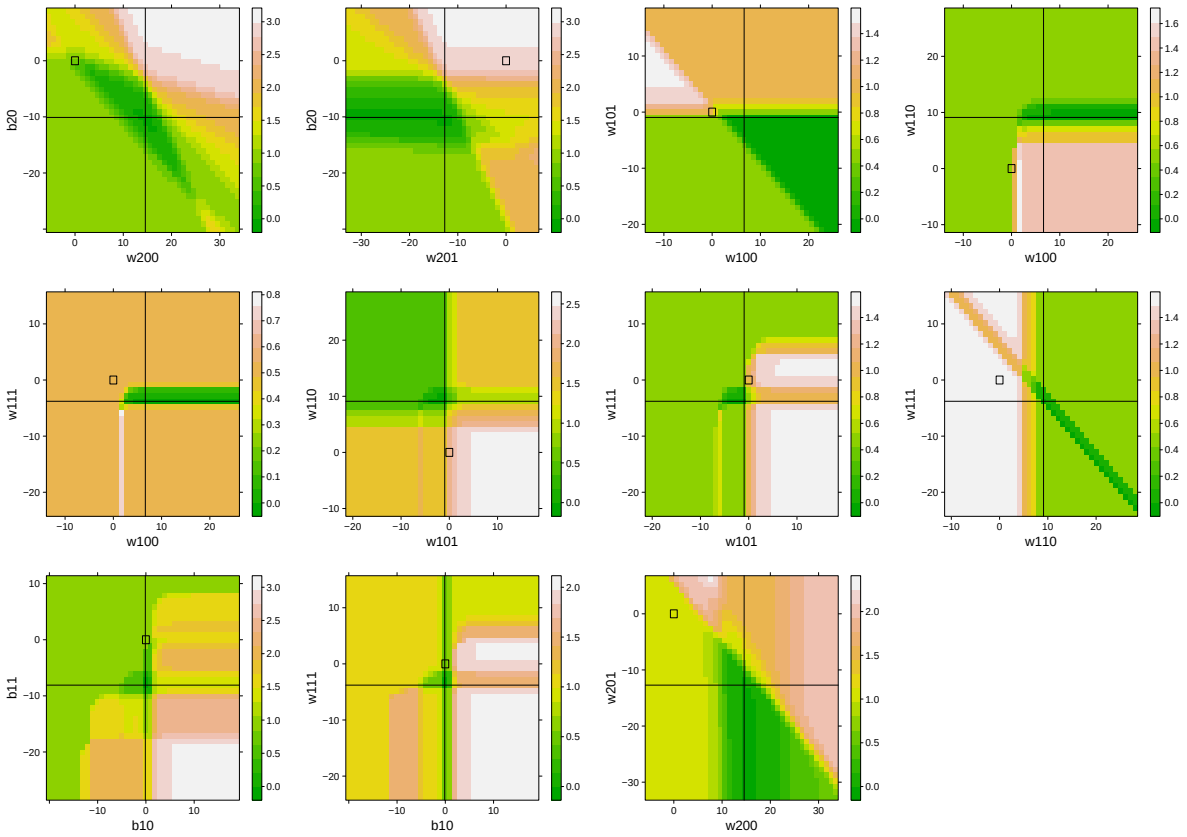


Figure 4.44: Error volume slices for the successful $x \wedge y \to \neg(x \vee y)$ solution given by cluster 1, showing the error surface for two weights holding other parameters constant. Range large to ensure the random start range is included, which is marked by a square. the centre of the surface is the solution.

Noting this ridge and considering that there may be other similar complexities, an attempt was made to train multiple UESMANN networks to perform $x \wedge y \to \neg(x \vee y)$ with the same parameters ($\eta = 0.039$, 125000 iterations), but with a larger initial

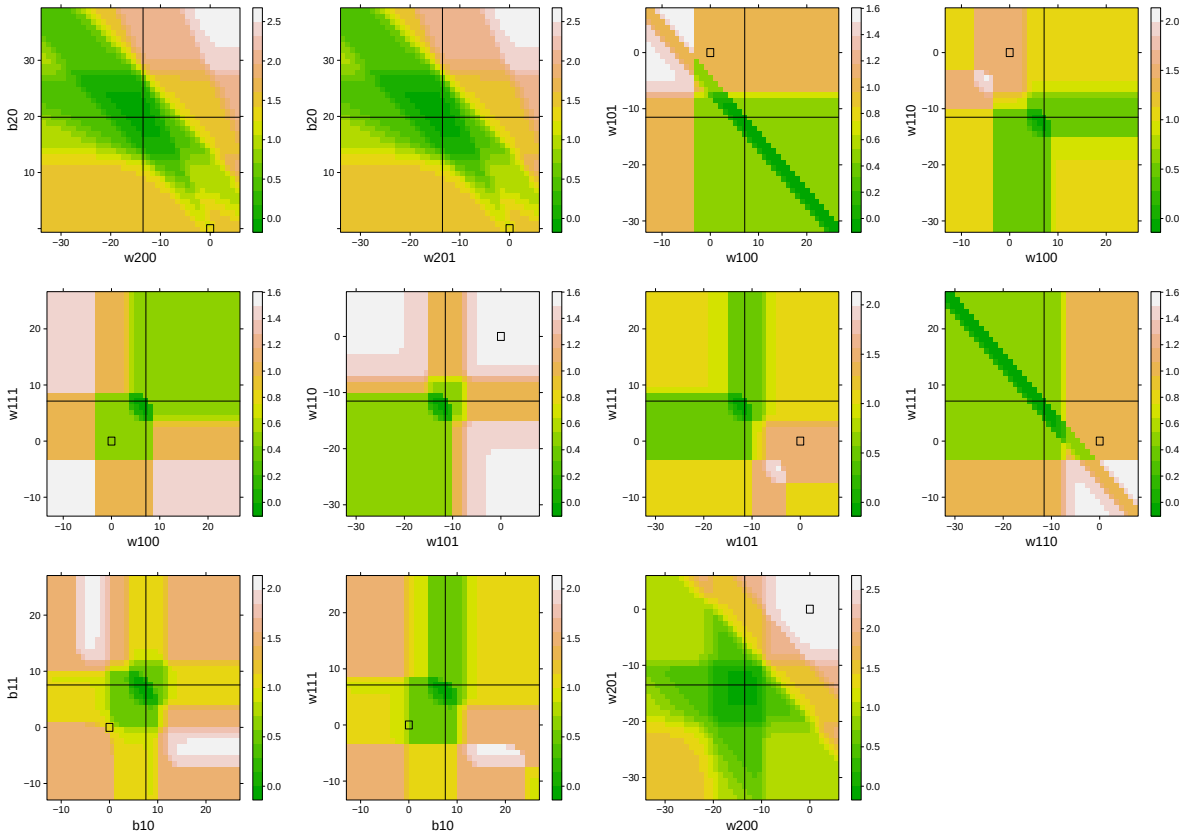


Figure 4.45: Error volume slices for the successful $x \oplus y \longrightarrow x \wedge y$ solution given by cluster 1, showing the error surface for two weights holding other parameters constant. Range large to ensure the random start range is included, which is marked by a square. the centre of the surface is the solution.

random parameter range of $[-5, 5]$. A larger range of starting values may provide more routes for the network to converge to a solution — usually smaller initial weights are better to avoid the weights becoming saturated at the start of the gradient descent, and Bishop's rule of using $[\frac{1}{\sqrt{n}}, \frac{-1}{\sqrt{n}}]$ (where n is the number of inputs to the node) has been used thus far. Under the [-5,5] regime, the pairing finds a solution in 19% of networks: a considerably better performance, and one which correlates better with the Monte Carlo experiments.

This new initial range was used to attempt to generate all possible boolean pairings, as was done for Fig. 4.4 in Sec. 4.3. The same hyperparameters were used (learning rate $\eta = 0.1$, 75000 iterations). The results are shown in Fig. 4.46 with the new results in red, and the old results (initial weights in $[\frac{1}{\sqrt{n}}, \frac{-1}{\sqrt{n}}]$) in blue. These results show that networks with a smaller initial value range generally do better, probably by avoiding saturation as described above, but for a very few pairings the higher initial value range converges to a solution more often. These pairings are $x \wedge y \longrightarrow \neg(x \vee y)$ and its counterpart $\neg(x \wedge y) \longrightarrow x \vee y$, and $x \vee y \longrightarrow \neg(x \vee y)$ and

its counterpart $\neg(x \vee y) \rightarrow x \vee y$. These are the four most difficult pairings to train using back-propagation in a UESMANN network (see Table 4.1), and it now seems likely that these are caught by local minima between the relatively small starting range and the solutions. Increasing the initial weight range avoids these minima in these pairings, but causes problems for other pairings.

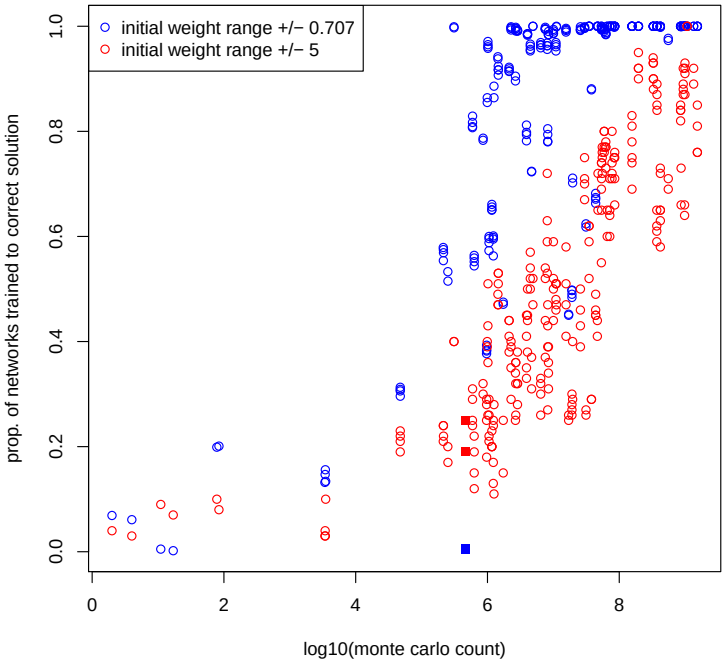


Figure 4.46: Proportion of successful convergences plotted against $\log_{10}$ of the Monte Carlo count for 2-2-1 UESMANN networks learning the boolean pairings at $\eta = 0.1$ with 75000 iterations. Two different initial weight ranges were used: the original Bishop's Rule weight of [-0.707,0.707] in blue, and an increased range of [-5,5] in red. The points marked with a solid square indicate the pairings $x \wedge y \rightarrow \neg(x \vee y)$ and $\neg(x \wedge y) \rightarrow x \vee y$. There are four of these, one for each function at both initial weight ranges. They all have a $\log_{10}$ count of approximately 5.8; the two blue squares are superimposed.

4.6 Summary of boolean UESMANN networks

Earlier (Sec. 3.1) we asked whether a UESMANN network can represent any pairing of boolean functions in the same dimensionality as an equivalent multilayer perceptron. The Monte Carlo simulations of Sec. 3.1.4 show that they can, although the solutions for some pairings are extremely small. A brief study was made in Sec. 3.2 of some other multiplicative global modulation methods with a single parameter,

with the conclusion that modulating the weights only (rather than only the biases, or weights and biases) provided the best distribution of solutions.

We also demonstrated in Secs. 3.1.1 and 3.1.2 that only a small subset of boolean pairings are representable by a single UESMANN node, and explained why this is so by deriving a set of inequalities which must be met for a solution to be possible. It was established that the fundamental operation of modulation in a UESMANN node is to halve the bias and narrow the unsaturated region of the activation function.

If we attempt to build networks with two hidden layers to perform pairings by combining nodes which perform boolean operations, there are several pairings which cannot be generated. These pairings have solutions, so must operate by using the hidden nodes' unsaturated outputs. They generally occur less frequently in the solution space, perhaps because the weights and biases require careful balancing in order to function correctly.

We noted above that solutions to some pairings are rare, covering small regions of the solution space. To find these solutions, the back-propagation of errors algorithm was modified to traverse the error gradient with respect to the modulated weight. No augmentations (such as momentum or weight decay) were used, in order to get a better understanding of the underlying behaviour. Tests showed that the algorithm works well on boolean problems, but that the error surface can be complex: increasing the learning rate beyond a certain point causes failure. Some pairings also require an increased range of initial random weights, suggesting that limiting the search start to a small area around the origin causes problems. This may be due to complexities in that part of the error space. However, raising the initial weight range causes problems for other pairings, possibly due to saturation in the nodes causing small gradients.

A brief comparison of the transition behaviour — how the network behaves as the modulator varies between 0 and 1 — was made with three other modulation methods (output blending, weight blending, and h-as-input). This was done for two pairings (see Figs. 4.32 and 4.43) and shows that

- UESMANN produces a fairly wide transition, weighted towards $h = 0$;
- output blending produces a symmetrical, wide transition which becomes a crisp transition on thresholding;
- weight blending produces an unpredictable transition due to competing conventions;
- h-as-input produces unpredictable transitions, which were wide in one pairing, narrow and inconsistent in another.

UESMANN's transitions were also consistent, in that the same pairing usually produced one type of transition. This is due to the limited number of solutions which are available: fewer than 10 for the pairings tested, with many of these being functionally identical. A brief study was made of the symmetries in the solution groups: pairings of commutative functions will have each solution represented by four areas in the solution space, while each solution in other pairings will be represented by two areas.

We will now study the behaviour of UESMANN in more complex classification problems: recognising line orientations and handwriting recognition. This will be done with particular reference to the transition region in the handwriting case. Output blending and h-as-input will be used as comparisons — weight blending can be discounted because of its competing conventions problem.

Part III

UESMANN in Classification

Chapter 5

Introduction and methodology

The previous chapter demonstrates that the UESMANN network architecture can perform well, learning any two binary boolean functions in networks of the same dimensionality as that required for learning a single function. In this chapter, we will explore UESMANN's performance on two classification problems of increasing complexity. It is possible that such networks may perform well in such problems, retaining the same interesting transition behaviour. While our eventual aim is to use the network as a controller in an adaptive system, it is useful to study classification problems before looking at regression and control problems.

Image classification problems are particularly useful, because in analysing the problem we can take advantage of the classification properties of the human visual cortex: we can look at the images ourselves and know how they should be classified, and compare them with the results of our network. Thus handwriting recognition is a good test because it is complex, yet easily understood because it is a problem we can easily perform ourselves. Control problems can be more difficult to understand, because it can be difficult to visualise what the outputs should be for a given set of inputs.

Image classification problems also are useful in analysing how a network's nodes actually calculate a complex function: we can generate images corresponding to the weights of the hidden nodes, and relate those to the input images as we do in Sec. 6.5.

For these reasons, and because one of the primary applications of neural networks is image classification, many of the benchmarks for neural network performance are based on classifying images. One notable example is the MNIST handwriting recognition database, which has become a *de facto* standard in the field [150, abstract] and which we use in Sec. 7. Indeed, much of the early work in neural networks is essentially image classification, such as the work of Rosenblatt [236] and Minsky and Papert [197], and the "T-C" problem in Rumelhart, Hinton and Williams [237].

UESMANN was tested on two such problems of increasing complexity: first, a binary classification problem involving recognising vertical or horizontal lines in images. This was chosen because it involves binary classification of simple shapes. As such, it should perform well with a low number of hidden nodes and have recognisable patterns of weights in the hidden layer, permitting analysis of the networks' function.

Following this, the well-known MNIST multi-class handwriting recognition problem [168] was used, with networks attempting to learn two distinct labellings of handwritten digits. This was chosen because it is a standard benchmark in classification.

For each of these problems we have the following questions:

- Can UESMANN learn to perform two distinct classifications in a single set of parameters? The previous work on boolean functions gives us confidence that it will perform fairly well.
- How well does UESMANN perform compared with output blending and h-as-input on these problems[1]? The convergence behaviour and the performances of the trained networks was compared: from the previous work, UESMANN's performance is likely to be comparable to h-as-input and worse than output blending in terms of both convergence times and trained performance.
- How does the behaviour of the three types of network change as the modulator h is varied between the extrema 0 and 1? UESMANN should have a wide transition region, morphing smoothly between the two behaviours. Output blending and h-as-input are likely to have narrower transition regions.

5.1 Methodology

To answer these questions, the following methodology was used for both problems (i.e. in both Chapters 6 and 7).

A number of networks were trained at different learning rates and hidden node counts. Given that the images are $28 \times 28 = 784$ pixels[2], the hidden node counts

[1]Weight blending was not tested: at the h extrema it is the same as output blending while its transition behaviour will be unpredictable due to competing conventions, as established by earlier work.

[2]This is the size of the images in the MNIST database; the line recognition images were designed to conform with these.

selected should be between 1 and that size. The values

$$3, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700$$

were selected, very loosely following a log curve: it is likely that adding nodes to small networks will have a greater effect than adding nodes to larger networks. For each of these network sizes, three learning rates were used to train ten networks to evaluate the convergence behaviour. The rates selected were 0.05, 0.2 and 1. Again, these are on a log curve. Ideally more networks would have been trained at each rate, but computing resources were limited. For each network, initial weights and biases were selected randomly using Bishop's rule, giving the range $[-\frac{1}{28}, \frac{1}{28}]$ for the hidden layer and $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$ for the output layer where n is the number of hidden nodes.

Each experiment used two sets of data, each consisting of a number of labelled images. One set, used for training, consisted of 60000 images. 10000 images from this set were held back for validation during training, leaving 50000 training images. The other set, used for testing of the final network after training, contained 10000 images. These are the sizes of the training and test sets of the MNIST database; the same amounts of data were generated for the line recognition experiments. Within the data each class has similar prevalence[3]. For binary classification, the output layer was thresholded at 0.5: greater or equal to 0.5 is positive, otherwise the image is not in the class. The training examples have 0 or 1 outputs accordingly. For the multiclass handwriting problem, the output encoding is "one-hot": there is one output for each class, and in the examples the output whose index corresponds to the label is 1 while the others are zero [204, p. 215]. When running the network, the highest output gives the predicted class.

Both experiments' data sets contain noise. The line recognition data has Gaussian noise in the background of the images (See Fig. 6.2, p. 170), while the MNIST data has no background noise, but is inherently noisy because it contains digits handwritten by humans.

Each network was trained for 150000 pair presentations, which is a mere three iterations through the training set of 50000 images. More iterations were not possible due to the limited computing resources. During training, the 10000 images held out from the training set were divided into 50 slices. At regular intervals, the network was tested against one of these small sets in order to evaluate convergence; this

[3]The lines data has three classes: horizontal, vertical and blank. The test data does not divide by three, so these classes are represented by 3333, 3333 and 3334 examples respectively. The variation in the MNIST data set is slightly higher, as will be discussed in Sec. 7.

avoided testing against the whole set, which was prohibitively expensive. The slices were rotated through during training. Further details of the training algorithm are given for each set of experiments.

Following training (during which accuracy was recorded using the validation slices) the best network found during the training was taken as the result. This is the network whose sum of squared errors at the outputs, summed for $h \in \{0, 1\}$ and over training examples, is a minimum. Note that this is not necessarily the network after the final training iteration — the error does not increase monotonically.

Having evaluated that the networks converge at all training rates, and having studied the convergence behaviour, the data for all but the lowest rate was discarded and the best networks taken for each attempt at $\eta = 0.05$. This network was tested using the test set at each h level $\{0, 1\}$, generating two confusion matrices for each h (2×2 for lines, 10×10 for handwritten digits).

We now wish to compare the performances of the networks. To compare a large number of networks, a single performance metric is required. The most common metrics are generated from 2×2 tables of confusion giving false positives, false negatives, true positives and true negatives. In binary classification problems like line recognition, these are identical to the confusion matrix. However, multiclass problems like handwriting recognition require the confusion matrix to be processed further to generate such metrics. We will discuss how this was done in Sec. 7.3, but the general approach requires generating one table of confusion for each class from the confusion matrix and either finding the element-wise mean of these tables and calculating the metrics, or generating the metrics for each table and finding the metric mean. This process is performed for the results at each h level, and the minimum found, to find the metric for the network at its worst performance.

Given that TP, FP are the number of true and false positives and TN, FN are the number of true and false negatives, available metrics include:

- Accuracy — the proportion of correct results, given by $(TP + TN)/(TP + TN + FP + FN)$
- Precision (or positive predictive value) — the proportion of positives which are true, given by $TP/(TP + FP)$
- True positive rate (sensitivity or recall) — the proportion of true results which are classified correctly, given by $TP/(TP + FN)$
- False positive rate (specificity) — the proportion of false results which are classified correctly, given by $FP/(TN + FP)$

- F_1 score — the harmonic mean of precision and sensitivity, given by

$$F_1 = \left(\frac{precision^{-1} + sensitivity^{-1}}{2} \right)^{-1}$$

- ϕ, the Matthews correlation coefficient, given by

$$\phi = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Accuracy is easily understood, but is known to have problems if the prevalences of the classes are different (the so-called "accuracy paradox" [285]). Here, the prevalences are almost equal for all classes in the lines data, but vary slightly for the handwriting recognition data. Moreover, accuracy can be slightly deceptive in that 0.5 is the "worst" score, since an accuracy of zero implies a perfectly perverse classifier (i.e. a classifier which misclassifies all items). The F_1 score suffers from similar prevalence problems to accuracy and does not fully deal with true negatives [223]. Also, calculating the F_1 score gives a division by zero if the recall (true positive rate) and precision (positive predictive value) are both zero, which occurs when no positives were detected (by very poor networks). This can be resolved by assigning a score of zero to such case: this is a poor result, since we know there are different classes in the data set [232] but should more correctly be considered undefined.

The Matthews correlation coefficient (ϕ) is essentially a binary classification analogue of the well-known Pearson's correlation coefficient, measuring the correlation between the true-false and positive-negative "axes". However, if the number of negatives or positives is zero, it also leads to division by zero. Again, this can be resolved by assigning a score of zero, i.e. "no correlation." In these tests, the Matthews correlation coefficient was used, despite the division by zero problem, because of its relationship with the existing Pearson's-ρ and the varying prevalences of different labels in the handwriting data set. For each network a value of ϕ was obtained for each modulator level $h \in \{0, 1\}$, we will combine these by finding the minimum, denoted ϕ_{min}.

In the lines problem ROC curves for each network were also plotted at each learning rate, although instead of using a classifier threshold to generate the points the number of hidden nodes was used. These show the sensitivity and specificity of the network type as the hidden nodes change. This was not practical in the handwriting recognition problem, as is discussed in Chapter 7.

With the line recognition experiments, it was possible to generate networks with sufficiently few hidden nodes to permit analysis of their function. This analysis demonstrates how each network type works with tasks with many inputs.

Finally, to analyse the transition behaviour, two different strategies were used. For the binary classification (line recognition) problem, the number of positives in each class recognised in the test set was plotted over a range of h values. For the handwriting recognition task, graphs were plotted showing the region of the h range for which one of the endpoint labellings is being correctly performed — in other regions, some intermediate labelling function is being performed. To provide more detail we will show examples of the labellings at the intermediate points for some of the networks.

In UESMANN, we would predict that both functions produce intermediate functions which are, in some sense, true intermediates between the end points. For example, we should be more likely to see "recognise no lines" and "recognise all lines" as intermediates in the line recognition task, and some form of smooth digit-by-digit labelling change in the MNIST task, rather than the "nonsensical" functions which might be produced by the weight blending network. These intermediate functions are likely to cover wider regions of the h range than in the corresponding output blending and h-as-input networks.

Chapter 6

UESMANN in line recognition

In these experiments, a large number of monochrome 28×28 pixel images were generated containing near-horizontal lines, near-vertical lines, and blanks, drawn in white (1) on a black (0) background. All images were overlaid with noise and a degree of blurring. Each image was labelled with 0 (horizontal), 1 (vertical) or 2 (blank). Some sample images are shown in Fig. 6.1. The endpoint functions should detect horizontal and vertical images at $h = 0$ and $h = 1$ respectively.

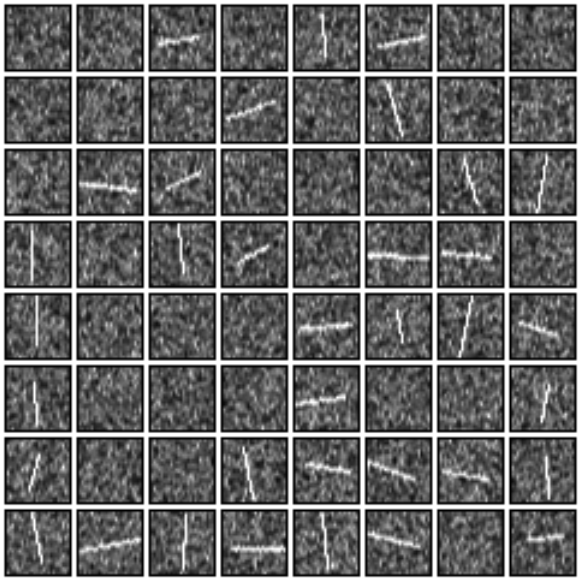


Figure 6.1: Example images for the line recognition experiments

6.1 Image generation

Images were generated using Algorithm 6. This uses Bresenham's line algorithm to draw the lines (without antialiasing), first generating Gaussian noise using a Box-

Muller transform [224, p. 364] with $\mu = 0.4$ and $\sigma = 0.15$. The resulting image is blurred by convolution with a 3×3 kernel, given in the algorithm. The kernel was chosen to both dilate and blur the image. The line end points were uniformly randomly selected from fixed ranges, which are shown in Fig. 6.2. These were chosen in an ad-hoc fashion to generate lines whose orientation and length clearly varies, but which can clearly be called "vertical" or "horizontal."[1]

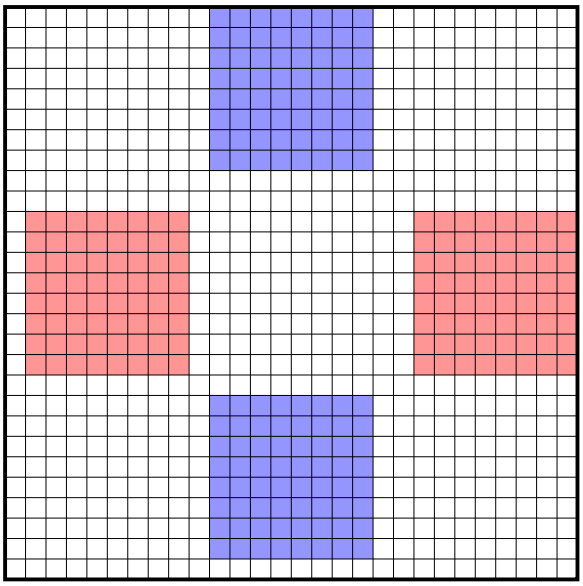


Figure 6.2: Line end-point zones for line generation. Horizontal lines are drawn between two points randomly selected from each red zone, vertical lines similarly use the blue zones.

6.2 Network training

The networks were all trained using Algorithm 7, which is essentially identical to Algorithm 1 with the addition of validation slices. The validation set was divided into 50 slices, and validation was performed on a slice every 150 pair presentations in order to evaluate convergence behaviour.

6.3 Convergence behaviour

First, we will examine how the different network types converge at all three chosen learning rates. This is done to establish that sufficient training iterations have been

[1]The bottom row and left column are always empty — this was an early programming error which there was insufficient time to amend.

Algorithm 6 Line data generation

$T \leftarrow$ image, 28×28 pixels of value 0
fill T with gaussianNoise($\mu = 0.4, \sigma = 0.15$)
if not a blank image **then**
  $x_0 \leftarrow U([1, 8])$
  $x_1 \leftarrow U([20, 27])$
  $y_0 \leftarrow U([10, 17])$
  $y_1 \leftarrow U([10, 17])$
  if line is to be vertical **then**
    $x_0 \leftrightarrow y_0$
    $x_1 \leftrightarrow y_1$
  end if
  draw line in T from (x_0, y_0) to (x_1, y_1)
end if
$I \leftarrow 0.25\left(T * \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}\right)$ *(i.e. convolve with* 3×3 *binomial kernel)*
output I and associated label (0=horizontal, 1=vertical, 2=blank)

provided, and to compare the numbers of required nodes to converge to a solution at different learning rates (after this, we will only consider $\eta = 0.05$).

6.3.1 Control convergence behaviour

The network type used here is output blending: to recapitulate, two separate $784 \times n \times 1$ networks are trained to recognise horizontal or vertical lines. The outputs of these two networks are linearly interpolated between using the parameter h. As such, this type also serves as a control: the performance at the end points $h \in \{0, 1\}$ is the performance of a single network trained using plain back-propagation to perform the horizontal or vertical line recognition task.

6.3.1.1 Convergence periodicity

As noted above, every 150 pair presentations the network was run on a slice of the validation set. The convergence data for $\eta = 1$ for the control with $h = 0$ is shown in Fig. 6.3 without smoothing, with all attempts shown (thus there are 10 points at each iteration for each hidden node count). This plot has a large amount of periodic noise: every sample is from a different slice of the validation set, with the slices repeating every 50 samples. Samples are taken every 150 pair presentations, leading to a periodicity of 7500 pair presentations, which is what the plot shows.

Algorithm 7 Learning two different classifiers for vertical/horizontal lines

$n_{validationExamples} = 10000$ {we hold out 10000 validation examples}
$n_{cv} = 150$ {number of iterations per validation}
$n_s = 50$ {number of validation slices}
$N \leftarrow$ new network of $784 \times n_h \times 1$ nodes {i.e. n_h hidden nodes}
E = full training example set, **indices start at 0**
$n_e = |E| - n_{validationExamples}$ {calculate number of training examples}
$T = \{E_i : 0 \leq i < n_e\}$ {get training set}
$V = \{E_i : n_e \leq i < |E|\}$ {get validation set}
$S_{0 \cdots n_s - 1} = n_s$ disjoint subsets of V of equal size {the validation set slices}
$i_{slice} = 0$ {the validation slice index}
for all weights i, j and biases b_i **do**
  $w_{ij} \leftarrow U(\frac{-1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$ where d is the number of weights in the node and $U(p, q)$ is a uniformly distributed random number in the range $[p, q]$: this is Bishop's rule.
  $b_i \leftarrow U(\frac{-1}{\sqrt{d}}, \frac{1}{\sqrt{d}}$ {d, U defined as above}
end for
$err_{min} = -1$ {initialise minimum error to rogue value}
for i=0 **to** 150000 **do**
  $(\mathbf{i}_e, l_e) \leftarrow E_{i \bmod |T|}$ {get the current example: $\mathbf{i}_e$ is input image for example e, l_e is the label}
  $h \leftarrow 0$ {set the modulator level of the network for training}
  present example $(\mathbf{i}_e, f_1(l_e))$ to the backprop algorithm (Algorithm 2 for UESMANN) and update the weights and biases accordingly; $f_1(l_e)$ is the output of function 1 on that label for that image, where

$$f_1(x) = \begin{cases} 1 & \text{if} \quad x = 0, \{0 \text{ if line is horizontal}\} \\ 0 & \text{otherwise} \end{cases}$$

  $err_1 \leftarrow (N(\mathbf{i}_e) - f_1(l_e))^2$ {get squared error of network in this example at $h = 0$}
  $h \leftarrow 1$
  present example $(\mathbf{i}_e, f_2(l_e))$ to the backprop algorithm and update:

$$f_2(x) = \begin{cases} 1 & \text{if} \quad x = 1, \{1 \text{ if line is vertical}\} \\ 0 & \text{otherwise} \end{cases}$$

  $err_2 \leftarrow (N(\mathbf{i}_e) - f_2(l_e))^2$ {get squared error of network in this example at $h = 1$}
  $err = err_1 + err_2$ {calculate total error}
  if $err < err_{min} \vee err_{min} < 0$ **then** {If the error is the smallest found, store this best network}
    $err_{min} = err$
    $N_{best} = N$
  end if
  if $i \bmod n_{cv} = 0$ **then** {n_{cv} is number of iterations per convergence measurement}
    run all examples in $S_{i_{slice}}$ through the network with both $h = 0$ and $h = 1$ and record the number of runs for which either function gave an incorrect classification.
    $i_{slice} = (i_{slice} + 1) \bmod n_s$ {rotate through validation slices}
  end if
end for

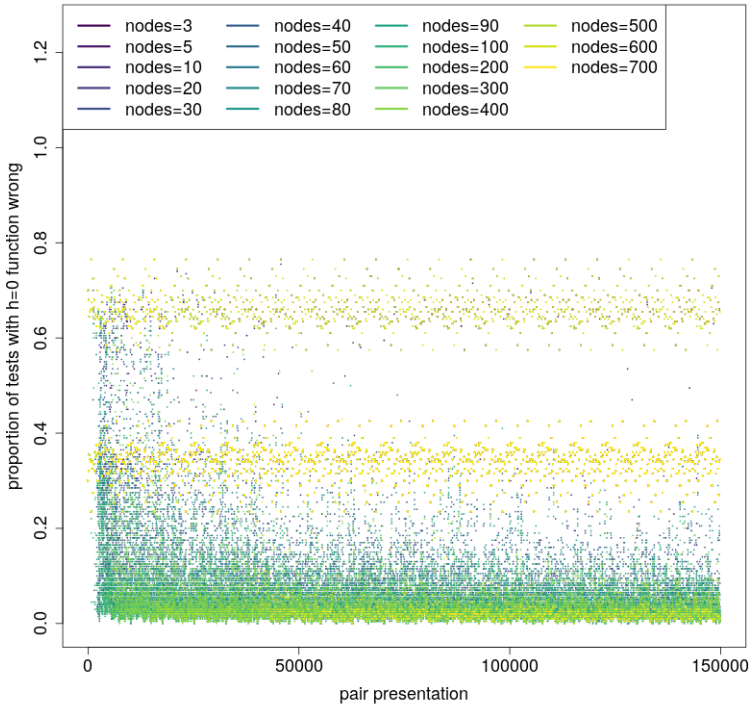
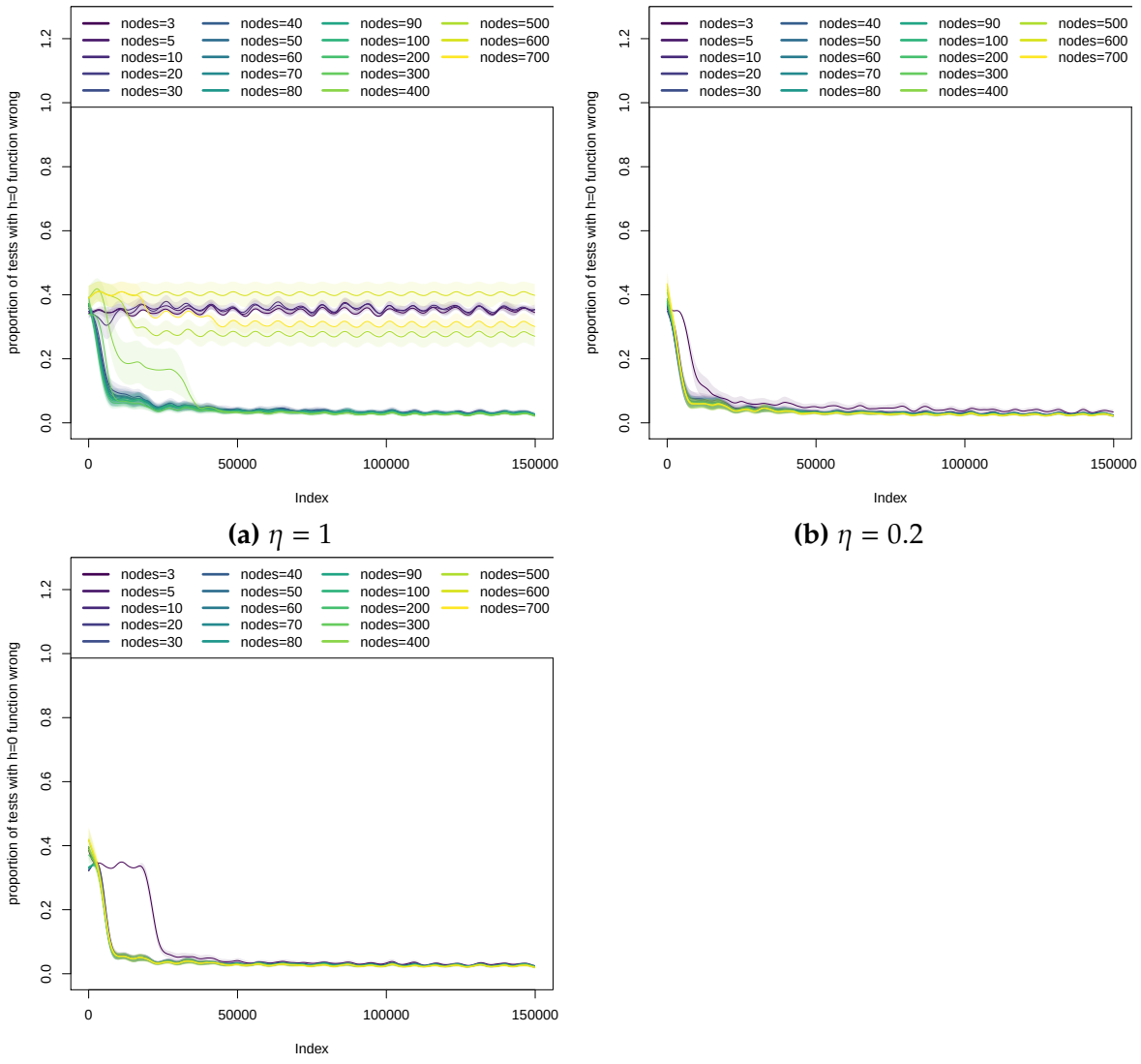


Figure 6.3: Unsmoothed convergence data for $\eta = 1$ in the control for identifying horizontal lines (class 0), showing all attempts, and thus 10 points at sample point (every 50 pair presentations).

6.3.1.2 Smoothed results

This raw data is hard to read and there is an apparent high error rate at around 0.65: this is an illusion due to many outliers of similarly coloured nodes. To aid legibility, the means of errors for each of the hidden node counts was taken and smoothed using a polynomial spline (using R's `pspline` package). Separate splines were generated for the mean±0.25 standard deviations (this small fraction was chosen for legibility). This results in the plot in Fig. 6.4a, shown with the plots for all learning rates, Fig. 6.4. This shows networks with less than 20 hidden nodes failing, and higher node counts being more successful with an error rate of less than 0.05. This trend continues until the node count rises to around 400 (this is the curve which takes some time to converge to a solution). After this, performance is poor, possibly due to overfitting. Higher learning rates converge to a solution much more quickly as expected, with both low and high node counts also converging to solutions. The accuracy of these solutions is about the same as for $\eta = 1$, indicating that all learning rates probably find the same solutions.

This is fairly good performance for plain back-propagation with no enhancements on a very simple task. We will now analyse the convergence behaviour of the modulatory network types.



(a) $\eta = 1$

(b) $\eta = 0.2$

(c) $\eta = 0.05$

Figure 6.4: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ in the control at $h = 0$ for identifying horizontal lines, showing the means of all attempts for a given node count, with shading delimiting the region within 0.25 standard deviations of the mean.

6.3.2 Output blending convergence behaviour

In output blending, we are training two networks and switching between them (in these tests $h \in \{0, 1\}$) only accepting as "correct" tests those for which both networks produce the right answer. The control described above tests only one of these

networks. For example, a test image with a horizontal line must generate positive from the $h = 0$ network and negative from the $h = 1$ network in the same pair, while the control tests only the $h = 0$ network.

However, the two networks are trained completely independently. If we treat the error probability of the networks as two independent random variables E_0 and E_1, the probability of either network being in error will be $E_0 + E_1 - E_0E_1$ (by the basic rules of probability). If we assume that $E_0 \approx E_1$, since both line detection tasks should be of roughly the same difficulty, then this becomes $2E_0 - E_0^2$. The convergence of output blending should therefore have this relationship to the convergence of the $h = 0$ plain back-propagation control.

Using the same technique as Fig. 6.4 to plot the results for output blending, this time using as our metric the number of examples in the convergence slices for which both functions produced the correct classification (i.e. was able to identify horizontal lines at $h = 0$ and vertical lines at $h = 1$), we obtain the convergence behaviour in Fig. 6.6. This shows roughly the relationship with the control which we describe above, but with some variation: for example, the model predicts that in the limit as the control error approaches zero, the output blending error should approach half the control error. This does not hold, therefore one of our assumptions is wrong.

The incorrect assumption here is that the two functions are converging at an equal rate. If we plot (without smoothing, for this demonstration) the convergence of the two different functions (in output blending, the two separate networks) for $\eta = 1$ with 100 hidden nodes, we obtain Fig. 6.5. This shows that the $h = 1$ function has a higher error rate that $h = 0$. If the functions are reversed (so that horizontal lines are recognised at $h = 0$ rather than vertical lines), we see the opposite pattern: this is not an artefact of the training system (nor can it be — the two networks are independent here). It appears that vertical lines are harder to recognise that horizontal lines, but the reason for this is unknown: they should be equally difficult, given that one is simply a rotation of the other. This is likely to be due to a problem with the generation of the training data, which will be investigated as part of future work.

It should be borne in mind when looking at later figures that this error rate is that for the validation slices and has been smoothed: accuracy figures for the final network will be calculated on the entire test set, and will not agree. Note that the validation error drops but then rises towards the end of training for some node counts: this is likely due to the high learning rate causing the network to overshoot the solution. Fig. 6.6 shows the convergence plots for all selected learning rates, showing that at lower learning rates all nodes converge to a solution. At both $\eta = 0.2$ and $\eta = 0.05$ the best smoothed mean performance is around 0.05, although the

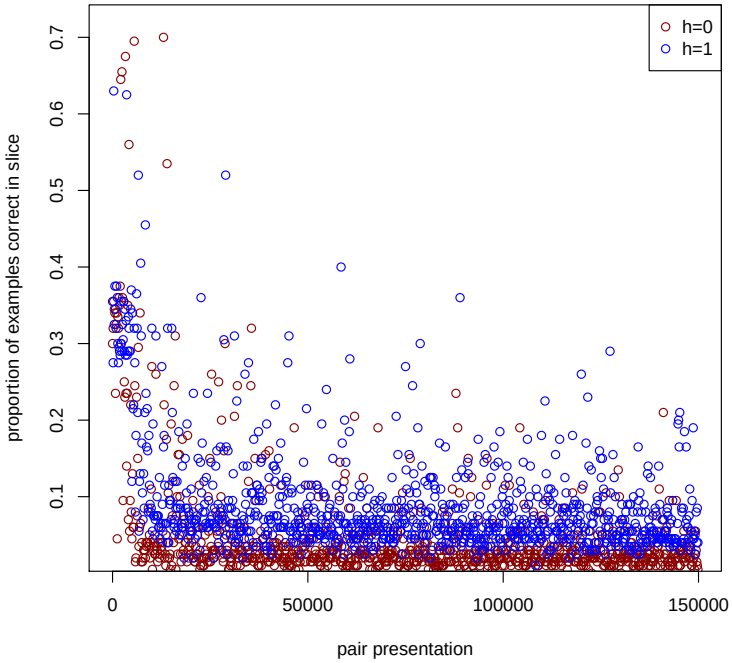


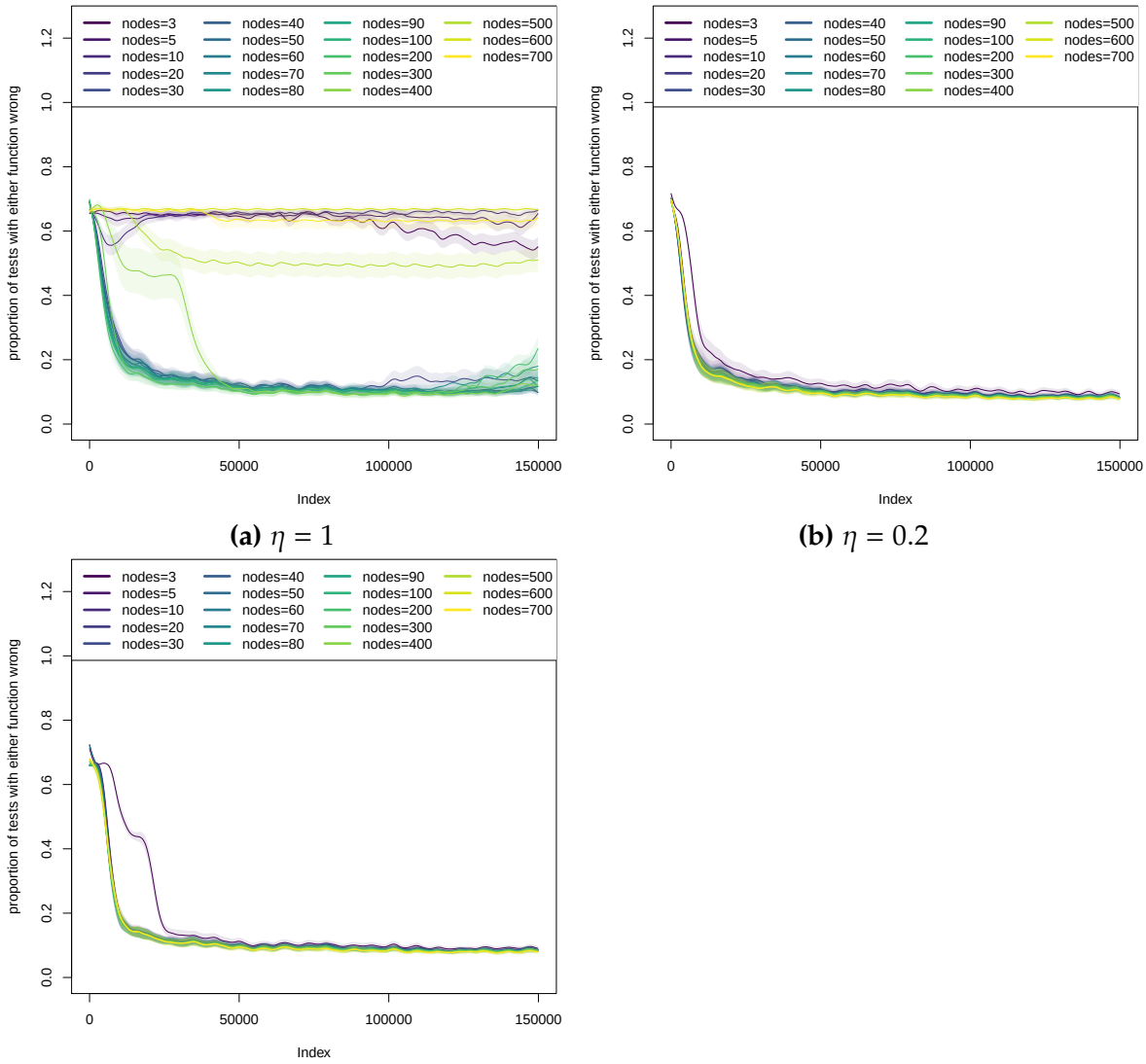
Figure 6.5: Convergence (unsmoothed) of output blending for line recognition at $\eta = 1$ with 100 hidden nodes, showing the two functions separately.

nodes are still converging very slowly when the training completes. Unfortunately the available resources did not permit a longer training time.

6.3.3 h-as-input convergence behaviour

The convergence behaviour for h-as-input networks of the same topologies as the previous experiments at the three different learning rates is shown in Fig. 6.7.

At the highest learning rate ($\eta = 1$) no nodes converge to a solution. At $\eta = 0.2$, all but the lowest node counts ($n \leq 40$) do so, with the rest appearing to overshoot the solution minima. At $\eta = 0.05$ all node counts converge to a mean error rate of around 0.09. This is poorer than output blending, as is to be expected given that this is a single network rather than one for each function. Convergence also takes longer than in output blending, with networks converging to a solution at around 100000 pair presentations (at $\eta = 0.05$), and only starting to converge after 25000 presentations in what appears to be a flat region of the error surface. In output blending, the error rate starts to fall immediately as the training begins: the error surface is likely to be much simpler in a network learning a single function.



(a) $\eta = 1$

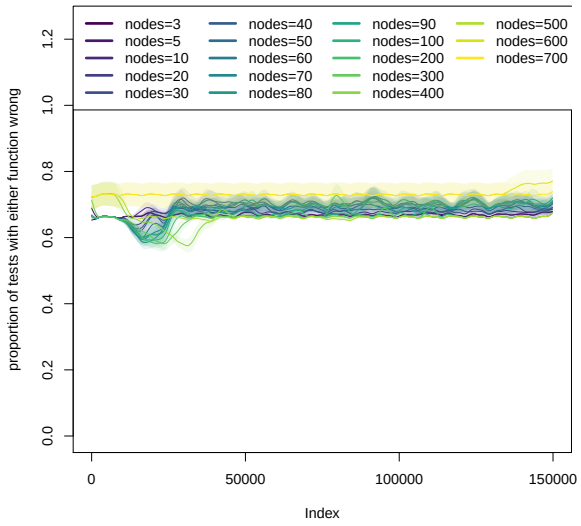
(b) $\eta = 0.2$

(c) $\eta = 0.05$

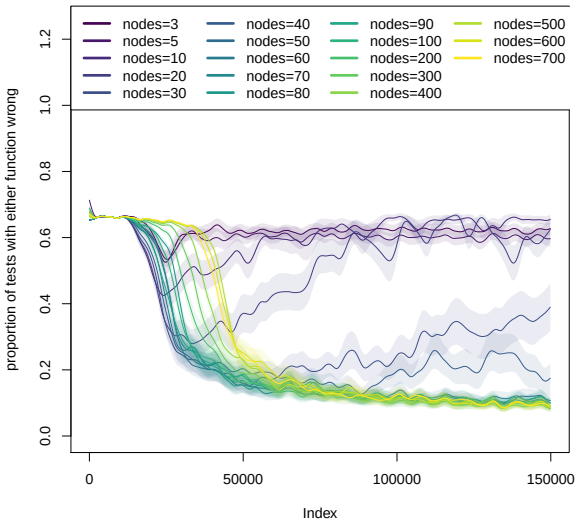
Figure 6.6: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ for output blending, showing the means of all attempts for a given node count, with shading delimiting the region within 0.25 standard deviations of the mean.

6.3.4 UESMANN convergence behaviour

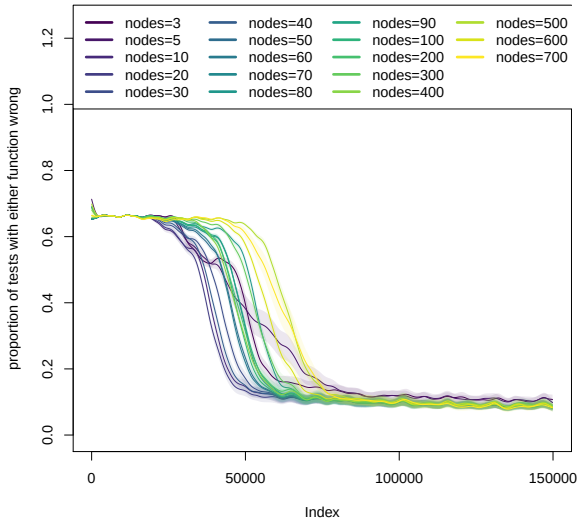
The behaviour of UESMANN networks in the same set of $784 \times n \times 1$ topologies at different learning rates is shown in Fig. 6.8. Across all learning rates the flat region seen in the h-as-input results is not present, and $\eta = 1$ performs slightly better (but still poorly) in this regime. At $\eta = 0.2$, performance is worse, however — this appears to show a strong local minimum in which this network becomes trapped. At the lowest learning rate, low node counts ($n \leq 40$) converge to poor solutions, while greater node counts converge to a mean error rate of around 0.09, comparable with h-as-input (although the latter network type trains better at lower node counts at



(a) $\eta = 1$



(b) $\eta = 0.2$

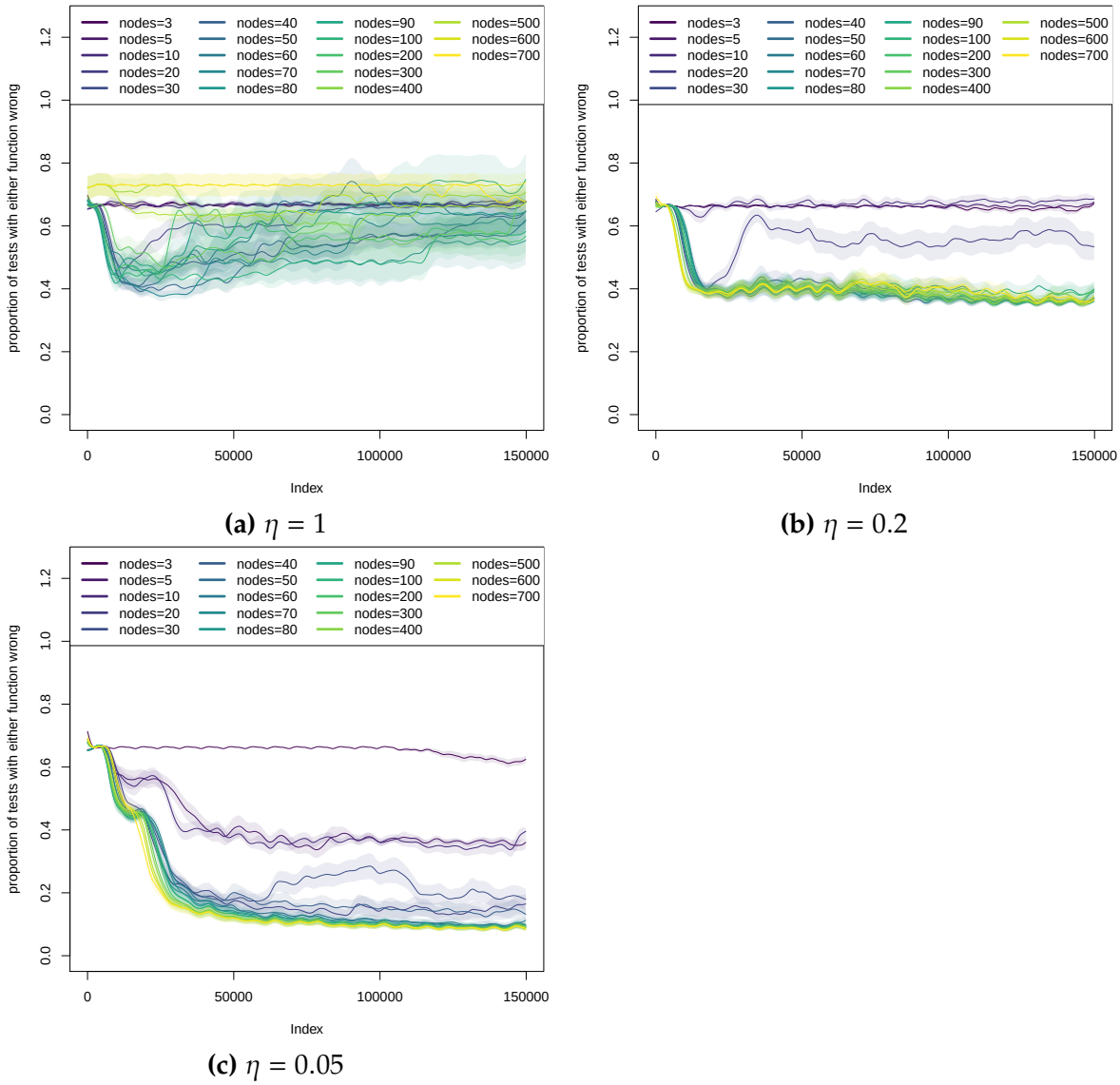


(c) $\eta = 0.05$

Figure 6.7: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ for h-as-input, showing the means of all attempts for a given node count, with shading delimiting the region within 0.25 standard deviations of the mean.

this learning rate). This lower learning rate permits the system to escape the local minimum which affects $\eta = 0.2$; this can be seen in the "kink" in the plots at around 25000 iterations. A similar flat region is present in the h-as-input convergence plots of the previous section, which may indicate similarities in the error surface or at least the sizes and curvatures of the minima.

UESMANN appears able to learn a solution to recognising two different line orientations in a single network, although it requires more hidden nodes than h-as-input to do so reliably. At higher node counts (greater than 50) UESMANN and h-as-input have roughly the same performance (according to validation).



(a) $\eta = 1$

(b) $\eta = 0.2$

(c) $\eta = 0.05$

Figure 6.8: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ for UESMANN, showing the means of all attempts for a given node count, with shading delimiting the region within 0.25 standard deviations of the mean.

6.4 Performance of the different networks at $\eta = 0.05$

We have established that all three networks are able to learn how to recognise horizontal and vertical lines at $\eta = 0.05$. We will now examine the performance of the networks at this learning rate in more detail.

The performance of a binary classifier is often shown as a Receiver Operating Characteristic, or ROC, curve. This plots the true positive rate against the false positive rate, resulting in a curve in which a "good" classifier is close to the top left-hand corner (high TPR, low FPR). In a traditional ROC curve, the threshold of a single classifier is varied to generate the curve. Here, we instead show multiple

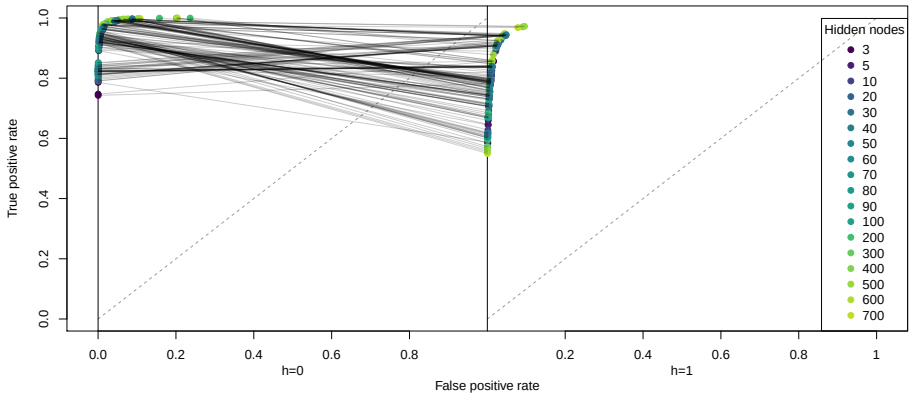
classifiers (the networks), changing the hidden node count to create the curve. The performances of all networks at each hidden node count are plotted.

Plotting an ROC curve is complicated by the fact that we have two different classification problems in a single classifier. Plotting two curves is misleading, because networks which perform well in one problem may fail in another. An alternative would be to plot each $h \in \{0, 1\}$ pair as a single point, taking the minimum of the true positive rate and the maximum of the false positive rate to show the worse performance. However, this would fail to show which of the two functions performed poorly.

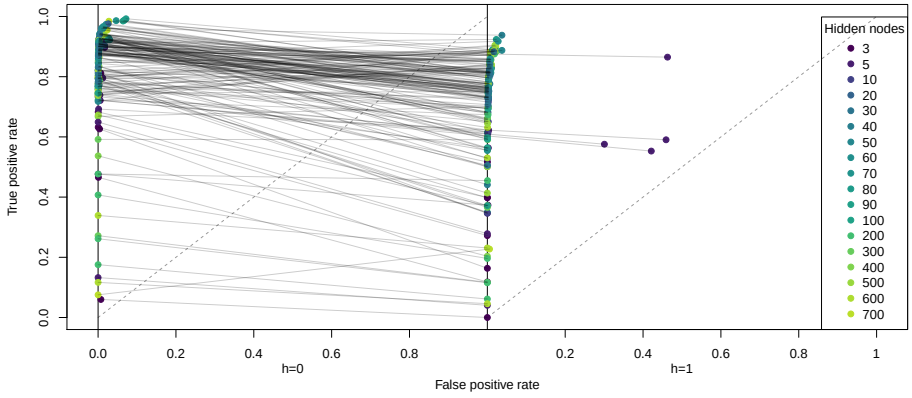
Our solution is to plot both curves side-by-side, and to link the results for the same network with lines. A network which performs both functions well will be in the top left-hand corner in both plots. The results for all three network types at $\eta = 0.05$ is shown in Fig. 6.9.

We see that output blending has the most consistently good results, with most networks achieving performances close to the optimum (true positive rate of 1 against false positive rate of 0) in $h = 0$ and slightly worse at $h = 1$ (showing a lower true positive rate). There appears to be a negative correlation between the performances: if $h = 0$ performs well, $h = 1$ will not perform as well and *vice versa*. However, this is illusory: the two networks are completely independent and the strongest correlation (Pearson's $r = 0.765$ at 50 hidden nodes) is not significant at $p < 0.05$.

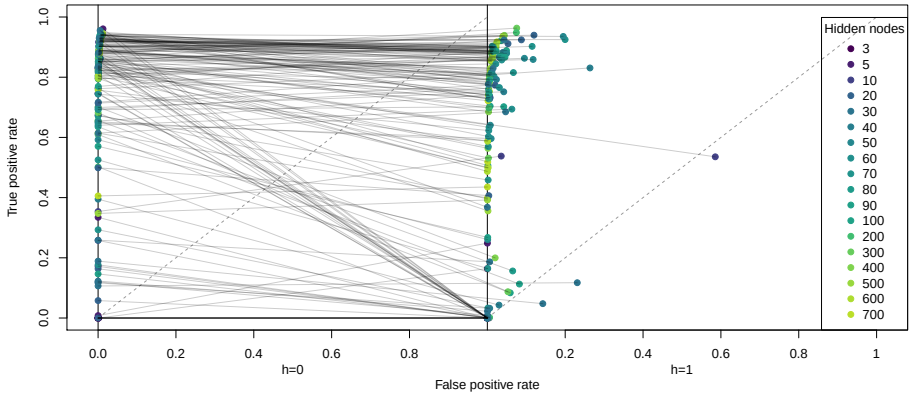
The h-as-input networks mostly perform well (we will see more details later) but with a large number of outliers giving a low true positive rate, and a few with low node counts giving a high false positive rate at $h = 1$. Again, the $h = 1$ performance is somewhat poorer.



(a) output blending



(b) h-as-input



(c) UESMANN

Figure 6.9: ROC curve (true positive rate against false positive rate) for all trained networks for each network type at $\eta = 0.05$. The left hand shows $h = 0$, the right hand side shows $h = 1$. Each network's pair of points are linked with a grey line. The dotted line indicates the performance of a random classifier: below this, the classifier is "perverse".

UESMANN has the poorest results overall, working with a very simple modulatory scheme in a single set of weights. Low node counts in particular perform extremely poorly, particularly at $h = 1$: the group of networks which fail to find any true positives at $h = 1$ are all networks with 10 hidden nodes or fewer. The solid line along the bottom is formed by all but one of the 3 hidden node networks: these networks fail to find any positives. Three other classifiers at low node counts are perverse at $h = 1$: they predict the wrong class more often than the correct class. However, individual networks with intermediate counts can perform well (see below).

The $h = 1$ (vertical line) examples appear to be slightly more difficult to learn in all network types. If this were true only in UESMANN we would have reason to suspect the algorithm, but this is not the case. It is particularly interesting that this also appears in output blending, where the two networks are completely independent. However, time did not permit further investigation — it may simply be that the examples provided in the data sets were skewed in some way.

The plots in Fig. 6.9 show general trends for the networks, and show that the performance at $h = 0$ and $h = 1$ does not necessarily correlate. However, it is hard to see how many individual networks at each node count are performant. We will therefore combine the plots using a single metric: the Matthews correlation coefficient described above. This produces two values, one for each network at $h \in \{0, 1\}$, which we shall further combine by taking the minimum over h: we are interested in networks which perform well on both functions. The results are shown in Fig. 6.10 as a multiple box plot.

UESMANN networks generally converge to reasonable solutions when they contain between 80 and 500 hidden nodes. The h-as-input networks require between 20 and 100 nodes, after which local minima appear. Output blending is performant over the entire range, because each network of which it is composed is trained to a single function.

While UESMANN generally performs poorly at fewer than 50 hidden nodes, at higher node counts networks are generated which perform remarkably well. Even at the modest count of 30 nodes, the best UESMANN network generated outperforms the other network types (admittedly, this is from a sample of only 10 of each, and the worst UESMANN network at this hidden node count is perverse). Indeed, the best UESMANN network outperforms the best h-as-input and output blending networks at many node counts (30, 50, 60, 70, 90, 100, 200, 300).

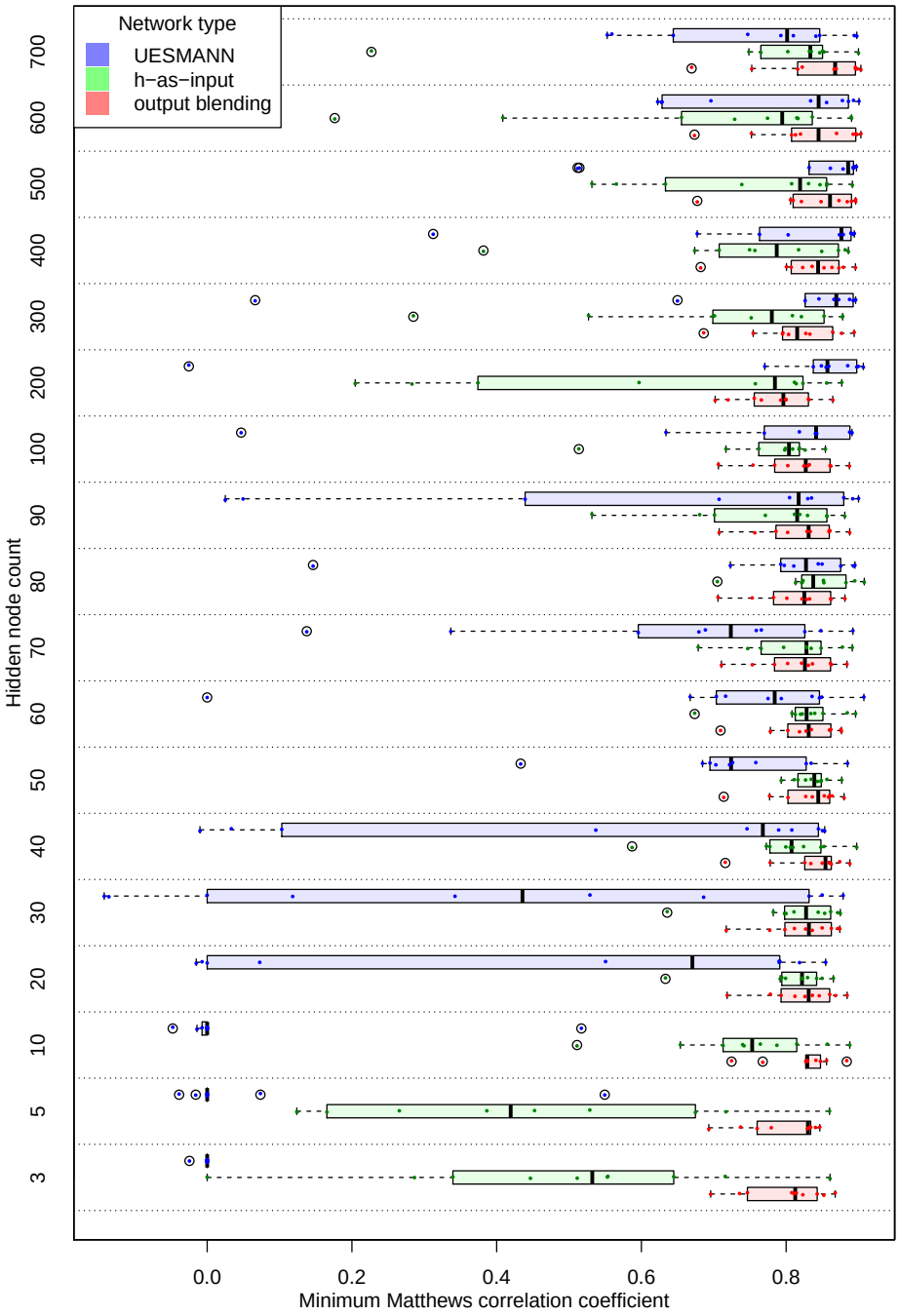


Figure 6.10: Box plot of minimum Matthews correlation coefficient ϕ_{min} for all networks of all three network types, across both $h = 0$ and $h = 1$. The actual network points are shown as red dots.

Performant networks are achievable at fairly low hidden node counts, but are rare: work for the next section showed that 100 repeats of the $\eta = 0.05$ experiment for UESMANN networks with only 3 hidden nodes generated some networks which achieved a ϕ_{min} greater than 0.9. Their rarity is shown in the histogram in Fig. 6.11. Nevertheless, they may exist for a given problem and can be found provided enough

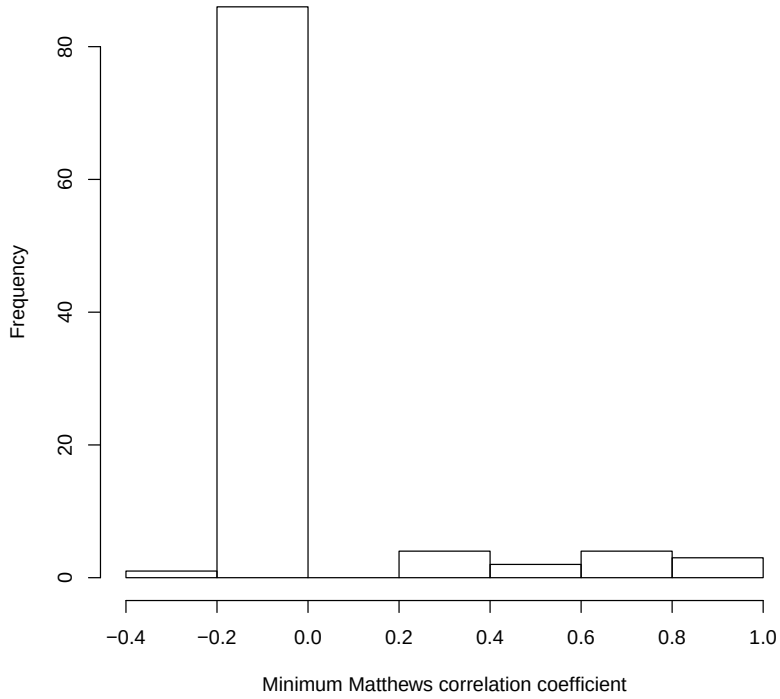


Figure 6.11: Histogram showing the frequency of minimum F_1 score values in 100 networks generated at $\eta = 0.05$ in UESMANN networks performing line recognition with 3 hidden nodes.

restarts are used. It might be possible to use early stopping to help with this: if the convergence fails to fall for a suitably long period, restart. However, this might fail to find solutions. Consider the convergence plot in Fig. 6.8c: the network navigates two plateaux which might cause an early stopping algorithm to restart too early. The same applies to other algorithms, of course: Fig. 6.7c shows multiple h-as-input networks starting in a very large plateau which takes nearly a full iteration through the data set to escape.

6.5 Analysis of network function

Because successful networks can be quite small in this problem, it may be possible to analyse the network behaviour for typical solution networks. This can be done by visualising the weights into each hidden node as a grid of pixels in a Hinton diagram [122]. This analysis should be tractable in networks with few hidden nodes.

Although good solutions are available from the previous experiments for h-as-input and output blending, Fig. 6.10 shows that training only ten networks generates no good solution networks for UESMANN at very low node counts. To see if such networks exist, UESMANN network training was repeated for 100 different initial weight sets at $\eta = 0.05$ with 3 nodes and the same number of training iterations. Several good quality networks were found as described in the previous section.

6.5.1 Output blending

This is the most straightforward case. Effectively, we are analysing a different network at each extremum of the modulator. Hinton diagrams showing the weights of both networks are shown in Fig. 6.12. These networks are easy to understand from the diagram. For the purposes of this discussion, we will number the nodes in the hidden layer from left to right as 1, 2 and 3.

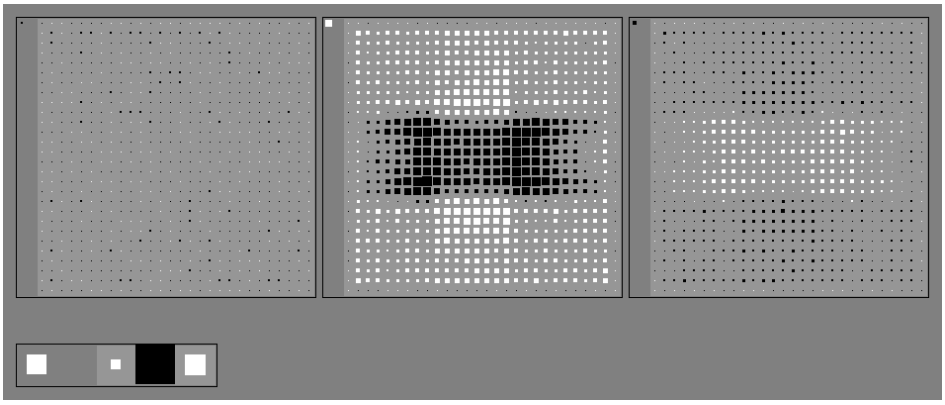
In the $h = 0$ network, the weights of node 2 show a pattern of weights which is inhibited by horizontal lines and excited by vertical lines. The weight for this node in the output layer is strongly negative, so the end result (combined with the inverse pattern and corresponding output layer weight of node 3) is to produce high values when horizontal lines are presented, and low values when vertical lines are presented. The bias on the output node may counteract the negative effect of the noise from the middle node.

Note that there is a large amount of redundancy here: a single node (effectively a Rosenblatt perceptron) could give a similar functionality and performance using a similar pattern of weights to that of the hidden node.

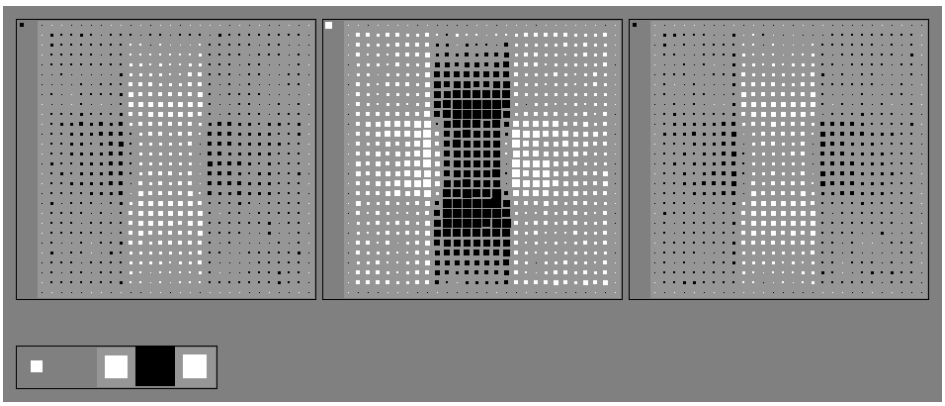
The $h = 1$ network, which recognises vertical lines, functions in a similar way. Here, node 2 is inhibited by vertical lines and excited by horizontal ones (and noise). In nodes 1 and 3 the pattern is reversed. Node 2 has a negative weight into the output layer while nodes 1 and 3 have positive weights. This results in excitation by vertical lines and inhibition by horizontal lines and noise.

6.5.2 h-as-input

The Hinton weight diagrams for this network are shown in Fig. 6.13. Again there is redundancy — hidden node 3 has very small input weights and a very small weight into the output layer and so can be discounted. Nodes 1 and 2 both have small weights, but in the distinctive patterns noted in the output blending networks for detecting line orientations. Here, the output is pulled down by lines of the right orientation, but pulled up by noise, more so in node 2. This is slightly unclear in the



(a) $h = 0$



(b) $h = 1$

Figure 6.12: Hinton diagrams of the weights in the hidden and output layer of both components of an output blending network with a good solution for line detection ($h = 0$ detects horizontal, $h = 1$ detects vertical). The three large figures show the input weights into each node in the hidden layer, with each square representing the weight from the corresponding pixel in the input. The bias is shown at the top left. The colour of the square is white if positive, black if negative, and the area of the square is proportional to the magnitude of the weight. The output layer is shown at the bottom: the three squares to the right show the weights from each hidden node, and the bias is shown to the left. All biases and weights within a node are normalised to the maximum absolute bias/weight within that node.

figures because the modulator weight magnitudes are so large. This large modulator input weight is positive in node 1, which is inhibited by vertical lines; and negative in node 2, which is inhibited by horizontal lines. Both these nodes have strongly negative weights into the output node.

Disregarding the modulator, node 1 will produce a negative output when a vertical line is presented and node 2 will produce a negative output when a horizontal line is presented. If the modulator is high, node 1 will be pulled high and node 2 will be pulled low. There will be no effect if the modulator is low.

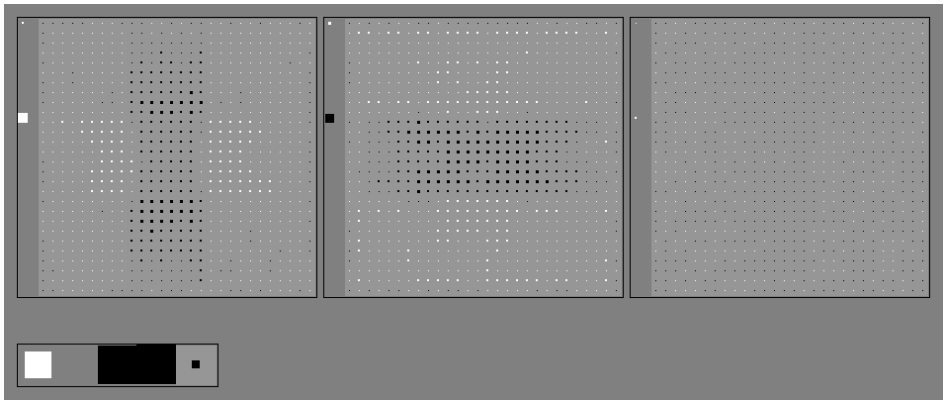


Figure 6.13: Hinton diagrams of the weights of an the h-as-input solution to line detection. The diagram is as described in Fig. 6.13, with the addition of the weight for the modulator input into the hidden layer. This is shown as an extra square halfway down the left-hand side of each node, below the bias.

If the modulator is low, horizontal lines are detected because node 2 will be pulled low, with that being negated by the negative weight into the output layer. While there is a small band of horizontal detection in node 1, it is counteracted by the large vertical detector overlaying it.

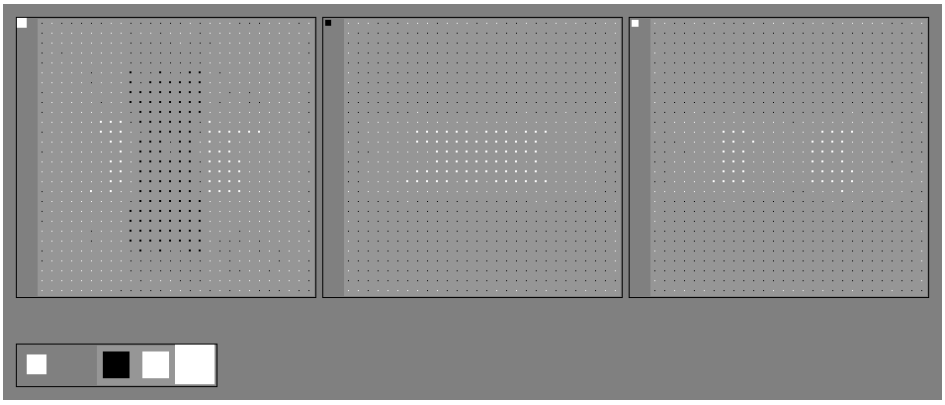
If the modulator is high, the detector in node 2 is inhibited by the large negative modulator input weight. Horizontal lines will pull node 2 lower, but to no effect. However, the vertical detector's output is now higher because of the modulator, but any vertical line present will pull down this node's output so that it no longer inhibits the output node, and the output node's bias will drive it high. Vertical lines are not detected with the modulator low because of the inhibitory effect of node 2's many positive input weights combined with the noise in the image.

It is likely that the learned model at these low node counts is overfitted to the data, relying on fairly precise amounts of noise in the background and lines which fall precisely within the areas defined by Algorithm 6.

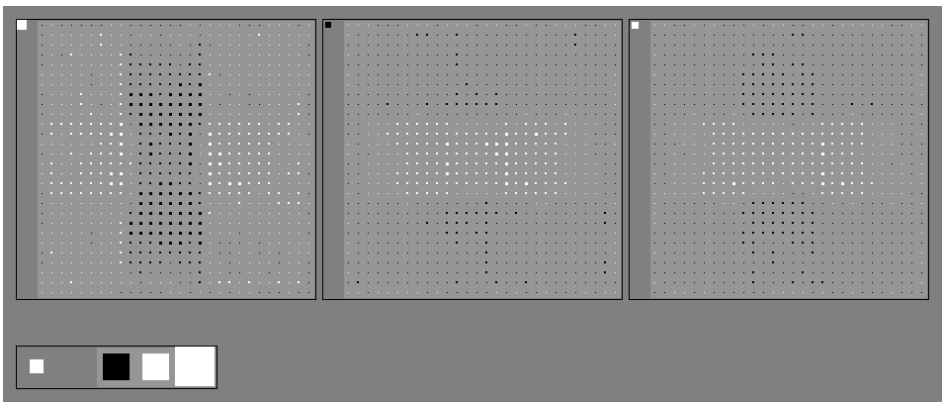
6.5.3 UESMANN

An example of a good 3 node UESMANN network (the best of the 100 found in the repeat experiments discussed above) is given in Fig. 6.14, shown as two networks (weights nominal and weights doubled). The network has a ϕ_{min} of 0.89. In this network, nodes 2 and 3 detect horizontal lines and excite the output node. Node 1 detects vertical lines, but using negative weights which inhibit the output when such a line is detected.

Under modulation, the output threshold is decreased. This will change the balance between the vertical detector of node 1 and the two horizontal detectors



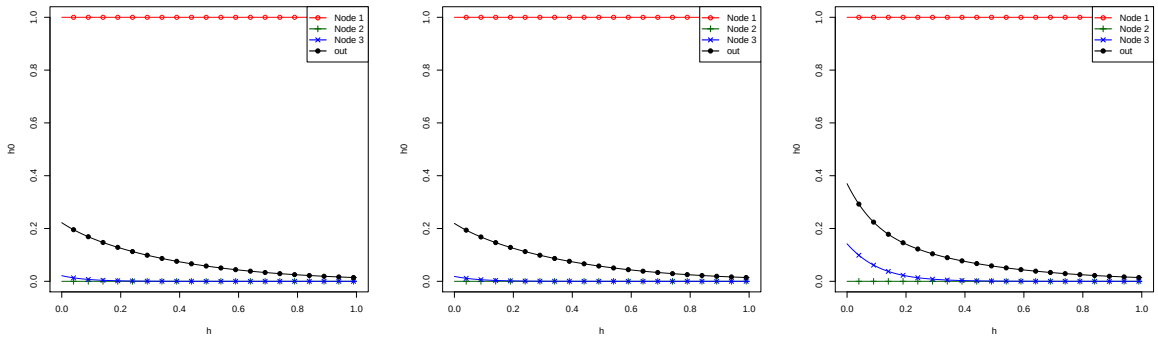
(a) $h = 0$



(b) $h = 1$

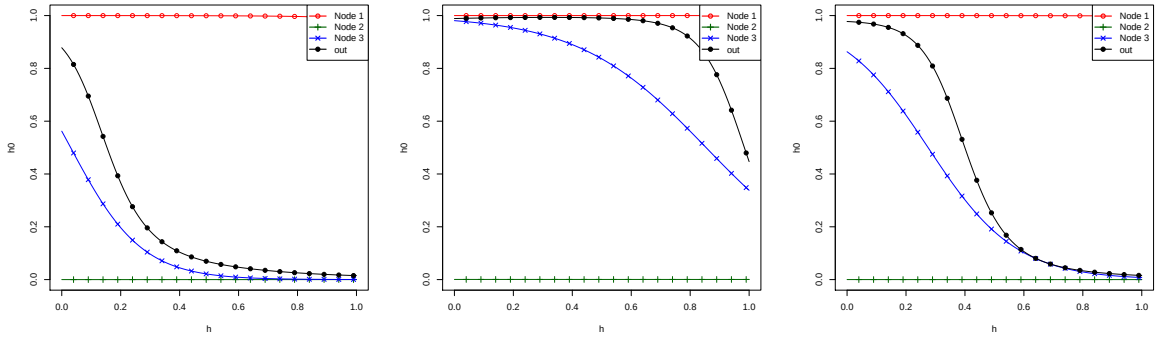
Figure 6.14: Hinton diagrams of the weights in the hidden and output layer of a UESMANN network with a good solution for line detection ($h = 0$ detects horizontal, $h = 1$ detects vertical). The diagrams are as described in Fig. 6.12, with one showing the network with nominal weight values at $h = 0$, and the other showing the weights doubled (but not the biases) at $h = 1$.

of nodes 1 and 2, causing a change in behaviour. There are also many non-zero weights which are connected to input pixels outside the line areas — UESMANN may be using these weights (which modulate) against the biases (which do not). To understand this better, plots were made of the hidden layer outputs and network output as the modulator moves from 0 to 1, for examples of the three different image types. These are shown in Fig. 6.15 for "blank" images (i.e. just noise), Fig. 6.16 for horizontal line images, and Fig. 6.17 for vertical line images.



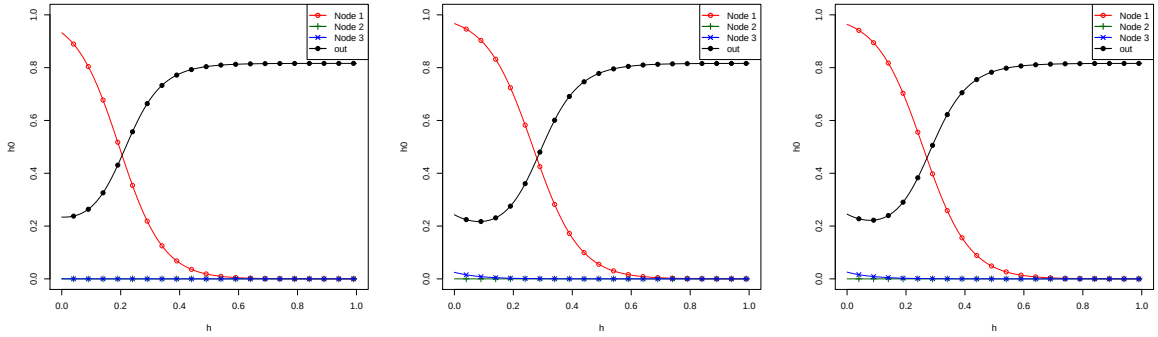
(a) Example 0 **(b)** Example 1 **(c)** Example 3

Figure 6.15: Plots of outputs of hidden layer and overall output for a good 3-node UESMANN solution to line recognition against h for three different blank images



(a) Example 14 **(b)** Example 20 **(c)** Example 21

Figure 6.16: Plots of outputs of hidden layer and overall output for a good 3-node UESMANN solution to line recognition against h for three different horizontal line images



(a) Example 2 **(b)** Example 4 **(c)** Example 5

Figure 6.17: Plots of outputs of hidden layer and overall output for a good 3-node UESMANN solution to line recognition against h for three different vertical line images

Several features are immediately apparent when these plots are read in conjunction with Fig. 6.14:

- Node 2 appears to be redundant — it never changes under either modulation or image type, always remaining low due to its negative bias, but is otherwise similar to node 3.
- Node 1 goes low when vertical lines are detected and the modulator is high, and has a negative weight into the output.
- Node 3 goes high when horizontal lines are detected and the modulator is low, and has a positive weight into the output.

With these basic ideas in mind, we will now look in detail at how the network behaves at the two modulator extrema.

6.5.3.1 Modulator low

- In a "blank" image, the white and black pixels in each hidden node are balanced, but there are more negative weights in every hidden node. All nodes but node 1 (with a large positive bias) are pulled down: node 2 by its negative bias and node 3 by the preponderance of negative weights being triggered by noise. Node 1 has a large negative weight into the output and the other two nodes have positive weights, so the output is low.
- In a horizontal line image, node 1 is high due to its bias and the positive horizontal contribution despite the negative vertical pattern, node 2 is still low because of its bias, but node 3 goes high: although it has an overall negative weighted input sum because of the negative weights, the horizontal line gives the positive weights a slightly larger contribution, increased further by the positive bias. Node 3's activation now drives the output high, since the output has a low positive bias.
- In a vertical line image, node 1 is still high because of its bias. The other nodes will be pulled down even more by the vertical line, so the output will be low (remembering node 1's negative weight into the output). This is very sensitive: if there is too large a vertical line, node 1 will be driven low and detect the vertical line erroneously at $h = 0$.

6.5.3.2 Modulator high

Here, as we have seen, the high modulator doubles the weights (or halves the bias) which changes the balance of the nodes — in particular, there are more negative

weights than positive in the weights of each hidden node, leading to a general decrease in activation relative to the bias.

- In a blank image, node 1's bias is still high enough despite the increased weights to drive it high. Nodes 2 and 3 are driven even further down because of the larger negative weights. The result is a low output.
- In a horizontal line image, node 1 remains high because of the positive weights on the horizontal line. Node 3 now remains low: the increase in weights means that the overall negative sum of the weighted inputs is larger, and the positive bias is no longer sufficient for the node to go high. Node 2 remains low, having a negative bias. With node 1 high and both nodes 2 and 3 low, the output is low and the horizontal line is not detected.
- In a vertical line image, node 1 now goes low: the increased contribution of the larger negative weights combined with the vertical line weights is now sufficient to overcome the positive bias. Nodes 2 and 3 remain low, being pulled down by the increased negative weights (some of which are coming from the vertical line). With all three hidden nodes low, the positive bias on the output is now able to pull the output high.

Thus the UESMANN solution is a complex balancing act in the biases and weights of both hidden and output nodes. As noted above, node 2 is indeed redundant. Its output never contributes to the function of the output node, because it is pulled low by the combination of its negative bias and negative weights. Therefore a 2 node solution exists: to obtain it we could simply remove node 2. It is not known how readily back-propagation would find such a solution; testing this has been left for future work.

6.5.3.3 The possibility of a UESMANN output node shifting between perceptrons

The above solution requires that both hidden and output nodes change their behaviour under modulation. Is it possible to create a single UESMANN node as an output layer in an otherwise conventional network which shifts between discrimination using one perceptron to another? In the lines problem, this would involve two "normal" unmodulated perceptron networks: one trained to recognise horizontal lines, the other trained for vertical lines. The outputs of these would feed into a UESMANN node which switches between them as the modulator varies.

We have already effectively proven that such a node is impossible. Referring back to the work on boolean functions in Sec. 3.1, we have shown that the pairing

$x \to y$ and all similar pairings involving a function of a single input to a function of the other input (such as $y \to x$ or $x \to \neg y$) are not possible in a single UESMANN node (see Fig. 3.5 on page 75). Since our putative node is effectively performing such a function — the output being a function which relies on only one of the inputs, different for $h = 0$ and $h = 1$ — such a node cannot exist.

6.5.4 Network function on solid colour images in networks with few nodes

All the images provided in both test and training sets have a noisy background of intermediate brightness. It is likely that successful solutions (particularly UESMANN and h-as-input, which have considerably fewer weights than output blending) will rely on this background. A simple way to test this is to present the networks with the highest Matthews correlation coefficient with images which are solid black (0) and solid white (1), at $h \in \{0, 1\}$. This should always give a negative result, since there are no lines in the images. This test was performed on the highest scoring 3-node networks of each type at $\eta = 0.05$, and the results are as follows:

- Output blending correctly gives negative results for both images at both h values.
- h-as-input detects no line in a black image at both h values, but detects a line at $h = 1$ in the white image.
- UESMANN detects a line at both $h = 0$ and $h = 1$ in the black image, and detects a line at $h = 1$ in the white image.

We can see that both h-as-input and UESMANN have devised solutions which rely on a grey background, while output blending appears to have avoided this. Indeed, we can see from Sec. 6.5.3 that UESMANN relies on the negative weights of background pixels to keep excitation low. It is also possible that h-as-input and UESMANN have overfitted solutions: they may be unable to detect lines which are outside the parameters determined by the line generating algorithm. This is as yet untested.

6.6 Transition behaviour

We will now study the transition behaviour of the 10 networks generated in Sec. 6.3 at $\eta = 0.05$. For each network, the modulator was varied between $h = 0$ and $h = 1$ and all 10000 examples in the test set were used. The number of positives for each

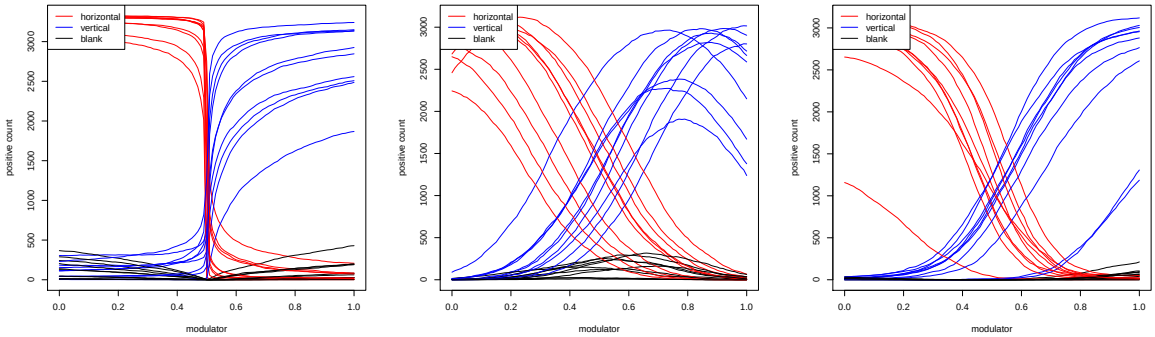
label (horizontal, vertical and blank) was measured. Thus at $h = 0$, a perfect network would show 3333 positive horizontal detections, no vertical and no blanks; while at $h = 1$ such a network would show 3333 positive vertical, no horizontal and no blanks. The three network types were run at three different hidden node counts to examine how the node count affects the transition behaviour. The results are shown in Fig. 6.18.

As expected from the results of previous boolean experiments, output blending produces good horizontal detection until $h = 0.5$, and then switches fairly abruptly to vertical detection, because the outputs of two detectors are being interpolated between and thresholded. The actual output of the network, while not shown, will decrease in magnitude until 0.5, and then increase again. Positive detection decreases around 0.5: at this point, marginal examples may not raise the network output over threshold, given that they are blended with a low result from the other network.

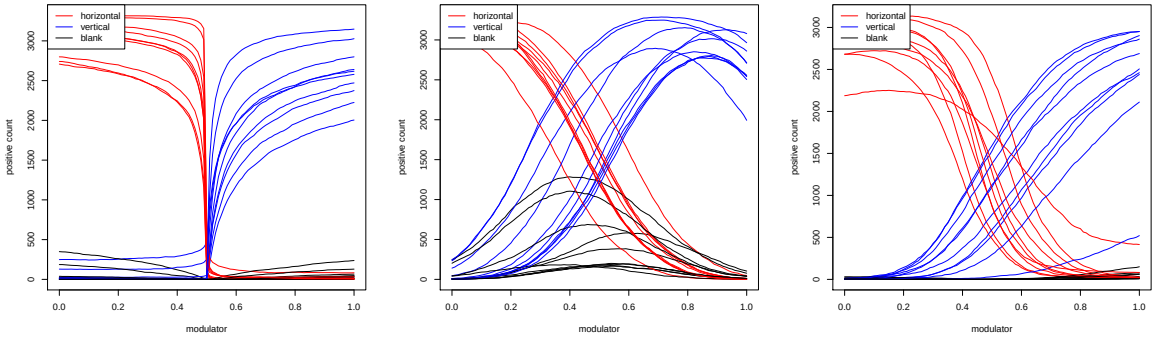
Both h-as-input and UESMANN produce smoother transitions, moving smoothly between horizontal and vertical line detection. Interestingly, the h-as-input networks appear to detect more lines when the modulator is at around 0.2 or 0.8 — the reason for this is not understood, but may be because detecting some of the other line type helps in some way. Perhaps a marginal horizontal line may be detected with the aid of an element of the vertical line detection. Another feature is that the crossover point – where vertical detections outnumber horizontal detections – is not symmetrical around 0.5, except at the lowest node count. In the best performing network (80 hidden nodes) it is closer to to 0.4.

UESMANN itself produces its best performance at $h = 0$ and $h = 1$ (ignoring the poor outliers). Unlike the h-as-input networks, the mean crossover point appears close to $h = 0.5$ although there is still asymmetry: the number of vertical detections rises more slowly than the number of horizontal detections falls. This effect increases as the number of hidden nodes decreases, and is likely to be caused the complex "balancing act" UESMANN uses to construct the transition as described in Sec. 6.5.3. This contrasts with h-as-input, where the asymmetry is in the crossover point, but not in the nature of the transition itself — UESMANN's transition is around 0.5, but the shape is different.

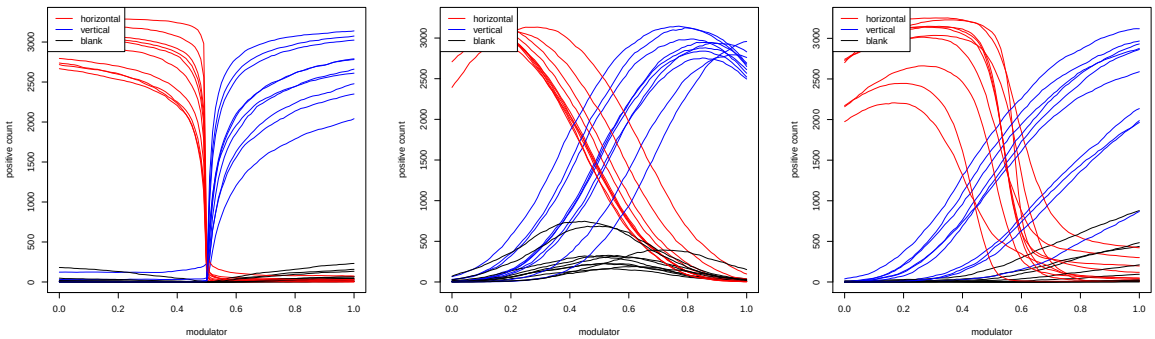
Fig. 6.19 shows the transition behaviour of the few UESMANN networks which performed well at 3 hidden nodes (of the 100 attempts at this low node count, performed separately as described above). This plot shows a degree of asymmetry not seen at higher node counts, and shows a large amount of variation in the network behaviour: notably in network 17, the second-best network, whose vertical line



(a) Output blending, 500 hidden nodes **(b)** h-as-input, 500 hidden nodes **(c)** UESMANN, 500 hidden nodes



(d) Output blending, 80 hidden nodes **(e)** h-as-input, 80 hidden nodes **(f)** UESMANN, 80 hidden nodes



(g) Output blending, 50 hidden nodes **(h)** h-as-input, 50 hidden nodes **(i)** UESMANN, 50 hidden nodes

Figure 6.18: Transition behaviour of the three network types at 50, 80 and 500 hidden nodes, $\eta = 0.05$. Each curve is the behaviour of one of the 10 networks generated previously. The plot shows the number of positives of each class labelled in the test set which produce an output greater than 0.5 from the network.

detection activates rapidly at $h = 0.6$. It is likely that this network uses a rather different solution from that described above.

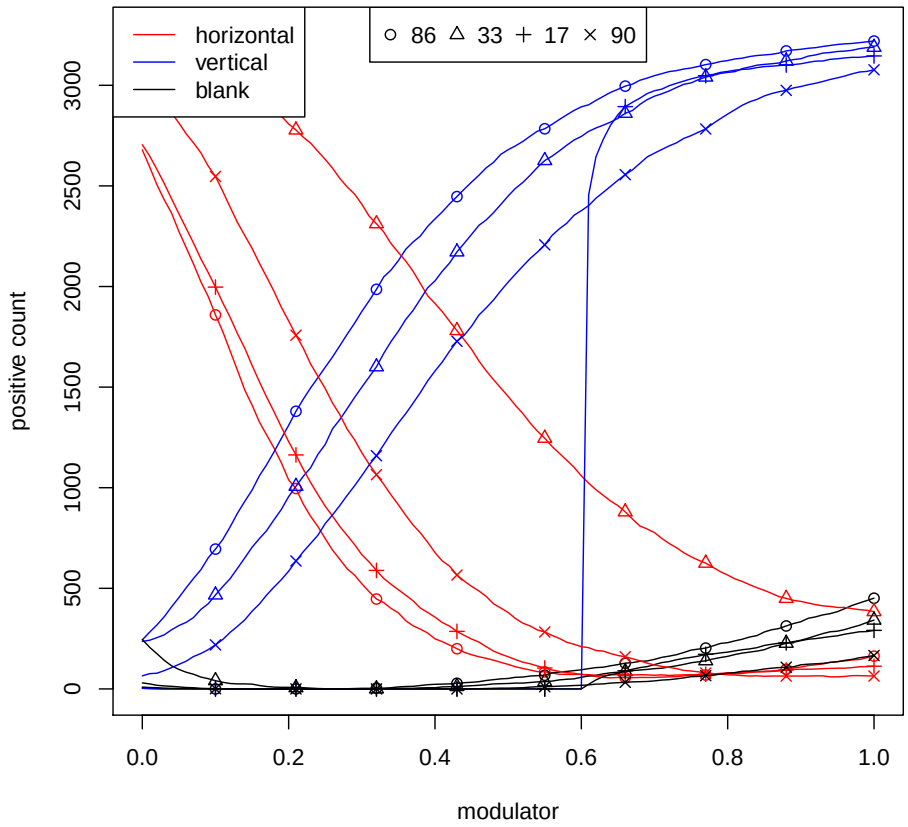


Figure 6.19: Transition behaviour of UESMANN networks with 3 hidden nodes, $\eta = 0.05$. The plot shows the number of positives of each class labelled in the test set which produce an output greater than 0.5 from the network. The lines are marked with points according to their network index in the 100 attempts, as shown in the legend above the plot, where they are ordered by performance (90 has the highest minimum Matthews correlation coefficient).

6.7 Summary

We have established that UESMANN can be trained to perform a simple discrimination task with three hidden nodes, morphing between vertical and horizontal line detection. Analysis of that solution shows that the task can also be performed with two hidden nodes, although it remains to be shown whether the network can be trained to find such a solution. At low node counts, many restarts of the training algorithm may be required before a local minimum is avoided. Such solutions require

a careful balancing of weights and biases across both the hidden and output layer, and cannot be achieved by simply modifying the output weights of a single-function network (see Sec. 6.5.3.3).

At intermediate hidden node counts solutions are readily found, and may outperform output blending and h-as-input networks. However, the operation of UESMANN requires learning a careful balance between the weights and biases in the network to perform the desired transition under modulation. This can make UESMANN prone to overfitting, exploiting "background" information in the training data which may not be present in other data to generate this balance.

UESMANN's transition behaviour (as in the boolean experiments) is smooth, showing a gradual shift from horizontal line to vertical line detection, with a decrease in sensitivity around 0.5. It appears to be less prone to false positives than h-as-input or output blending. At intermediate node counts, UESMANN also transitions more symmetrically than h-as-input (output blending is always symmetrical, but very sharp in transition).

The performance of UESMANN may be improved by the use of the enhancements discussed in Sec. 4.1.6: these include adaptive learning rates and momentum. These would also improve results for the other modulatory methods.

Chapter 7

UESMANN in handwriting recognition

Handwriting recognition is an important and difficult problem, and, with the advent of large databases of examples has become a valuable test case for machine learning algorithms. The MNIST database is freely available, and has become a *de facto* standard among such databases. It consists of 70000 handwritten digits, divided into a training set of 60000 and a test set of 10000 examples. Each example consists of a 28×28 8-bit monochrome image of a digit, size normalised and centred, with its associated label (the numerical value of the digit) [168]. The first 64 images in the training set are shown in Figure 7.1.

Figure 7.1: The first 64 images of the MNIST database

In this set the prevalences of the classes are not equal, as shown in Table 7.1. However the variation is small. As described in the previous section, this is part

of our rationale for using the Matthew's correlation coefficient as opposed to the simpler accuracy. The variation is sufficiently small that it should only have a small effect on the training.

Table 7.1: Prevalences of the different digits in the two parts of the MNIST database, by proportion of the total.

Digit	Training set	Test set
0	0.099	0.098
1	0.112	0.114
2	0.099	0.103
3	0.102	0.101
4	0.097	0.098
5	0.090	0.089
6	0.099	0.096
7	0.104	0.103
8	0.098	0.097
9	0.099	0.101
Standard deviation	0.054	0.059

7.1 Training

In these experiments, the networks were trained using an analogue of Algorithm 7. Instead of using a single output, "one-hot encoding" [204, p. 215] was used. In this scheme, a single categorical variable is replaced with a vector of boolean elements, one for each category. Each element is true if and only if the original variable has the corresponding categorical value. In our case, we are using numeric outputs as boolean values: true is represented by 1, and false by 0. Thus we have the encoded values in Table 7.2, and our network has 10 outputs.

The generating functions for the outputs were therefore

$$f_1(\mathbf{i}, n) = \begin{cases} 1 & \text{if} \quad l = n \\ 0 & \text{otherwise} \end{cases} \tag{7.1}$$

and

$$f_2(\mathbf{i}, n) = \begin{cases} 1 & \text{if} \quad alt(l) = n \\ 0 & \text{otherwise}, \end{cases} \tag{7.2}$$

where $\mathbf{i}$ is the image, n is the output index, l is the label (an integer in the range [0,9]) and $alt(l)$ specifies an alternate labelling to learn at $h = 1$. We have specified this

Table 7.2: One-hot encoding for numeric digits, showing the corresponding encoding as a vector of values **x** for each digit.

Digit	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

labelling as switching adjacent digits: that is,

$$alt(l) = (1, 0, 3, 2, 5, 4, 7, 6, 9, 8)_l, \quad l \in \{0, 1, 2 \cdots 9\}, \tag{7.3}$$

where the notation $(a \cdots b)_i$ indicates the selection of an element of a vector $(a \cdots b)$ with zero-based index i. This provides a maximal Hamming distance between the two encodings: no output has the same value for the same digit across the two encodings. In other respects the algorithm is identical, and uses the same number of training and test examples and validation slices.

7.2 Convergence behaviour

As before, we will first examine the convergence behaviour at all three learning rates and for all hidden node counts. The values chosen are the same as those in the line recognition experiments.

7.2.1 Control convergence behaviour

Again, we are actually looking at the $h = 0$ accuracy for the output blending experiment, which is effectively a single network trained to recognise the nominal MNIST labelling. The accuracy on the validation slices during training is shown in Fig. 7.2, smoothed using splines as in the line recognition experiments. At $\eta = 1$, all but the very highest and lowest hidden node counts converge to solutions. The lowest counts perhaps have too few hidden nodes to represent a solution, while the highest are unable to find the small minima given the high learning rate. At the two lower

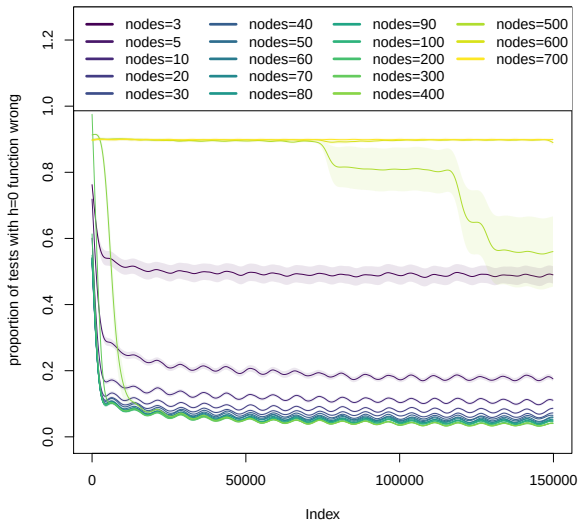
learning rates these high counts learn good solutions. However, from both these plots it is clear that training has stopped while the solution networks have not yet entirely converged. Unfortunately there were insufficient resources to complete the experiment to convergence: judging from the curve, an order of magnitude more iterations might be required to ensure this. However, the solution networks are clearly close to a solution.

It is noteworthy that the oscillatory behaviour due to the cyclic nature of the validation is stronger here than in the line experiments: there is more variation in how a given network performs on different parts of the validation set. This is likely to be due to increased variability within the slice — some examples are simply harder to classify than others.

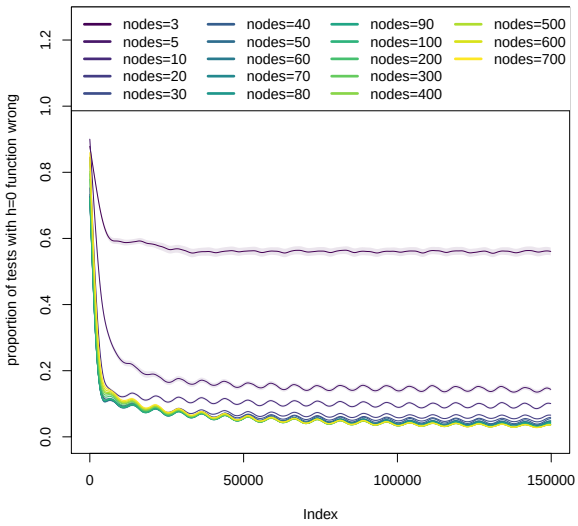
7.2.2 Output blending convergence behaviour

Fig. 7.3 shows the convergence of the output blending networks. If the difficulty of the two functions is the same, we should see the relationship described in Sec. 6.3.2: if the error for the plain back-propagation control is E_c, then the output blending error is $E_{ob} = 2E_c - E_c^2$. This holds in some cases as we would expect, because the two functions are simply different permutations of the output. Where it does not, this is probably because the variation is wide, there are too few samples, and these samples do not follow a normal distribution, making the mean misleading.

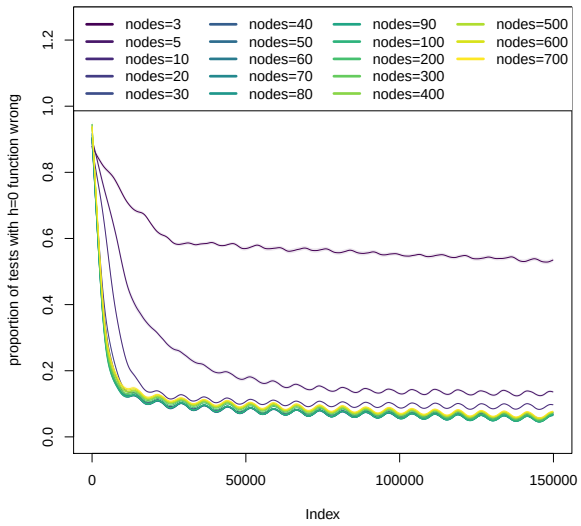
The convergence plots show that a hidden node count above 30 will converge to a solution at the two lower learning rates, but that more nodes are needed at lower rates. At $\eta = 1$, higher node counts fail to converge. Initial convergence toward solutions is rapid, taking place within 50000 pair presentations, although the networks continue to converge slowly throughout the run. Unfortunately there were insufficient computing resources to run the training further.



(a) $\eta = 1$

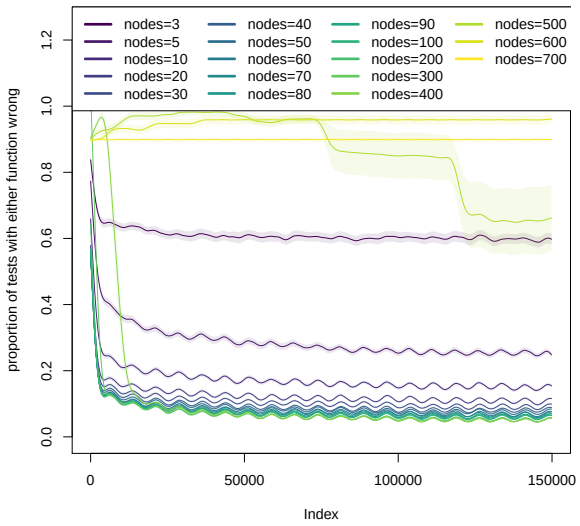


(b) $\eta = 0.2$

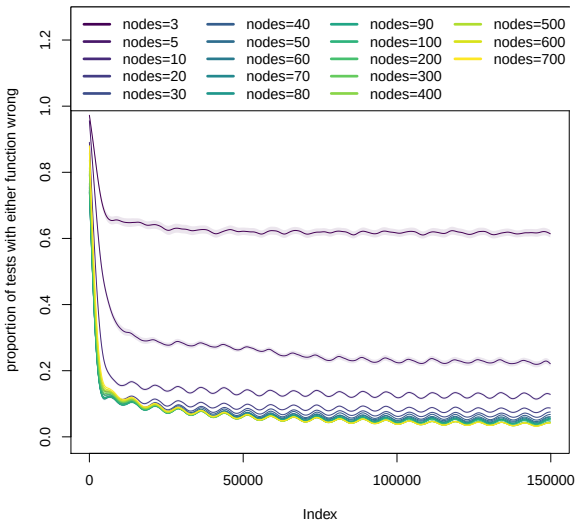


(c) $\eta = 0.05$

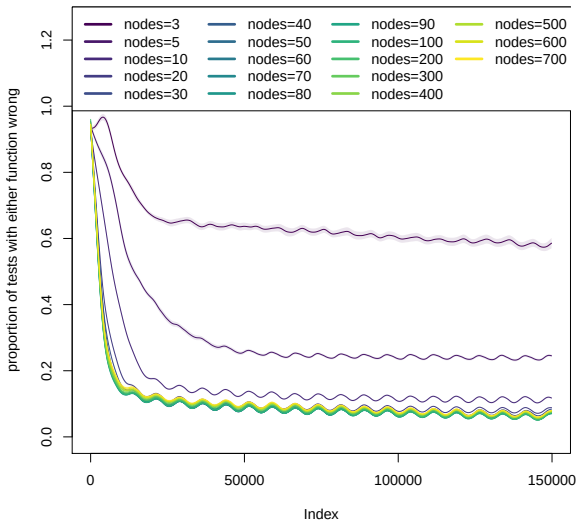
Figure 7.2: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ for plain back-propagation learning the nominal MNIST labelling, showing the means of all attempts for a given hidden node count, with shading delimiting the region within 0.25 standard deviations of the mean.



(a) $\eta = 1$



(b) $\eta = 0.2$

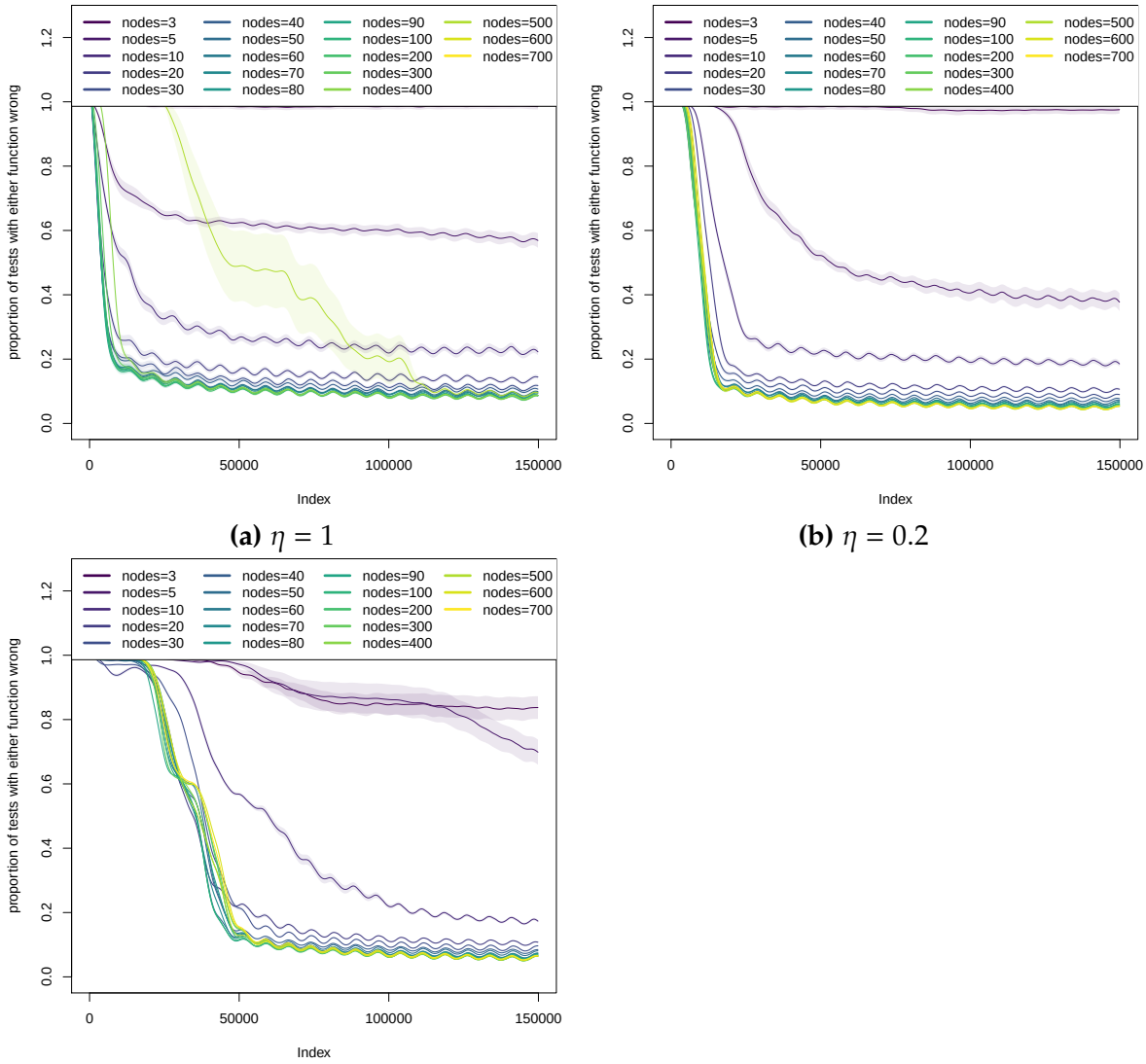


(c) $\eta = 0.05$

Figure 7.3: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ for output blending on MNIST, showing the means of all attempts for a given hidden node count, with shading delimiting the region within 0.25 standard deviations of the mean.

7.2.3 h-as-input convergence behaviour

Convergence plots for h-as-input are shown in Fig. 7.4. Here, all networks converge to solutions at all but the very lowest hidden node counts, but this convergence takes considerably longer and (in the case of $\eta = 0.05$) involves navigating at least one plateau. This is better behaviour than that for the line recognition experiment, where higher learning rates cause the networks to skip over the solution minima (compare Fig. 6.7).



(a) $\eta = 1$

(b) $\eta = 0.2$

(c) $\eta = 0.05$

Figure 7.4: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ for h-as-input on MNIST, showing the means of all attempts for a given hidden node count, with shading delimiting the region within 0.25 standard deviations of the mean.

7.2.4 UESMANN convergence behaviour

Convergence plots for UESMANN are shown in Fig. 7.5. The $\eta = 1$ networks perform poorly, converging to local minima at different performance levels for different hidden node counts. The higher node counts perform poorly. This indicates that the error surface is complex, particularly with higher dimensions, and routes into good minima cannot be found. Interestingly, the number of networks which converge to a solution is higher than that for line recognition, although those solutions may be worse (compare Fig 6.8). This seems to suggest that although the solution minima are worse than those for line recognition, the error surface topography is simpler (but still complex).

At lower learning rates the higher node counts perform better than the lowest node counts. Performance at $\eta = 0.05$ and $\eta = 0.2$ is similar, with mid-range node counts performing best at $\eta = 0.2$ and highest node counts performing best at $\eta = 0.05$. This indicates that the low learning rate is able to find routes down the complex surface. However, the highest node count performs poorly, perhaps finding its way into a local minimum.

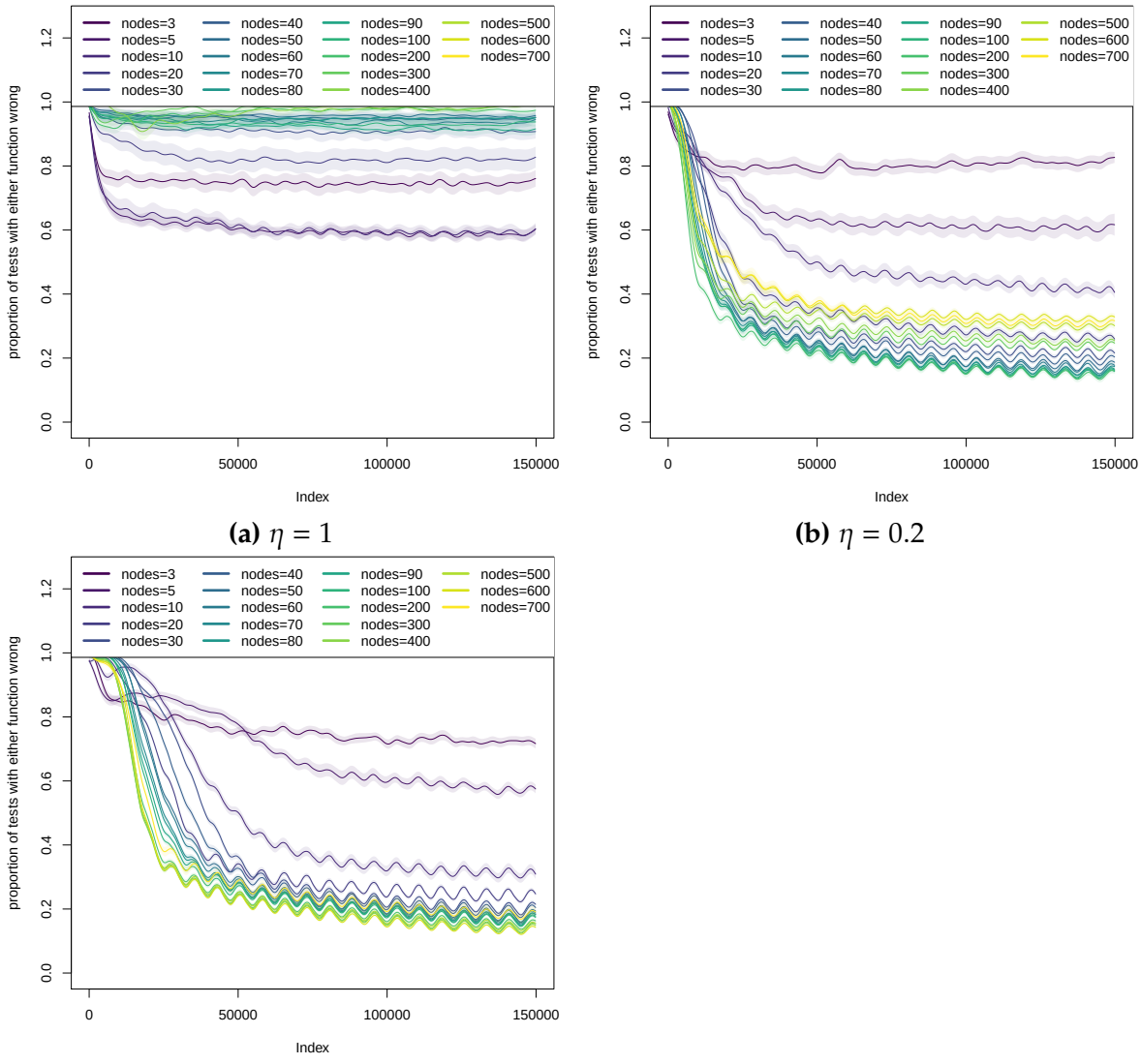
7.3 Performance of the different networks at $\eta = 0.05$

7.3.1 Generating a metric

As discussed in Sec. 5.1, generating a single performance metric for a multiclass classifier presents some difficulties. The metrics described in Sec. 5.1 require a table of confusion rather than a 10×10 confusion matrix. We are able to generate a table of confusion for every class, and once we have these tables two methods for obtaining the metrics are available:

- find the element-wise mean of the ten class tables, and find the metric on the result;
- calculate the metric for each of the class tables, and find the mean of the resulting values.

Data will be lost, however, since the classifier may perform differently on different classes. This is inevitable with any given metric. If we use both methods to find ϕ_{min} (our metric of choice, used in the previous section) for all networks of a particular type, and plot the difference between them against the second method, we obtain the



(a) $\eta = 1$

(b) $\eta = 0.2$

(c) $\eta = 0.05$

Figure 7.5: Smoothed convergence data for $\eta \in \{1, 0.2, 0.05\}$ for UESMANN on MNIST, showing the means of all attempts for a given hidden node count, with shading delimiting the region within 0.25 standard deviations of the mean.

plot in Fig. 7.6. This shows that the difference between the two methods is usually small, rising to approximately 0.09 when the performance of the network is poor.

Also, if we were to use the mean table method of deriving metrics, F_1 would always be equal to PPV. This is because every false positive is a false negative for another class, so $FN = FP$ in the mean table. This leads to $PPV = TPR$, and the F_1 score is the harmonic mean of those two values.

Feeling intuitively that it is better to generate the metric as late in the processing as possible, we have elected to find the tables of confusion for each class, calculate

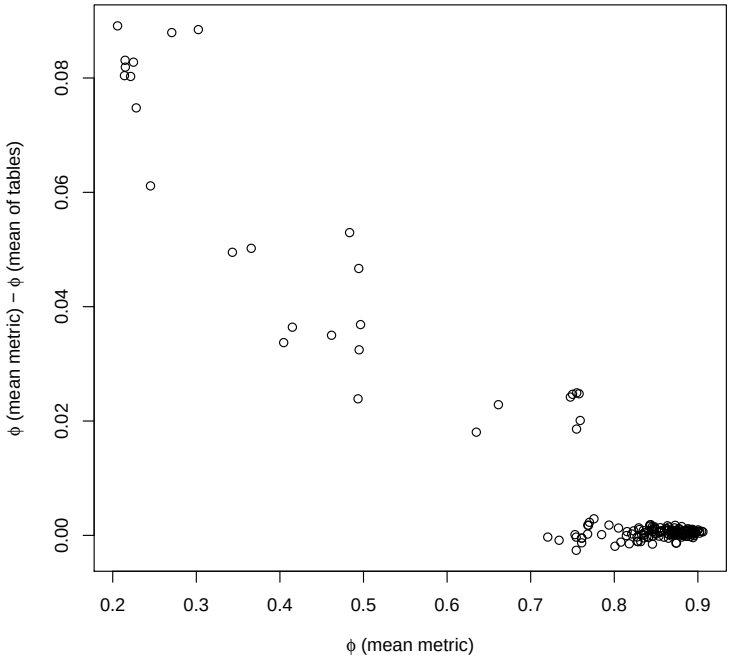


Figure 7.6: Difference between two different methods of calculating the Matthews correlation coefficient for all network results. Here, ϕ (mean of metric) is the mean of the coefficient for each table; ϕ (mean of tables) is the coefficient of an element-wise mean of all tables.

the metric of each table (the Matthews correlation coefficient ϕ_{min} as with the line recognition experiments) and find the mean of the results.

If we were free to use any metric, not one of those given in Sec. 5.1, one possibility would be the trace of the confusion matrix, which gives the number of correct classifications[1]: the true positive count. This may be useful in the multiclass case, but is equivalent to the simple accuracy (once division by the number of examples is factored in) in the binary case. In the multiclass case, it is sensitive to the class prevalences in the data. To ameliorate this, the values can be normalised by dividing by the sum of their columns (the class prevalences), although this makes every class equally important to the metric, even if extremely rare. We do not have this problem in the MNIST data set. Performing this operation will produce the true positive rate, TPR.

Another possibility is the minimum of the TPRs for all classes: this will give a figure for the worst performance, so that a network which recognises most classes perfectly but completely fails on one will give a poor result. This is useful in con-

[1]If the classes were ordered such that similar classes were adjacent, the "earth mover" distance between the matrix and the "perfect" confusion matrix could be used.

junction with the minimum Matthews correlation coefficient ϕ_{min}, which gives an average of the worst performances. We will refer to this metric as the minimum true positive rate, TPR_{min}.

7.3.2 ROC curves

ROC curves are not shown. This is because when aggregated across the classes there are nine true negatives for every true positive (since there are ten classes). Thus

$$TPR = 9TNR \tag{7.4}$$

where TPR is the false positive rate and TNR is the true negative rate. Since the false positive rate FPR is equal to $1 - TNR$,

$$TPR = 1 - (9FPR). \tag{7.5}$$

If we calculate the ROC curve using the element-wise mean of the class tables, all networks must fall on this line (because we are finding the total of all false positives and true negatives). We have already established that finding the mean of the metric for all class tables will be very close to this line, particularly for good networks. Exploratory plots showed little deviation from the line of Eq. 7.5. Thus it is unlikely the ROC curve will provide useful information.

7.3.3 Performance overview

If the data from the tables of confusion are combined into a single metric using the mean metric method described above, and the result plotted as was done for the line experiments in Fig. 6.10, we obtain Fig. 7.7. This clearly shows the difference in performance, and an increase in consistency over the line experiments within each network type, although this may be an artefact of the processing used to collate multi-class data into single tables of confusion. We would expect output blending to perform best, but h-as-input appears to outperform it once the hidden node count passes 80. The reason for this is not understood, and is not the focus of this thesis, but would be interesting to study in the future. UESMANN does not perform as well, but still achieves a ϕ_{min} of 0.906 and an minimum accuracy of 0.983. The best h-as-input network achieves $\phi_{min} = 0.949$, accuracy 0.991; while the best output blending network achieves $\phi_{min} = 0.942$, accuracy 0.990. These latter values are quite close, but well outside the variation within the networks as the box plot shows.

The results for best networks (in terms of the minimum ϕ) are in Tables 7.3, 7.4 and 7.5.

A notable feature is the rapid drop in mean performance for UESMANN at 700 hidden nodes, with the networks falling into a bimodal distribution. It appears that at this higher node count there are local minima which claim about half the initial networks. This explains the poor convergence behaviour observed for these networks in Fig. 7.5.

Table 7.3: Output blending best networks, sorted by ϕ_{min}. See Table 7.9 for the row header meanings.

Hidden nodes	run	NPV	PPV	Accuracy	F_1	TPR_{min}	ϕ_{min}
80	7	0.994	0.948	0.990	0.948	0.918	0.942
90	0	0.994	0.947	0.990	0.947	0.912	0.941
70	8	0.994	0.947	0.990	0.947	0.918	0.941
90	3	0.994	0.947	0.989	0.947	0.904	0.941
70	1	0.994	0.947	0.989	0.946	0.909	0.941
80	4	0.994	0.946	0.989	0.946	0.914	0.941

Table 7.4: h-as-input best networks, sorted by ϕ_{min}. See Table 7.9 for the row header meanings.

Hidden nodes	run	NPV	PPV	Accuracy	F_1	TPR_{min}	ϕ_{min}
300	1	0.995	0.954	0.991	0.954	0.920	0.949
600	7	0.995	0.954	0.991	0.953	0.913	0.948
300	6	0.995	0.953	0.991	0.953	0.925	0.947
300	8	0.995	0.953	0.991	0.953	0.928	0.947
300	5	0.995	0.953	0.991	0.953	0.926	0.947
300	0	0.995	0.953	0.991	0.952	0.919	0.947

Table 7.5: UESMANN best networks, sorted by ϕ_{min}. See Table 7.9 for the row header meanings.

Hidden nodes	run	NPV	PPV	Accuracy	F_1	TPR_{min}	ϕ_{min}
600	3	0.991	0.916	0.983	0.915	0.853	0.906
700	4	0.991	0.915	0.983	0.914	0.850	0.905
500	6	0.990	0.913	0.983	0.911	0.821	0.902
400	4	0.990	0.913	0.982	0.911	0.843	0.902
400	7	0.990	0.911	0.982	0.910	0.843	0.900
300	9	0.990	0.910	0.982	0.909	0.835	0.899

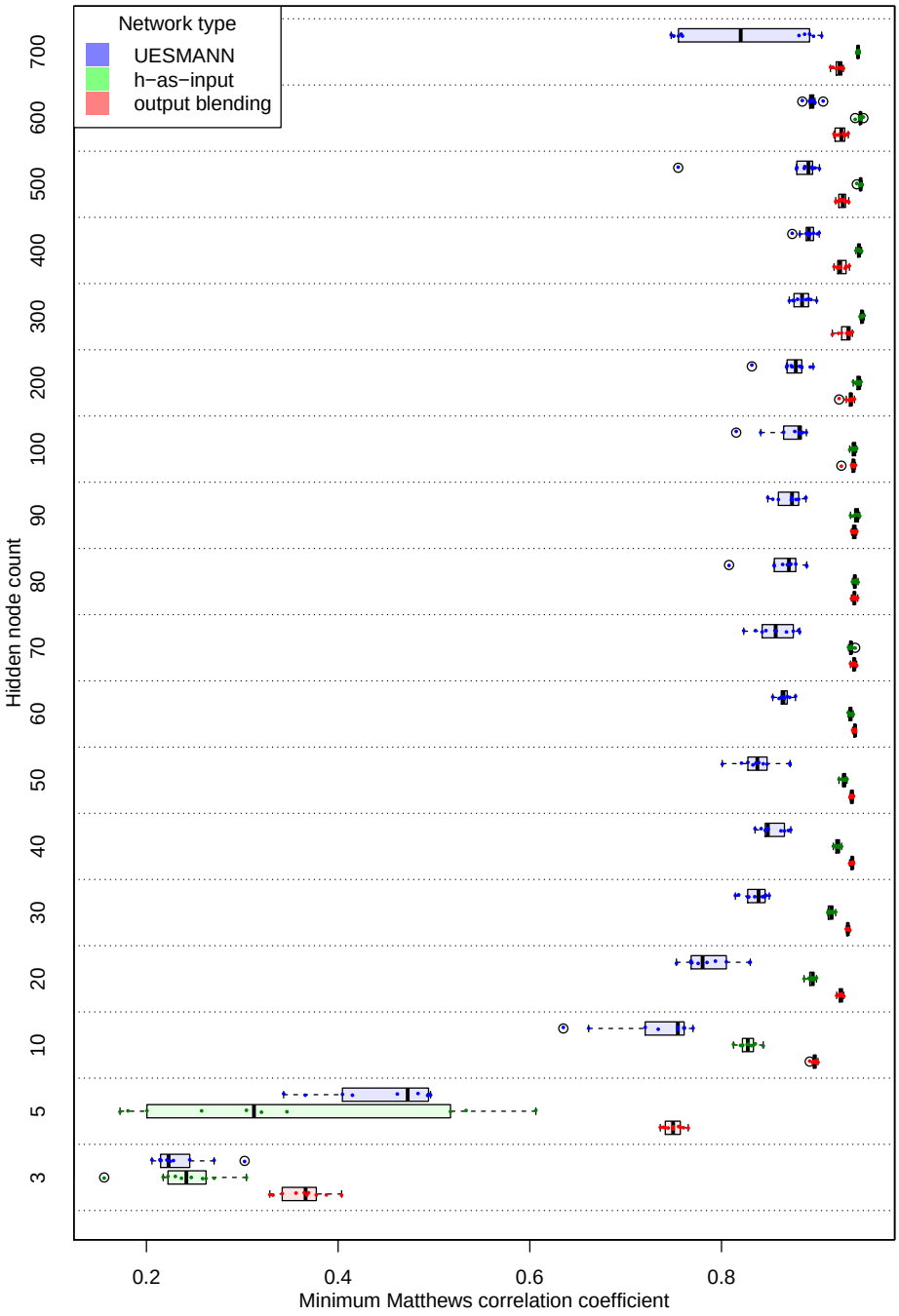


Figure 7.7: Box plot of minimum Matthews correlation coefficient for all networks of all three network types at $\eta = 0.05$, across both $h = 0$ and $h = 1$. The actual network points are shown as red dots.

7.3.4 Some example UESMANN confusion matrices

In this section, examples of confusion matrices will be shown from UESMANN networks with both good and bad performances, at different hidden node counts. All networks were trained at $\eta = 0.05$.

7.3.4.1 Five hidden nodes: a poor result

The value of the Matthews correlation coefficient was plotted at both $h = 0$ and $h = 1$ for this node count (Fig. 7.8), showing no significant difference between the two modulator levels (confirmed with a Wilcoxon rank-sum test: $p > 0.1$). Network run 3 (of 10) was selected for more detailed examination because it lies near the mean at both levels.

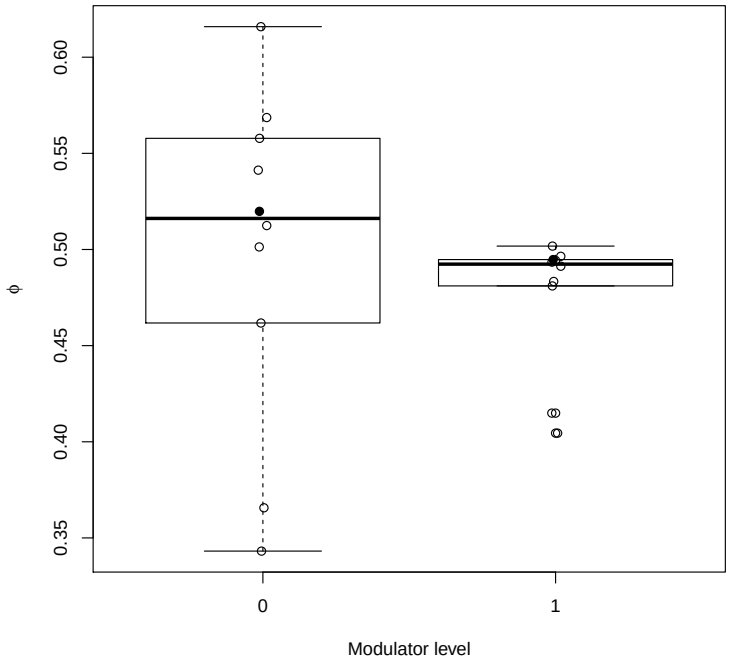


Figure 7.8: Box plot showing the value of the Matthews correlation coefficient ϕ for all UESMANN networks trained for MNIST at $\eta = 0.05$ with 5 hidden nodes. Network 3 is shown as a black dot, other networks are shown as circles.

Tables 7.6 and 7.7 show the confusion matrices for run 0 (of ten) of UESMANN at $\eta = 0.05$ with only five hidden nodes. Of the two modulation endpoints, the performance is best at $h = 0$, particularly for recognising "1" and "4". It is poor for the digits "5" and "8", however, with "5" often mistaken for "2" and "8" in particular mistaken for "9", probably due to the similarity in the shapes, while "1"

has no similarly shaped digits. Note also that "8" is never predicted, and "5" only three times (only two of which are correct).

Table 7.6: MNIST UESMANN 5 hidden nodes at $\eta = 0.05$, run 3, confusion matrix at $h = 0$

		Actual										
		0	1	2	3	4	5	6	7	8	9	Total
Predicted	0	**864**	59	37	14	0	35	12	22	30	13	1086
	1	24	**989**	5	4	1	38	25	6	5	1	1098
	2	43	18	**784**	419	18	305	99	12	77	11	1786
	3	17	3	119	**471**	4	103	2	19	35	9	782
	4	2	20	19	7	**911**	143	19	18	88	129	1356
	5	0	1	0	0	2	**2**	0	0	1	0	6
	6	3	1	11	4	11	4	**421**	61	4	4	524
	7	17	4	27	36	0	34	374	**804**	7	18	1321
	8	0	0	0	0	0	0	0	0	**0**	0	0
	9	10	40	30	55	35	228	6	86	727	**824**	2041
	Total	980	1135	1032	1010	982	892	958	1028	974	1009	10000
	TPR	0.882	0.871	0.760	0.466	0.928	0.002	0.439	0.782	0.000	0.817	

Table 7.7: MNIST UESMANN 5 hidden nodes at $\eta = 0.05$, run 3, confusion matrix at $h = 1$

		Actual										
		0	1	2	3	4	5	6	7	8	9	Total
Predicted	0	**964**	13	0	10	23	0	19	9	8	18	1064
	1	66	**879**	18	28	34	2	3	22	3	8	1063
	2	7	48	**789**	834	221	10	16	81	9	32	2047
	3	0	0	2	**1**	0	0	2	0	0	0	5
	4	0	0	0	0	**0**	2	0	1	0	0	3
	5	3	1	7	20	77	**754**	1	7	22	19	911
	6	0	1	12	14	14	12	**796**	66	24	9	948
	7	4	15	27	20	18	0	66	**726**	3	2	881
	8	26	18	111	78	260	26	80	39	**78**	128	844
	9	65	5	44	27	245	176	45	7	862	**758**	2234
	Total	1135	980	1010	1032	892	982	1028	958	1009	974	10000
	TPR	0.849	0.897	0.781	0.001	0.000	0.768	0.774	0.758	0.077	0.778	

For $h = 1$ we need to swap adjacent columns to find the actual digit being recognised given the alternative labelling described in Eq. 7.3[2]. We can see that the digit "1" (now labelled as "0") still has the best performance. The digit "5" (labelled "4") is the worst performer, and other digits with similar shapes ("2", "5" and "9") are also rarely classified correctly.

[2]Compare the column totals in Tables 7.6 and 7.7 to get a visual "handle" on the alternative labelling.

Informal experimentation with other networks at this node count shows a similar pattern, where digits which have distinct shapes are easily distinguished, while those which do not, fail.

The individual class confusion matrices for the $h = 0$ table are shown for reference in Table. 7.8.

Table 7.8: MNIST UESMANN 5 hidden nodes at $\eta = 0.05$, run 3, class confusion tables at $h = 0$

(a) Class 0

		Actual		
		0	not 0	Total
Predicted	0	**864**	222	1086
	not 0	116	**8798**	8914
	Total	980	9020	10000

(b) Class 1

		Actual		
		1	not 1	Total
Predicted	1	**989**	109	1098
	not 1	146	**8756**	8902
	Total	1135	8865	10000

(c) Class 2

		Actual		
		2	not 2	Total
Predicted	2	**784**	1002	1786
	not 2	248	**7966**	8214
	Total	1032	8968	10000

(d) Class 3

		Actual		
		3	not 3	Total
Predicted	3	**471**	311	782
	not 3	539	**8679**	9218
	Total	1010	8990	10000

(e) Class 4

		Actual		
		4	not 4	Total
Predicted	4	**911**	445	1356
	not 4	71	**8573**	8644
	Total	982	9018	10000

(f) Class 5

		Actual		
		5	not 5	Total
Predicted	5	**2**	4	6
	not 5	890	**9104**	9994
	Total	892	9108	10000

(g) Class 6

		Actual		
		6	not 6	Total
Predicted	6	**421**	103	524
	not 6	537	**8939**	9476
	Total	958	9042	10000

(h) Class 7

		Actual		
		7	not 7	Total
Predicted	7	**804**	517	1321
	not 7	224	**8455**	8679
	Total	1028	8972	10000

(i) Class 8

		Actual		
		8	not 8	Total
Predicted	8	**0**	0	0
	not 8	974	**9026**	10000
	Total	974	9026	10000

(j) Class 9

		Actual		
		9	not 9	Total
Predicted	9	**824**	1217	2041
	not 9	185	**7774**	7959
	Total	1009	8991	10000

The metrics for this network are given in Table 7.9. The low TPR_{min} shows that the network performs very badly on some classes, completely failing to detect some digits, as we have seen. The other metrics show the poor average performance at both modulator levels: three hidden nodes is insufficient to learn all digits reliably,

although we have seen that digits with distinctive shapes, such as "1" and "4", can be learned, and that the modulation functions even at this very low node count.

Table 7.9: Performance metrics for MNIST UESMANN 5 hidden nodes at $\eta = 0.05$, run 0, showing the performance at both modulator levels and the minimum performance. The metrics are the positive predictive value $TP/(TP + FP)$, the negative predictive value $TN/(TN + FN)$, the accuracy, the F_1 score, the minimum true positive rate (see Sec. 7.3.1) and the Matthews correlation coefficient ϕ.

Modulator h	NPV	PPV	F_1	TPR_{min}	ϕ
0	0.958	0.556	0.538	0.000	0.520
1	0.954	0.524	0.520	0.000	0.495
min	0.954	0.524	0.520	0.000	0.495

7.3.4.2 40 hidden nodes: a better performance

Fig. 7.9 shows the box plot for UESMANN network performance 40 hidden nodes at both modulator levels. Network 2 was selected for confusion matrix analysis given its proximity to the mean at both modulator levels. At this hidden node count, the $h = 1$ result is significantly better than the $h = 0$ result ($p < 0.01$, Wilcoxon rank-sum test).

Tables 7.10 and 7.11 show the confusion matrices for the best network with 40 hidden nodes. The performance here is better, with a true positive rate of at least 72% on all classes (this is still a poor performance compared with other modulation methods at this node count). At $h = 0$, the network performs well on "0", "1" and "4", and poorly on "5", which is now mistaken for "4" — interestingly, given that "4" is recognised successfully. At $h = 1$ the performance is a little better (as can be seen in Fig. 7.9), with the worst performances again on digit "5" (labelled as "4"). The metrics are shown in Table 7.12: the higher TPR_{min} shows that the worst performance of the network is much higher than that of the network with only 5 hidden nodes, as we should expect.

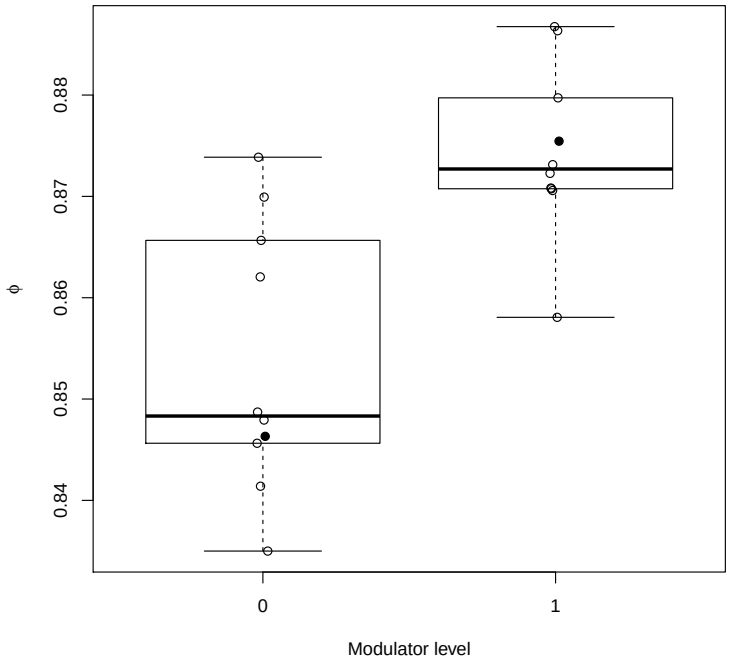


Figure 7.9: Box plot showing the value of the Matthews correlation coefficient ϕ for all UESMANN networks trained for MNIST at $\eta = 0.05$ with 40 hidden nodes. Network 2 is shown as a black dot, other networks are shown as circles.

Table 7.10: MNIST UESMANN 40 hidden nodes at $\eta = 0.05$, run 2, confusion matrix at $h = 0$

		Actual										
		0	1	2	3	4	5	6	7	8	9	Total
Predicted	0	**924**	30	10	0	0	4	6	1	5	5	985
	1	19	**1083**	9	5	5	4	5	17	6	6	1159
	2	3	3	**883**	142	4	8	5	37	6	7	1098
	3	3	4	64	**812**	0	27	1	3	35	8	957
	4	4	1	21	7	**929**	158	19	13	27	43	1222
	5	6	1	6	15	14	**643**	17	0	15	6	723
	6	10	2	15	1	9	9	**849**	91	16	0	1002
	7	5	0	14	12	2	12	50	**847**	7	13	962
	8	6	9	9	9	5	17	5	1	**789**	57	907
	9	0	2	1	7	14	10	1	18	68	**864**	985
	Total	980	1135	1032	1010	982	892	958	1028	974	1009	10000
	TPR	0.943	0.954	0.856	0.804	0.946	0.721	0.886	0.824	0.810	0.856	

Table 7.11: MNIST UESMANN 40 hidden nodes at $\eta = 0.05$, run 2, confusion matrix at $h = 1$

		Actual										
		0	1	2	3	4	5	6	7	8	9	Total
Predicted	0	**1047**	14	0	4	2	1	6	3	4	4	1085
	1	62	**934**	1	6	7	2	3	11	6	4	1036
	2	5	4	**887**	73	44	0	9	0	14	20	1056
	3	3	2	70	**885**	3	9	23	3	3	10	1011
	4	1	6	31	12	**761**	63	4	24	14	33	949
	5	1	2	1	10	31	**871**	0	9	28	3	956
	6	1	7	12	18	11	9	**935**	48	15	17	1073
	7	3	9	0	16	11	5	39	**851**	4	14	952
	8	0	0	2	4	5	17	7	1	**870**	25	931
	9	12	2	6	4	17	5	2	8	51	**844**	951
	Total	1135	980	1010	1032	892	982	1028	958	1009	974	10000
	TPR	0.922	0.953	0.878	0.858	0.853	0.887	0.910	0.888	0.862	0.867	

Table 7.12: Performance metrics for MNIST UESMANN 40 hidden nodes at $\eta = 0.05$, run 2. See Table 7.9 for more details.

Modulator h	NPV	PPV	F_1	TPR_{min}	ϕ
0	0.985	0.865	0.860	0.721	0.846
1	0.988	0.888	0.888	0.853	0.875
min	0.985	0.865	0.860	0.721	0.846

7.3.4.3 600 nodes: the best UESMANN performance

Although some UESMANN networks achieve a good performance with fewer nodes, the networks with 600 hidden nodes are consistent. The performance at the different modulator levels is shown in Fig. 7.10, which again shows a higher performance at $h = 1$ ($p < 0.01$, Wilcoxon rank-sum test). We will examine the confusion matrices for network 6, again because both performances are close to the means.

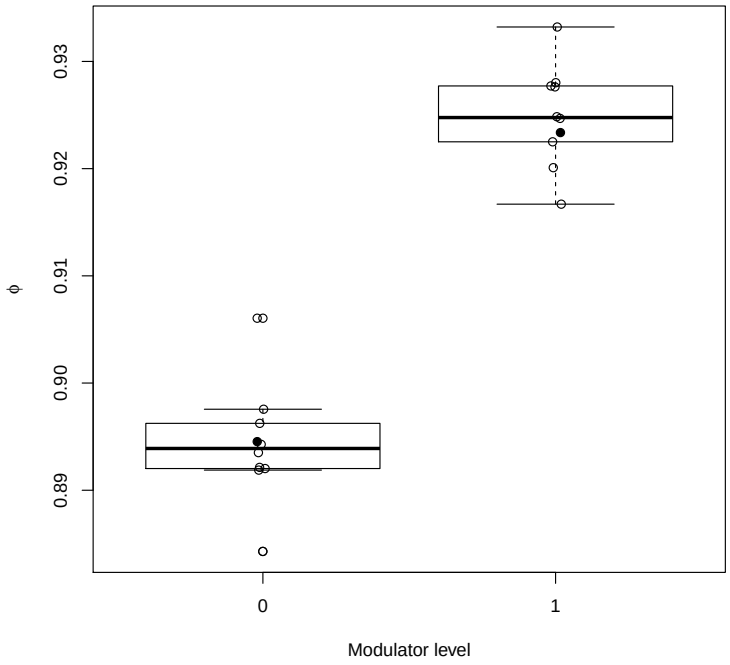


Figure 7.10: Box plot showing the value of the Matthews correlation coefficient ϕ for all UESMANN networks trained for MNIST at $\eta = 0.05$ with 600 hidden nodes. Network 6 is shown as a black dot, other networks are shown as circles.

Tables 7.13 and 7.14 show the confusion matrices. These show the poorest performance is once again on the digit "5" at both modulator levels, and the best on the digit "1". Once again, this is likely to be because "5" is too similar to other digits, while "1" is distinctive. The metrics are shown in Table 7.15, with a TPR_{min} showing the worst performance — the 77.8% accuracy on "5" at $h = 0$. The other metrics are also higher, as expected.

Table 7.13: MNIST UESMANN 600 hidden nodes at $\eta = 0.05$, run 6, confusion matrix at $h = 0$

		Actual										
		0	1	2	3	4	5	6	7	8	9	Total
Predicted	0	**963**	2	13	2	2	10	11	5	7	11	1026
	1	0	**1098**	5	0	2	1	5	7	1	5	1124
	2	0	1	**803**	31	1	4	1	19	0	0	860
	3	1	8	154	**940**	5	19	0	16	11	9	1163
	4	0	0	6	1	**849**	41	4	3	9	11	924
	5	5	1	2	10	43	**775**	11	0	9	7	863
	6	6	6	11	0	23	6	**883**	20	6	1	962
	7	3	2	14	10	3	9	35	**937**	6	8	1027
	8	2	17	18	10	10	18	8	3	**901**	53	1040
	9	0	0	6	6	44	9	0	18	24	**904**	1011
	Total	980	1135	1032	1010	982	892	958	1028	974	1009	10000
	TPR	0.983	0.967	0.778	0.931	0.865	0.869	0.922	0.911	0.925	0.896	

Table 7.14: MNIST UESMANN 600 hidden nodes at $\eta = 0.05$, run 6, confusion matrix at $h = 1$

		Actual										
		0	1	2	3	4	5	6	7	8	9	Total
Predicted	0	**1089**	29	0	2	2	0	2	3	5	1	1133
	1	22	**933**	0	6	3	1	2	6	4	6	983
	2	2	0	**902**	22	14	0	1	1	8	5	955
	3	7	3	84	**982**	6	7	34	1	4	7	1135
	4	1	1	9	2	**802**	14	1	3	1	9	843
	5	1	1	1	3	37	**920**	2	2	11	4	982
	6	0	4	5	7	4	6	**939**	24	5	6	1000
	7	4	7	1	5	9	2	33	**912**	2	11	986
	8	0	1	3	2	7	28	12	1	**945**	39	1038
	9	9	1	5	1	8	4	2	5	24	**886**	945
	Total	1135	980	1010	1032	892	982	1028	958	1009	974	10000
	TPR	0.959	0.952	0.893	0.952	0.899	0.937	0.913	0.952	0.937	0.910	

Table 7.15: Performance metrics for MNIST UESMANN 600 hidden nodes at $\eta = 0.05$, run 6. See Table 7.9 for more details.

Modulator h	NPV	PPV	F_1	TPR_{min}	ϕ
0	0.990	0.907	0.904	0.778	0.895
1	0.992	0.932	0.931	0.893	0.923
min	0.990	0.907	0.904	0.778	0.895

7.3.4.4 700 hidden nodes: a bimodal performance at $h = 0$

As noted above the performance of UESMANN falls at 700 hidden nodes. To examine this behaviour further, the performances of the networks were plotted separately for $h = 0$ and $h = 1$ as for previous node counts. This gives the plot in Fig. 7.11, which shows that the bimodal performance in Fig. 7.7 comes from a bimodal distribution in ϕ at $h = 0$ only: at $h = 1$ the network performs well, achieving a mean of 0.929. While the $h = 1$ performance of the 600 hidden node network has a mean ϕ of 0.924, the difference is not significant (Wilcoxon rank-sum test, $p > 0.01$). However, this shows that at 700 hidden nodes good performances are achievable, but about half the networks are finding a local minimum at $h = 0$.

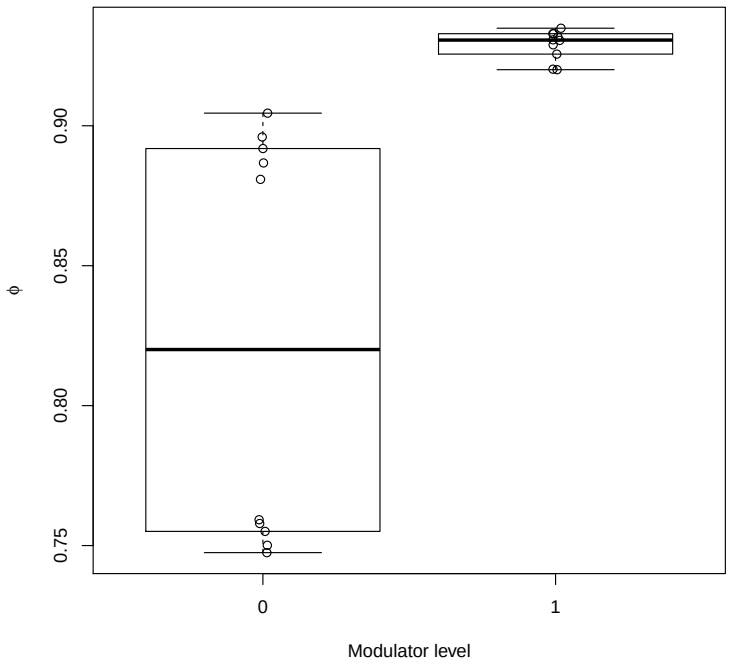


Figure 7.11: Box plot showing the value of the Matthews correlation coefficient ϕ for all UESMANN networks trained for MNIST at $\eta = 0.05$ with 700 hidden nodes.

7.3.4.5 Summary

The failure mode of UESMANN networks with low node counts is to confuse digits with common shape features: even without examining the weights and activations of the network, we can say that the shapes “5” and “2” have features in common, as do the shapes “3” and “8”. Certain digits become activated by a larger range of features than they should be, because of the low number of feature detectors available: for

example, Table 7.6 shows "2" "2", "3", and "5"; "9" being activated by "8" and "9"; and "5" and "8" receiving no (or very little) activation. This is a similar mode of failure to that seen in plain back-propagation: consider the confusion matrix for a typical (near-mean performance) plain back-propagation network with 3 hidden nodes shown in Table 7.16. We see here that "2" is strongly activated by "2", "3", "5", "6" and "8", while "4" and "8" receive no activation. More feature detectors become available at higher node counts, so this failure mode disappears and the system improves by combining the hidden nodes in more complex ways.

Table 7.16: Confusion matrix for plain back-propagation with 3 hidden nodes at $\eta = 0.05$, run 1, nominal labelling

		Actual										
		0	1	2	3	4	5	6	7	8	9	Total
Predicted	0	**968**	0	21	13	3	31	31	2	12	7	1088
	1	0	**1107**	6	0	2	3	3	7	6	7	1141
	2	4	21	**864**	723	16	552	798	3	668	10	3659
	3	0	3	33	**106**	31	78	54	5	143	5	458
	4	0	0	0	0	**0**	0	0	0	0	0	0
	5	4	1	49	82	93	**140**	53	7	79	36	544
	6	0	0	0	0	0	0	**0**	0	0	0	0
	7	4	1	16	15	78	11	2	**960**	8	309	1404
	8	0	0	0	0	0	0	0	0	**0**	0	0
	9	0	2	43	71	759	77	17	44	58	**635**	1706
	Total	980	1135	1032	1010	982	892	958	1028	974	1009	10000
	TPR	0.988	0.975	0.837	0.105	0.000	0.157	0.000	0.934	0.000	0.629	

One interesting feature of the final metrics is that the $h = 0$ performance is worse than the $h = 1$ performance, most noticeably at higher node counts where the metrics are more consistent (see Table 7.15). This was tested for all hidden node counts, with the differences shown in Fig. 7.12. The reverse holds for the line recognition experiments. The reason for this consistent but different performance at the different modulator levels is not understood, given that the two modulator functions are (in a sense) equivalent, and requires further study. It may be related to the finding in Sec. 7.2.2, that one of the two functions appears to be harder to train (although this should not be the case).

However, UESMANN is clearly able to learn to recognise handwritten digits in this simple experiment, using a very simple back-propagation algorithm with no enhancements.

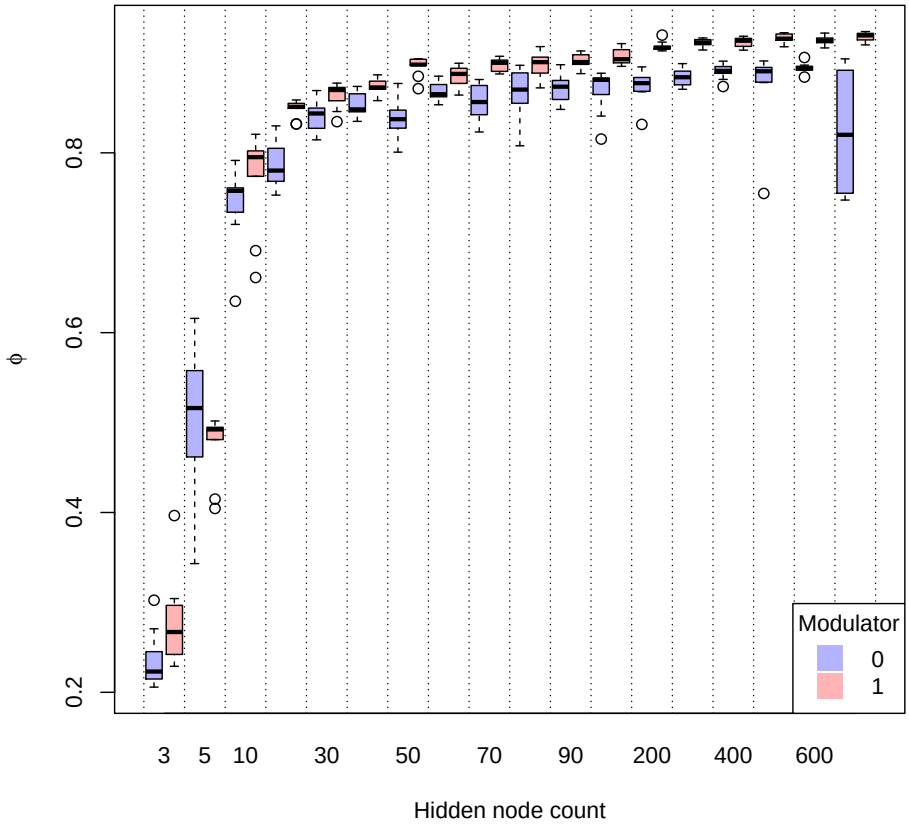


Figure 7.12: Values of ϕ for UESMANN MNIST at all hidden node counts at $\eta = 0.05$, plotted separately for $h = 0$ and $h = 1$.

7.3.5 Transition behaviour

A plot of how all outputs change as the modulator varies, as was done for the line recognition experiments, would be difficult to read. Here a different approach is used:

- For all ten networks trained of each type, hidden node count, and learning rate, 100 examples were passed to the network at 100 different values of h, and both the label and the network output (i.e. the index of the most strongly activated output node) were recorded.
- Within this data, the mode of the predicted digit at each value of h was found for each network. This gives the most common digit output from the network for examples with a given label at a given h value. In other words, it gives how the network tends to predict each digit at each h value.

- These labels are then pasted together to form a ten-digit "function signature" describing the most common function performed by the network at that h value. We now have a table which gives the function most commonly performed by each network at each h value.

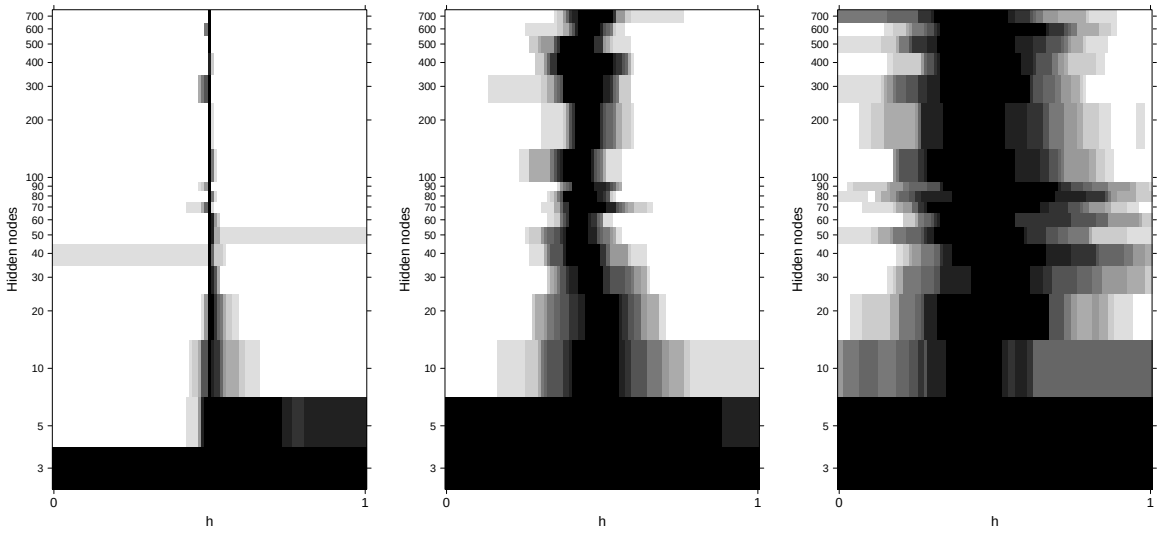
- Two function signatures are special: those which are given in the problem specification. They are "0123456789" which should be generated at $h = 0$, and "1032547698" which should be generated at $h = 1$. We have, at each h value, a value representing the function which is most commonly performed by each network. We now append a numeric value to each row in the table, which is 1 if each network performs one of these two end-point functions at that h value, and 0 otherwise.

- We aggregate the table by the network number (there are ten for each network type and hidden node count), finding the mean of the "is end-point" value at each modulator level — this will be 1 if all networks produce an end-point function, and zero if none do so.

We can now plot the proportion of networks for each hidden node count which are performing either of the two end-point functions at each h level. The width of the region in which non-end-point functions are being performed will be the size of the transition from one function to another. Plots for the three network types are shown in Fig. 7.13.

The narrow transition region for output blending is consistent with previous experiments. The UESMANN region is wider than that of h-as-input, which is also similar to transition behaviour we have seen before. There is a slight tendency for networks at lower node counts to have a wider transition region, but this generally occurs where the network is performing poorly. UESMANN's wide transitions also occur at node counts where the network is performing well.

To show the transition behaviour of an individual network, a similar technique was used. At each h value, the full 10000 example training set was presented to the network. Each example's label and the index of the highest output (i.e. the predicted label) were recorded. The mode of the predicted label for each example label was calculated at each h value. For these examples, networks with 60 hidden nodes were used: this is the lowest node count which produces consistent performance. Table 7.17 shows the output blending result, which shows a sharp transition from the $h = 0$ to $h = 1$ function. Only $h = 0.5$ shows an intermediate response, as we might expect: the two network outputs are precisely balanced here.



(a) Output blending **(b)** h-as-input **(c)** UESMANN

Figure 7.13: Transition region tendencies for the three network methods trained for MNIST, running on the first 100 examples in the held-out test set. White denotes that an end function (i.e. one of the two functions for which the networks were trained) is being performed, while black denotes that some other function is being performed. See Sec. 7.3.5 for details of how these plots were constructed.

Table 7.17: Most common output given the test MNIST examples passed to a typical 60 hidden node output blending network at each value of h. The transition region is delineated by vertical lines.

h	0.00	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1.00
Label 0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1
Label 1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
Label 2	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	3	3	3	3	3	3	3	3	3	3
Label 3	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	2	2	2	2	2	2	2	2	2	2	2
Label 4	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	5	5	5	5	5	5	5	5	5	5	5
Label 5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	4	4	4	4	4	4	4	4	4
Label 6	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	7	7	7	7	7	7	7	7	7	7	7
Label 7	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	6	6	6	6	6	6	6	6	6	6
Label 8	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	9	9	9	9	9	9	9	9	9	9	9
Label 9	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	8	8	8	8	8	8	8	8	8	8

Table 7.18 shows the transition of a typical h-as-input network. This shows a narrow transition region, with three h values showing an intermediate response. Bold numerals indicate the network is outputting the $h = 0$ labelling, while normal numerals indicate the $h = 1$ labelling.

Table 7.19 shows the transition for a UESMANN network. In this particular network, the transition is wide but is complete at $h = 0.45$. Table 7.20 shows the

Table 7.18: Most common output given the test MNIST examples passed to a typical 60 hidden node h-as-input network at each value of h. The transition region is delineated by vertical lines. Bold numerals indicate the network is outputting the $h = 0$ labelling, while normal numerals indicate the $h = 1$ labelling.

h	0.00	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1.00
Label 0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1
Label 1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0
Label 2	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	3	3	3	3	3	3	3	3	3	3	3
Label 3	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	2	2	2	2	2	2	2	2	2	2	2	2
Label 4	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	5	5	5	5	5	5	5	5	5	5	5
Label 5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	4	4	4	4	4	4	4	4	4	4	4
Label 6	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	7	7	7	7	7	7	7	7	7	7	7
Label 7	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	6	6	6	6	6	6	6	6	6	6	6
Label 8	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	9	9	9	9	9	9	9	9	9
Label 9	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	8	8	8	8	8	8	8	8	8	8

transition for a different UESMANN network: the transition is slightly wider and more central. Transition regions for different networks trained with the same parameters are not necessarily the same (as also shown by the grey regions in Fig. 7.13).

Table 7.19: Most common output given the test MNIST examples passed to a 60 hidden node UESMANN network (attempt 2) at each value of h. The transition region is delineated by vertical lines.

h	0.00	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1.00
Label 0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1
Label 1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0
Label 2	**2**	**2**	**2**	**2**	**2**	**2**	**2**	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Label 3	**3**	**3**	**3**	**3**	**3**	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Label 4	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	5	5	5	5	5	5	5	5	5	5	5	5	5
Label 5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	4	4	4	4	4	4	4	4	4	4	4
Label 6	**6**	**6**	**6**	**6**	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Label 7	**7**	**7**	**7**	**7**	**7**	**7**	**7**	6	6	6	6	6	6	6	6	6	6	6	6	6	6
Label 8	**8**	**8**	**8**	**8**	**8**	**8**	**8**	9	9	9	9	9	9	9	9	9	9	9	9	9	9
Label 9	**9**	**9**	**9**	**9**	**9**	**9**	**9**	8	8	8	8	8	8	8	8	8	8	8	8	8	8

Table 7.20: Most common output given the test MNIST examples passed to a 60 hidden node UESMANN network (attempt 7) at each value of h. The transition region is delineated by vertical lines.

h	0.00	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1.00
Label 0	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Label 1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0
Label 2	**2**	**2**	**2**	**2**	**2**	**2**	**2**	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Label 3	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	2	2	2	2	2	2	2	2	2	2	2	2
Label 4	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	5	5	5	5	5	5	5	5	5	5	5	5	5
Label 5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	4	4	4	4	4	4	4	4	4
Label 6	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	7	7	7	7	7	7	7	7	7	7	7
Label 7	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	6	6	6	6	6	6	6	6	6	6	6	6	6
Label 8	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**8**	9	9	9	9	9	9	9	9	9	9	9
Label 9	**9**	**9**	**9**	**9**	**9**	**9**	**9**	**9**	8	8	8	8	8	8	8	8	8	8	8	8	8

7.4 Conclusion

The experiments above show that UESMANN is able to learn two different labellings in a classification problem using simple back-propagation, which we have already shown requires a careful balancing of weights and biases across the network, not just in the output layer (see Sec. 6.5.3.3). The performance of UESMANN on this problem is not as good as output blending or h-as-input. While it should be noted that these two methods require many more parameters at the same hidden node count (particularly output blending) the UESMANN performance is still noticeably weaker and requires more hidden nodes, removing the parameter count advantage. However, it may be possible to improve the performance by using some of the many enhancements to back-propagation which have not been used here (such as the various types of momentum and weight decay). It should also be borne in mind that there was insufficient time to run the training to convergence in all cases: given the extra training time required for UESMANN networks, it may be that the performances of the three networks types are considerably closer when given more time to converge. The best performances are shown in Table 7.21.

Table 7.21: Performances of the best networks of each type on the MNIST handwriting recognition problem.

Type	hidden nodes	run	NPV	PPV	Accuracy	F_1	TPR_{min}	ϕ_{min}
h-as-input	300	1	0.995	0.954	0.991	0.954	0.920	0.949
Output blending	80	7	0.994	0.948	0.990	0.948	0.918	0.942
UESMANN	600	3	0.991	0.916	0.983	0.915	0.853	0.906

In comparison with the line recognition problem, UESMANN performs worse than its counterparts here. In those experiments, UESMANN's performance was comparable to h-as-input at intermediate to high hidden node counts. However, the convergence behaviour is different at higher learning rates: in the line experiments, UESMANN failed to converge to solutions at $\eta = 1$ and $\eta = 0.2$, requiring the lower $\eta = 0.05$ to find a solution. In the MNIST experiments, $\eta = 0.2$ was able to find solutions comparable to those at $\eta = 0.05$. This suggests that the error surfaces in the line experiments had a more complex topography (for the same hidden node count) which required that the algorithm find narrow routes to the solution, while in MNIST routes to the solution (however poor) were easier to find. It may also be the case that better solutions exist, and that a lower learning rate (or some of the enhancements to back-propagation mentioned in Sec. 4.1.6) might find them.

There are two unexplained features of the results which require more study: firstly, the h-as-input behaviour is consistently (if only slightly) better than that of output blending at higher node counts. This was not predicted: output blending learns the two problems independently in two separate networks, and so should perform better than trying to learn two functions in a single network with an extra "pixel" of data identifying the function. While this behaviour is interesting (and potentially useful), it is not within the scope of this thesis.

The second unexplained feature is the propensity for UESMANN to consistently perform better at one modulator level than the other — in this case, $h = 1$ has a better performance. This may be related to the finding that the output blending convergence and control (i.e. single function) performances do not follow the expected relationship $E_{ob} = 2E_c - E_c^2$, which appears to suggest that the two functions are not equally difficult to learn. This should not be the case, given that the difference between the two functions is a simple transformation of the output layer. Again, we must leave this for future work.

The transition behaviour of the three network types is consistent with that seen in previous experiments: output blending produces a crisp transition, h-as-input produces a narrow transition region, and UESMANN produces a wide transition. This may be useful in certain applications, such as control: rather than having a "hard switch" between two behaviours, it may be beneficial for some features of one behaviour to gradually give way to those of another. This may avoid (or at least damp) oscillations between the behaviours. We will investigate this possibility in the next part of this dissertation.

Part IV

UESMANN in a Homeostatic Control Problem

Chapter 8

We have seen that it is possible to train a multilayer perceptron with a sigmoid activation function to perform two different functions using back-propagation of errors, performing the second function when the weights are doubled. Thus far, we have studied the properties and performance of such a network in classification problems, yielding either a boolean result by thresholding at 0.5 (in the case of training for boolean functions and line recognition) or a classification (by selecting the highest output). This novel network, UESMANN, is able to learn such function pairings, but may not perform as well as naïvely interpolating the outputs or using an extra input to carry the modulator. However, it shows interesting behaviour in the transition when the modulator is between the two endpoints (0 and 1).

We can therefore answer two of our original research questions on p. 9: it is possible to build such a system, and it can be made considerably more simple than existing systems (of which the Neal/Timmis system described in Sec. 2.5.6 is perhaps the most simple). Our remaining questions ask what engineering advantages such a system might have, and what we might learn about biology from it. Leaving the latter question for the time being, we have seen a possible answer to the former: the transition region between the two modulator extrema might have some interesting properties. The output blending method shows (indeed, is comprised of) a linear interpolation between the outputs of the two networks of which it is composed: in classification problems this tends to manifest as a sharp transition, because a perfectly trained network will output 0 and 1 at its extrema[1]. A h-as-input network will tend to produce a wide transition which is fairly predictable, but UESMANN produces the widest transition of the three with different initial weights leading to correct behaviour at the end points but sometimes quite different transitions (see Figs. 4.43, 6.18, and 7.13). However, these transitions vary less than those for weight blending. This is partly because there are usually fewer solutions for a UESMANN

[1]If the networks are imperfectly trained and produce different values, we might obtain a more complex transition but it is likely to be narrow.

network than for the two independent networks used in weight blending, but also may be because the network is, in itself, a unitary object which is (in some sense) a compromise between the two solutions.

This transition behaviour might be useful in a control problem, providing a graded (but non-linear) response to a graded stimulus. This might prevent oscillatory behaviour: in a system with a narrow or hard transition, if the process variable (to use the control systems theory terminology) crosses the threshold a large change will be seen in the output, which is likely to cause the process variable to cross the threshold again in the other direction. Too wide a transition may also cause problems: consider a robot whose $h = 1$ is phototaxis. If the transition encompasses the entire modulatory range, such a robot would exhibit a slight amount of phototaxis with even a small amount of modulator. This might cause a nominally straight path (perhaps to explore or obtain resources) to become curved towards the light, leading to inefficiency. Naturally, this could be "fixed" with some form of non-linear mapping (such as a sigmoid), but the underlying nature of the transition remains.

Traditional control systems, such as PID control, ameliorate oscillatory behaviour with graded changes but require design by hand. "Traditional" artificial endocrine systems (such as the Neal/Timmis AES) are also typically designed by hand, although their constituent networks may be trained[2]. UESMANN uses a global modulator to switch gradually from one learned behaviour to another in a simple network.

We noted above that different networks trained on the same input might produce different transition behaviour, finding different solutions because of the stochastic initial parameters. We may find useful non-linearities for a given problem in the transition behaviours of some of these networks.

This work is largely motivated by the same problem domain as most AES work: adaptive behaviour in robotic systems. The key adaptive behaviour is homeostasis: maintaining essential variables within a fixed range. We will therefore test UESMANN in a homeostatic robotic problem, to assess whether the wide transition range (and its variability in different networks trained with the same hyperparameters) produces useful behaviour, and whether unforeseen emergent behaviours from non-linearities in the transition can be adaptive.

[2]There are exceptions: in some AESs, such as the Timmis/Neal/Thorniley system described in Sec. 2.5.6.4, the hormone response can be learned — unlike UESMANN, however, these systems are complex.

8.1 Methodology

The primary motivation for neuromodulatory systems (and artificial endocrine systems in particular) is the achievement of adaptive behaviour: behaviour which changes according to the environment, in order to permit more useful work to be done. In the context of biology, this is the "useful work" of continuing the species; in robotic systems it is whatever the engineer requires.

In the simple neuromodulatory systems under study (UESMANN, output blending, and h-as-input), our adaptive behaviour consists of switching between two behaviours for which the network has been trained using a large number of examples (i.e. supervised learning). Our control problem is therefore a regression problem, or rather two regression problems (given that we have two functions) combined into one. In the case of output blending, we solve the two problems separately and interpolate between their results; in the case of h-as-input we create a single regression problem using which of the two problems we are solving as an extra input; and in the case of UESMANN we attempt to find a solution to both problems in which a global multiplicative parameter interpolates (non-linearly) between the two solutions. This is not a completely "adaptive" system, in that the two sub-problems have fixed solutions which have already been found by other means (in order to generate the training examples). It is, however, a similar approach to that of Neal and Timmis [209] and Sauzé and Neal [243]. The adaptation is in the transition between the two modes of operation.

As was noted above, a key feature of adaptive behaviour is homeostasis. A typical homeostatic problem is the maintenance of available energy (in robots, typically battery charge) — this is referred to as *energy homeostasis* both within biology [202] and in the SYMBRION and REPLICATOR projects [147, 128].

Balancing the need to perform a task and the need to recharge or refuel is a basic problem in self-sufficient autonomous robots [192]. A good example is the robotic vacuum cleaner: this must clean effectively, while also returning to a charging station regularly. If it remains close to its charging station, it will not clean a large area; while if it ventures too far, it will discharge and require rescue [141, 291, 306]. We can refer to the work to be done (cleaning in this example) as the "primary" task, while maintaining energy levels is the "secondary" task.

In our experiments a task loosely termed "exploration" — simply heading away from obstacles while maintaining as much speed as possible — was chosen as the primary task for several reasons:

- The UESMANN architecture and the other modulatory methods used for comparison are designed to transition between two behaviours — recharging and exploration are two very different behaviours.
- Often the primary task requires covering a wide area by some exploratory algorithm [128]: robot vacuum cleaners are the prototypical example [98, 306]. Other applications include search and rescue [104] and planetary exploration [250].
- Avoiding areas already explored was thought to add too much complexity which might confound the results, although this could be achieved with approaches such as stigmergy [68].
- The two tasks (recharging by phototaxis and the primary exploration task) are easily defined and it is straightforward to produce examples from which the networks can be trained.
- The tasks are conceptually similar to the robot sailing problem used by Sauzé and Neal [243] and Neal and Timmis [208] (which use the Neal/Timmis AES).
- Both tasks are very simple and are intended to provide a basis of comparison for future studies. More complex tasks might introduce factors which might complicate analysis of the results.

Our secondary task, maintaining power levels, is achieved by phototaxis — moving towards a light source — when charge is low. Other ways of achieving energy homeostasis could have been used: consider the sailing power management problem of Sauzé and Neal [243] described above. In order to recharge, their robot must simply use its actuators less. However, we elected to use a conventional land-based robot for convenience, with phototaxis as the recharging task. This is a more complex behaviour than simple down-regulation of the actuator change, chosen because it is of roughly the same complexity as the primary exploration task. Additionally, Sauzé and Neal use a waypoint following primary task: the network inputs are the rudder actuator position and heading error, and the output is the desired change in rudder actuator. We elected to use exploration as the primary task for its simplicity and parallels with phototaxis: in the former, we turn away from close obstacles; in the latter we turn towards the light.

Other tasks which do not involve energy homeostasis could be attempted: these include navigational safety and fault mitigation in a search and rescue setting [275], and temperature management [10, 87].

Thus the robot has two behaviours: "exploration", in which the robot moves in straight lines but turns away from obstacles; and "phototaxis", in which the robot moves towards the light source to recharge. At modulator $h = 0$ (see previous sections) the exploration behaviour is used, while at $h = 1$ the robot uses phototaxis. The modulator h is determined by

$$h = 1 - C \tag{8.1}$$

where C is the simulated battery charge, so the behaviour is selected by the level of charge[3]. This charge is reduced by both a base power usage value and some factor of the added motor outputs, and replenished by some factor of the light level given by summing the light sensors. The charge is in the range $[0, 1]$: it is clipped if it exceeds 1, and if it reaches zero the simulated battery is flat and the run ends prematurely.

In this regime, "good" behaviour is to explore the arena as much as possible while avoiding obstacles, returning to the power source when the charge is low. Once charge is replenished, the robot should return to exploration. Thus the behaviour is homeostatic, keeping the charge within a given range (although this range is somewhat wide).

8.1.1 The robot

The problem described above requires a wheeled or tracked robot, preferably with differential steering for ease of control. It also requires omnidirectional[4] detection of a light source (for phototaxis) and obstacle detection (for exploration). Many robots have obstacle detection with infra-red, laser or sonar range finding, but light source detection is rarer.

An ActivMedia Pioneer 2-DX [2] fitted with an omnidirectional camera was available, and fitted the requirements well. This robot, shown in Fig. 8.1, has two drive wheels and a rear caster which allows it to be controlled by two outputs: one for each drive wheel. The robot is shown in the test arena in Fig. 8.2 for scale. One disadvantage of the Pioneer is its large size, which made running the experiments difficult and time-consuming. Another is the low CPU specification, which is insufficient to run the Robot Operating System (ROS). This required development of a

[3]The behaviour is this way round (rather than $h = 0$ selecting phototaxis and $h = 1$ selecting exploration, which would give $h = C$) to reflect the idea that $h = 0$ is the "normal" behaviour, with the modulator becoming non-zero as an exception (corresponding to a notional "hunger hormone").

[4]The experiments could be performed with a simple front-facing light sensor, but this would complicate the phototaxis behaviour.

"bridge" to interface between ROS on a separate machine and the Pioneer's native ARIA API (see Appendix A).

Figure 8.1: "Bart", the Pioneer 2-DX robot used in the experiments. Note the camera, which is pointing upwards into a parabolic mirror mounted in an acrylic tube.

Other robots could have been used — any wheeled robot with a simple control system, an omnidirectional light sensor or camera, and a good number of front-facing range finding sensors would have been suitable. For example, the experiments would have been considerably more convenient to run on the much smaller e-Puck or Khepera with an omnidirectional camera[5]. However, the large number of front-facing sonar sensors on the Pioneer robot (and its availability) gives it an advantage over the smaller robots.

The omnidirectional camera was used as an 8-direction array of light sensors by radial sampling through the image. This is described further in Sec. 10.1.2. There is a light source in the arena, which acts both as a power source for a simulated battery and a beacon to the power source. The light sensors also act as "charge receptors," with a function of the sum of their values being used to increment the simulated charge (see Sec. 9.1.4).

[5]although the Khepera KD2-360 camera appears to be discontinued, and the e-Puck omnidirectional camera extension would have required a considerable amount of manufacturing and development [96].

Figure 8.2: The robot inside the experimental arena, which is bounded with sonar-opaque mesh. The mesh in the left of the image was positioned slightly differently in the experiments as shown in Fig. 10.1, and was held taut by clips during the runs.

In addition to the omnidirectional camera, the robot also has 16 sonar sensors for range finding, divided into two arrays, of which the only front was used. The positions and angles of the front array relative to the centre of the robot are shown in Fig. 8.3.

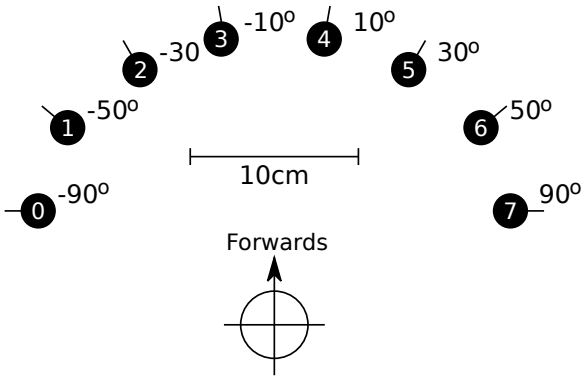


Figure 8.3: Sonar positions on both the simulated and real robot, after [2]. The centre of the robot's frame of reference is marked with a cross and circle. Each sensor is marked with its normal angle and (in the black circle) the sonar number.

The robot was simulated both in a simple simulator (used for generating training sets and initial testing) and in the Gazebo simulator [151]. Given that the robot has eight (active) sonar sensors and a light sensor with eight outputs as described above, this gives a total of 16 inputs to the neural network under test. There are two outputs from the network, one for each drive wheel.

8.1.2 Metrics

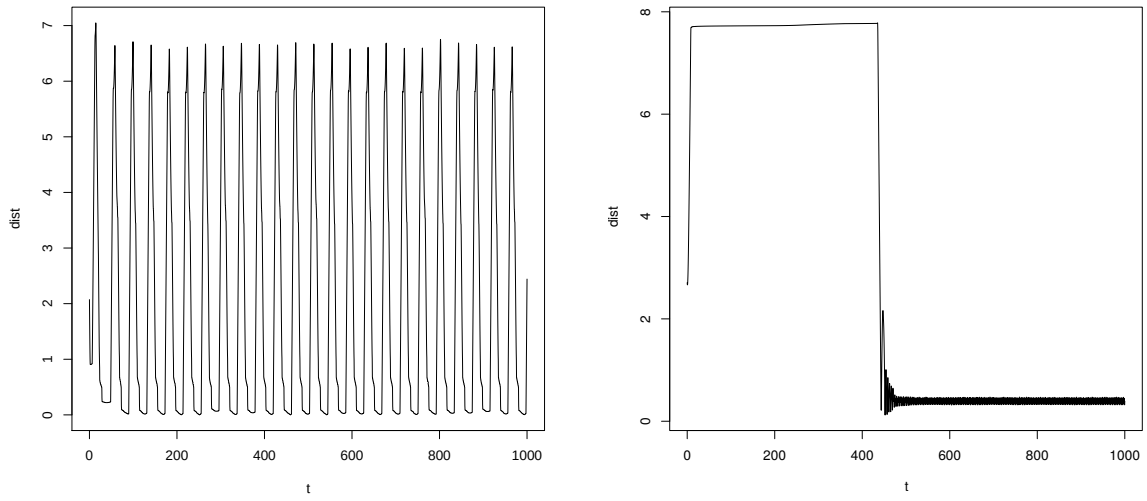
While typical runs of each network will be described in some detail, it would be useful to have a single metric to compare the performance of several networks against each other. This is particularly true in simple simulation runs: because these are inexpensive to perform, we will train several networks and simulate them, choosing which to carry forward into the Gazebo and robot simulations on the basis of this metric. In the Gazebo/robot experiments themselves, the metric will not be used: the behaviour of the real robot is best described qualitatively, particularly given the limited number of runs, and the Gazebo simulations are used as a basis for comparison.

A "good" network is one which survives for a long time (achieving homeostasis), and covers a large area of the test arena (exploration). Some work was done with occupancy grid metrics, but these were found to be deceptive in our test environment: firstly, a robot could achieve high occupancy by simply running itself at full speed in random directions, covering a large area (but not the arena edge) but not achieving homeostasis. Secondly, two similar networks could give different occupancy counts: one network could give a low count by producing a simple elliptical repeating "orbit", while the other could give a high count by rotating the path slightly on each orbit. This would tell us little about the underlying performance.

Disregarding (for now) the homeostatic survival aspect, we define a "good" network as one which moves the robot around the arena space, varying between near and far from the power source, while moving as far as possible from the power source to the edges of the arena. While this does not directly take into account the possibility that the robot might only occupy one segment of the arena, this is unlikely to result in a good score because the robot will not spend much time near the arena edge by so doing. Including the survival time, we can now construct three metrics.

8.1.2.1 Distance variation metric

Given that we can obtain a log of the robot's position and we know the position of the light, we can calculate the robot's distance from the light over time. We can then use a measure of variation in this distance — the population standard deviation was selected in these experiments. However, the plots in Fig. 8.4 reveal a problem with this approach. The second plot has a larger standard deviation than the first (correctly so) but shows no variation over the short term: there are large sections of the run in which the robot is not moving. Ideally, a useful measure of variation would show the distance varying throughout the run to demonstrate good exploratory behaviour.



(a) $\sigma = 2.292$ **(b)** $\sigma = 3.608$

Figure 8.4: Plots illustrating the problems with using the standard deviation as measure of distance variation (from simulated robot runs).

To deal with this, each run was divided into slices and the standard deviation found for each slice. The result is the mean of the standard deviations for all slices:

$$m_d = \frac{\sum_{i=1}^{n} \sigma\left(d_{(i-1)s<t\leq is}\right)}{n}, \quad s = t_{max}/n \tag{8.2}$$

where m_d is the metric, n is the number of slices, $d_{t_1<t\leq t_2}$ is the set of distances from the power source recorded between times t_1 and t_2, and t_{max} is the maximum time. It should be noted that despite the slicing operation, it is still possible that a run with a long unchanging distance could outperform a constantly changing distance if the slice boundaries coincide with behaviour changes or the number of slices is poorly chosen. For these experiments $n = 10$, which seems a good compromise between too many slices, which would result in very rapid moves to and from the power source being highly scored; and too few slices, which would result in the same problems seen in Fig. 8.4.

8.1.2.2 Edge-weighted distance traversed metric

The idea of "moving far from the power source" can be split into two components: "moving" can be measured by adding together the distances between successive positions in the log, and "far from the source" can be measured by using the distance

of each position from the source. These two measures are combined by multiplying the distance travelled in each log segment by the distance of the first point in that segment from the source, giving an "edge-weighted distance" measure:

$$m_T = \sum_{j}^{n-1} \left(\sqrt{(x_j - x_{j+1})^2 + (y_j - y_{j+1})^2} \sqrt{x_j^2 + y_j^2} \right) \tag{8.3}$$

where n is the number of points in the log and (x_j, y_j) is the position of point j relative to the power source.

8.1.2.3 Survival time metric

The most straightforward metric is the survival time

$$m_t = \max t \tag{8.4}$$

where t is the time of each point in the log: the experiment terminates when the robot's simulated battery charge reaches zero, or after a fixed time if this does not occur.

8.1.2.4 The combined metric

We now have three metrics measuring three different aspects of the robot/network performance, which we need to combine into a single metric. To achieve this, we use the method of Rodriguez and Weisbin [233] used by Tunstel [282] for calculating Mars Exploration Rover performance metrics. This method compares each metric against a reference value and calculates the ratio of the metric to this value. However, it expresses these ratios in information-theoretical terms as bits, by using the expression

$$\log_2 \left(\frac{m}{r} \right), \tag{8.5}$$

where m is the value of the metric in question and r is the reference value. Thus, if the metric performs twice as well as the reference the result is 1 bit, and if it performs half as well the result is -1 bit. These metrics are combined additively (that is, multiplicatively in the underlying ratios) thus:

$$m_c = \frac{1}{2} \sum_i \log_2 \left(\frac{m_i}{r_i} \right)^2. \tag{8.6}$$

In our system, this becomes

$$m_c = \frac{1}{2}\left(\log_2\left(\frac{m_d}{r_d}\right)^2 + \log_2\left(\frac{m_t}{r_t}\right)^2 + \log_2\left(\frac{m_T}{r_T}\right)^2\right). \tag{8.7}$$

In this metric, multiplication is used for the same reason that multiplication is used in combining the probabilities of independent events: we require a metric which shows how well the system performs at all tasks[233]. Of course, there is likely to be a complex interdependence between all the metrics in our case. Rodriguez and Weisbin [233] briefly discuss possibilities for dealing with such dependencies, but we shall ignore them for now — we need a simple metric with which we can compare performances taking into account as many requirements as possible. Calculating the dependencies between the metrics would be a very complex task.

This combination method has the advantage of converting metrics with different units and scales to dimensionless values (ratios), while using the binary bit expresses the result in a convenient and well-understood way.

8.1.3 Generating examples and training

UESMANN and the other network types use supervised learning paradigms: they are trained from examples of correct behaviour. As noted in the introduction to this chapter, this means that we are not learning what the sub-behaviours (phototaxis and exploration) are, but are instead constructing a single system incorporating both behaviours which will switch between them. To generate the large numbers of training examples required, the simple simulator (see Sec. 9.1) was used to run two rule-based controllers, switching between them periodically to change the behaviour. The inputs, outputs and modulator level (0 or 1) for the current behaviour were recorded and provide the examples for network training. More details on the training regime are given in Sec. 9.2. Once trained, the networks were carried forwards into the experiments.

8.1.4 The experiments

Experiments were performed using two different simulators and an actual robot. For some initial experiments the same simple simulator as that used in generating training examples was used, which has a very basic differential steering model with perfect sensors and actuators, and simulates robots in a simple enclosed square arena.

Multiple networks with different initial weights and biases were trained for each network type using the same power management constants (rate of charge, motor power expenditure, etc.) to provide a basis for comparison. While many experiments could have been performed with this simulator at different power expenditure and charge rates, it was decided to evaluate the networks using a single set of these parameters. This would allow us to see how well the networks behaved when these parameters were varied from in the real robot.

The best networks (i.e. those with the highest combined metric) of each type were carried forwards into a set of experiments using the Gazebo simulator, which has perfect sensors[6] but a more complex and realistic physics simulation. These experiments used a more accurate model of the final arena. The same networks were carried over onto the real robot and the runs compared with their Gazebo counterparts: it was predicted that the physical and sensor responses of the robot and its simulated Gazebo counterpart would differ in important ways, leading to differences in behaviour.

Studying the differences between the noise-free simulations and the real robot might provide useful insights into how well each network architecture is able to bridge the "reality gap" (see Sec. 2.4, p. 41): the difference between simulation and physical robot, which presents the control system with noisy, inaccurate data and imperfect actuators with complex responses.

In these experiments the performance of the networks was compared between Gazebo and the actual robot, using two different initial poses and two different power expenditure settings. Only one Gazebo run was made for each pose/setting combination in each network, since the Gazebo performance was deterministic given the small amount of sensor noise added: all Gazebo runs with the same parameters would be the same. Several runs were made for each set of robot settings.

To summarise:

- Ten networks are trained for each network type, each with different training data generated by the simple simulator with random course and position changes.
- These are tested in the same simulator (without the random changes), and the best network of each type is carried forward.
- These networks are evaluated in both Gazebo and on the real robot.

[6] A very small amount of noise was added to the sonar sensors (see below). No noise was added to the simulated light sensor, although Gaussian blurring was performed.

Chapter 9

Training and the simple simulator

9.1 The simple simulator

This simulator was developed primarily to provide a large set of training examples. It was was deliberately very simple, both to execute quickly and and to generate "low-fidelity" training examples which (it was hoped) would nevertheless be able to function in a real robot, crossing the "reality gap." Overfitting the networks to a complex simulation which did not resemble reality in important ways was deemed more of a risk than overfitting to a naïve simulator.

9.1.1 Kinematics

The kinematics of the simple simulator is described by Eqs. 9.1 and 9.2 which implement a simple differential steer system, giving a linear relationship between requested motor speeds and speed along the ground [178]:

$$\frac{d\theta}{dt} = \frac{s_l - s_r}{b} \tag{9.1}$$

$$\frac{d\vec{p}}{dt} = \frac{s_l + s_r}{2} \begin{pmatrix} \cos(-\theta) \\ \sin(-\theta) \end{pmatrix} \tag{9.2}$$

where

- $\vec{p}$ is the position,
- θ is the orientation,
- s_l and s_r are the speeds of the two wheels (i.e. their speeds along the ground),
- b is the distance between the two wheels.

9.1.2 Sonar

The sonar sensors have simulated positions and angles equivalent to those on the real robot, as shown in Fig. 8.3. In this simulation (and also in Gazebo) the sensor returns a distance by ray-casting along its normal, finding the nearest obstacle (linear object in the world). Because the training and test area are enclosed, an obstacle is always found. The eight return distances form the first part of the input to the network (and rule system, when this is used).

9.1.3 Light sensor

The omnidirectional light sensor is simulated by using linear elements in the world which act as notional emitters. The sensor traverses the world objects and calculates the angles of the ends of each emitter as viewed from its frame of reference. These angles are converted into indices into a buffer of 16 luminance values. Pixels in the buffer which encompass the emitter angles are set to the emitter colour, while the others are set to zero. This buffer is then blurred with a Gaussian kernel, with the blurring algorithm processing the pixels *modulo* 16. The resulting buffer is added to the global buffer, which is set to zero at the start of processing. This is described in Algorithm 8, and shown in Fig. 9.1. While the sensor is linear, it was found during initial testing that blurring in conjunction with using the floor and ceiling operators effectively widened the pixel coverage in the luminance buffer enough that a circular emitter was not required[1].

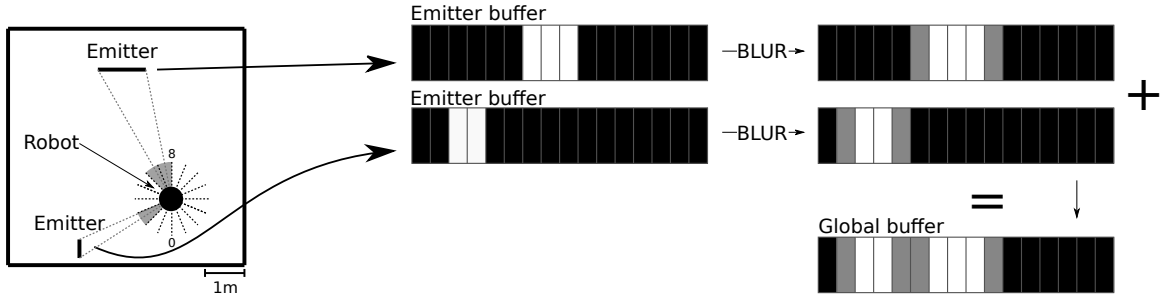


Figure 9.1: Operation of the simulated light sensor, showing how emitter end-point angles are converted into buffer pixels, which are blurred and summed. In the left-hand view of the arena, the robot is shown as a black dot with a dotted black line for each sensor ray (showing ray indices 0 and 8). The emitters are shown as solid thick lines. The angles to the emitter end-points are shown as dotted grey lines. The operations of the floor and ceiling operators to produce ray indices are shown as grey arcs. The figures to the right show the generated pixel buffers.

[1] In retrospect, a circular emitter would have been easier to implement!

Algorithm 8 Light sensor algorithm for the simple simulator

$\vec{b} \leftarrow$ buffer of 16 real light values, initialised to zero
for $e \in Emitters$ **do**
  $\vec{t} \leftarrow$ buffer of 16 real light values, initialised to zero
  $\theta_1 \leftarrow$ angle of emitter e start in robot frame
  $\theta_2 \leftarrow$ angle of emitter e end in robot frame
  (Ensure $\theta_1 < \theta_2$, and that θ_1 to θ_2 covers the emitter extent.
  This may result in either being greater than 2π.)
  $p_1 = \left\lfloor \frac{16}{2\pi}\theta_1 \right\rfloor \bmod 16$ {Convert to 0-15 using floor operator}
  $p_2 = \left\lceil \frac{16}{2\pi}\theta_2 \right\rceil \bmod 16$ {Convert to 0-15 using ceiling operator}
  for $i = p_1$ **to** p_2 **do**
    $t_i \leftarrow luminance$ {Add the emitter's contribution to the luminance buffer}
  end for
  $\vec{t} \leftarrow \vec{t}$ blurred with a Gaussian kernel of size 3, with $\sigma = 1$
  $\vec{b} \leftarrow \vec{b} + \vec{t}$
end for

At each iteration, the sum of the luminance buffer values $\sum b_i$ is used to increment the simulated battery charge as described below. The buffer is then downsampled by selecting every other value, and the resulting 8 values provide the second part of the input to the networks. Note that in the simulation, emitters do not act as obstacles, because the real emitter is not an obstacle.

9.1.4 Power

The power simulation described here is used by all the experiments, including the Gazebo and robot experiments, but is described here as part of the initial simple simulator experiments. The robot's battery has a charge in the interval [0,1]. We assume that the power usage is proportional to the commanded motor speed, plus some base usage. If the charge is C, the commanded speeds are s_l and s_r, the power input is p, the time step is Δ_t, the base usage is k_{base} and the motor power factor is k_m, then

$$C_t = \text{clamp}\left(C_{t-1} + \Delta_t\left(p - (k_{base} + k_m(s_l + s_r))\right)\right) \tag{9.3}$$

$$\text{where } \text{clamp}(x) = \max\left(0, \min(1, x)\right). \tag{9.4}$$

That is, the charge increases linearly with the power input, and decreases linearly with the sum of the motor power outputs and base power usage, with multiplicative constant factors determining by how much. A more realistic model might use a saturation model rather than hard clipping at full charge, such as making the power

added proportional to the difference between the maximum and the current power, but that was deemed an unnecessary complication at this stage.

The power input p is obtained from the sum of the input pixel values run through a simple low-pass filter (using Brown's simple exponential smoothing, an exponential weighted moving average[39]):

$$p_0 = 0 \tag{9.5}$$

$$p_t = k_{L\alpha} k_{power} \sum_i b_i + (1 - k_{L\alpha}) p_{t-1}, \tag{9.6}$$

where $\vec{b}$ is the light sensor output (see Algorithm 8), k_{power} is the light power factor determining how light is converted to power, and $k_{L\alpha}$ is the smoothing constant (set to 0.99).

9.2 Training

As noted above, all networks are trained using examples generated by the simple simulator (described above) using a virtual robot with two rule-based controllers, which we shall refer to as *exploration* and *phototaxis*. While each controller has access to all inputs, in practice *exploration* only uses the sonars and *phototaxis* only uses the light sensor. Italics will be used throughout to emphasise that these are labels for the behaviours described by the controllers rather than the English terms: for example, the *phototaxis* behaviour consists of what is commonly meant by "positive phototaxis" — moving towards the light — but also includes a stop behaviour when the light is close, to permit recharging without using the motors (which drain charge).

Training is done as follows for for each individual network:

- The simple simulator, described in Sec. 9.1 (p. 241), is run for 200000 simulator ticks in the training arena[2], which is rather different from the simple test arena[3], with internal walls to ensure a wide variety of training examples. It is also different from the final arena[4], which has more corners, each of which is less acute. This was partly due to operational difficulties, but also would expose any networks which overtrained on the examples such that they would only operate in an arena of similar geometry to the training arena.

[2]Fig. 9.2 (p. 247)
[3]Fig. 9.9 (p. 255)
[4]Fig. 10.1 (p. 274)

- In each tick, either the *phototaxis* or *exploration* rule-based controller runs. The sensor inputs and motor outputs are logged as training examples, with a modulator value: $h = 0$ if the controller used is *exploration* and $h = 1$ if *phototaxis* is used. These controllers are described in Sec. 9.2.1, and the process for generating the examples is given in more detail in Sec. 9.2.3.
- The simulator switches between the two controllers every 1000 ticks to ensure there are an equal number of training examples for each.
- The simulator has a small chance of randomly changing course or position each tick, to ensure loops are avoided.
- The network is then trained using the logged controller output described above, using the parameters given in Sec. 8.1.3 (p. 239).

Note that each individual network of the ten trained for each network type has a different set of training data, because the random position and course changes will be different in each run of the simulator. This random behaviour is disabled during the experimental runs of Sec. 9.4.

9.2.1 The rule-based controllers

The *exploration* controller is shown in Algorithm 9: if any sonar returns a distance less than 0.5m, find the smallest sonar distance on the left and right sides, and turn right if the left side value is smaller and right otherwise. The two front sonars are ignored for the purposes of determining turn direction, but are taken into account for the initial obstacle detection.

Algorithm 9 *Exploration* controller: d_n are front sonar bank distances, indexed 0-7 from left to right, $\bar{s}$ is a tuple of motor outputs $(left, right)$. See Fig. 8.3 for the sonar numbers.

if $\min d_{0\ldots7} < 0.5$ **then**
  if $\min(d_0, d_1, d_2) < \min(d_5, d_6, d_7)$ **then**
    $\bar{s} \leftarrow (1, -1)$
  else
    $\bar{s} \leftarrow (-1, 1)$
  end if
else
  $\bar{s} \leftarrow (1, 1)$
end if

The *phototaxis* controller is shown in Algorithm 10. Here, the pixels are processed to produce a value v indicating how far left or right the robot should turn: negative

for left and positive for right. This is scaled by k_{Ls} (which will depend on the number of pixels in the image) to produce the turn value. This value is added to 1 for the left motor and subtracted from 1 for the right motor to give the basic speed. The total number of pixels illuminated by the emitter is also taken into account so that the robot will slow down when it is close. This speed factor is a linear ramp from 1 when the total input light is K_{Lmax} to 0 when it is K_{Lmin}. To produce the motor speeds, the basic speeds are multiplied by this factor, and clamped to [0,1]. These motor speeds are then multiplied by k_{Lspeed} to give the values sent to the simulation.

Algorithm 10 *Phototaxis* controller: $\bar{l}$ is a vector of input pixels, v is the "direction" in which to move, t is the total light, $s_{l,r}$ is a tuple of motor outputs $(left, right)$. The $\text{clamp}_{[a,b]}$ operator clamps the value to $[a, b]$. Constants: k_{Lmax} is the number of fully illuminated pixels which will cause the motors to stop (by setting $d = 0$; d ramps down from 1 at no light to 0 at k_{Lmax}). k_{Ls} is the strength of the steering – higher, the turns will be tighter. k_{Lspeed} is a final speed multiplier to counteract the slowdown: note that the actual motor speeds are still subject to [-1,1] clamping.

$v, t \leftarrow 0$
$C = N/2$ {Centre pixel, as float}
for $i = 0$ **to** $N - 1$ **do**
  $p \leftarrow \big((i + C) \mod N\big) - C$ {p is pixel number with 0 as centre}
  $v \leftarrow v + pl_i$ {add to steer value}
  $t \leftarrow t + l_i$ {add to total light value}
end for
$d = \text{clamp}_{[0,1]}\left(1 - \frac{t - k_{Lmin}}{k_{Lmax} - k_{Lmin}}\right)$ {get max speed}
$s_l \leftarrow k_{Lspeed}\,\text{clamp}_{[-1,1]}\big((1 + k_{Ls}v)d\big)$
$s_r \leftarrow k_{Lspeed}\,\text{clamp}_{[-1,1]}\big((1 - k_{Ls}v)d\big)$

9.2.2 The training arena

Fig. 9.2 shows the simulated arena used for training. There are two internal walls to provide situations where the robot has obstacles on both sides, which have caps to help provide an obstacle when the robot has an "end-on" view of a wall. The light source here is a linear emitter in one corner. In *exploration*, the robot will avoid all the walls (including the internal walls) using the sonars. In *phototaxis* the robot will move towards the emitter using the light sensor, ignoring all walls — it will not leave the arena because both robot and emitter are contained within the square outer wall, but it will pass through the internal walls. The purpose of the internal walls is to provide more variety in the sonar training data.

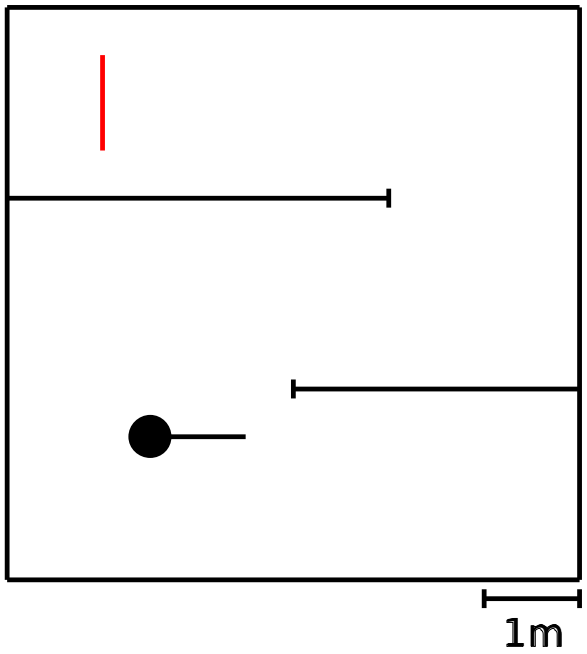


Figure 9.2: The arena used with the simple simulator to generate training examples using the two rule-based controllers. The red line is light-emitting and not an obstacle, and the initial position of the robot is indicated with a circle of the radius of the robot body, with a line showing the initial orientation.

9.2.3 Generating training examples

Examples are generated using a special version of the simple simulator, a screenshot of which is shown in Fig. 9.3. This shows the arena and emitter, the robot and sonar rays, and the light sensors in a ring around the centre – both the full set and the downsampled eight inputs sent to the networks. Various monitoring values are also shown. This program is typically used without a visual display, and halts after a fixed number of ticks have been output (200000 in our experiments). As stated above, it runs either the *exploration* or *phototaxis* controllers, switching between the two every 1000 ticks.

If permitted to run normally the simulator will fall into a simple looping course. This will not provide a varied enough set of examples for training; therefore a degree of randomness is incorporated externally into the simulation during example generation:

- Every tick there is a 1% chance that the robot will turn by a random value in the interval [-0.5,0.5] radians (28.6°).
- Every tick there is a 0.143% chance (1 in 700) that the robot will turn 180°.

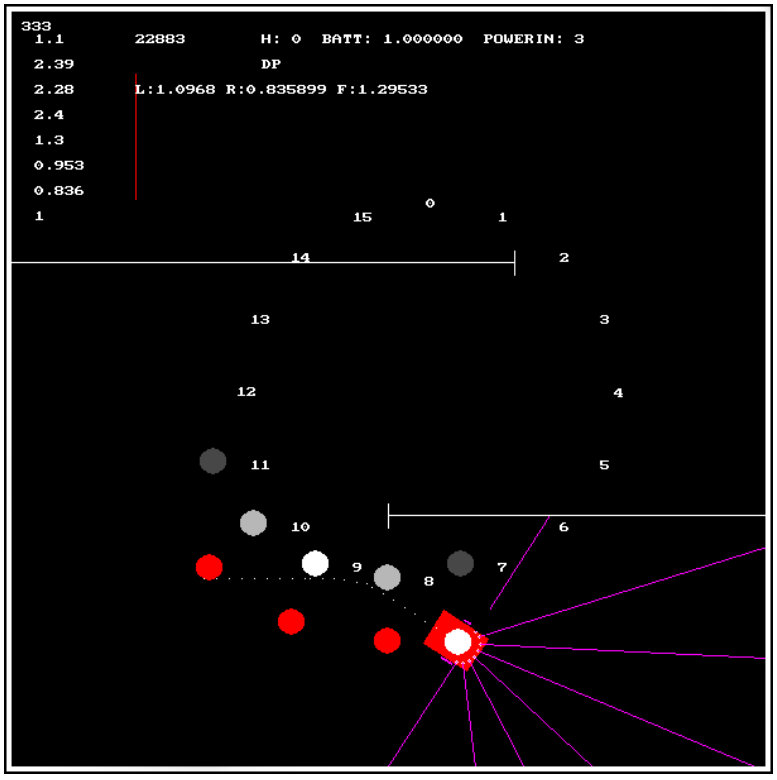


Figure 9.3: A screenshot of the `recordctor` program, which generates training examples. The partial circle of white dots shows the 16 "raw" light sensor pixels in a robot-centric visualisation, the three red dots outside show the corresponding decimated pixels for feeding into the network.

- Every tick there is a 0.143% chance that the robot will be "teleported" to a random location in the arena.

9.2.4 Training and network hyperparameters

Training was performed in a similar way to previous experiments, with the proviso that a fresh set of training data was generated for each attempt in addition to starting with a different set of initial weights and biases[5]. A set of 10000 examples of validation data was also generated: this amount was found to be sufficient to provide examples of most situations for validation purposes. As before, all examples were shuffled before each iteration through the training set.

In these experiments, 15 hidden nodes were used: this is a number between the number of inputs and the number of outputs, but at the high end of the range, and was chosen to provide a reasonable number of "feature detectors." The learning rate η was set to 0.1, based on earlier experiments and some informal experimentation

[5]This was done in order to provide more variety given the limited number of networks, but may introduce problems: see Sec. 11.5 (p. 313)

at this hidden node count. The number of pair presentations was 3×10^7 giving 150 iterations through the training set of 200000 examples: this high count was intended to provide enough time for networks to converge, and was again based on informal experimentation. The initial weights were selected using Bishop's rule, as in previous experiments.

9.3 Convergence behaviour

For each network type (output blending, h-as-input and UESMANN), 10 networks were trained using random initial weights/biases and a different training set generated using the simple simulator, using the parameters given in the previous paragraph. All networks were expected to converge, given the large number of pair presentations, and overfitting was considered unlikely given the number of examples and therefore the number of iterations through the sets. To give an idea of the final performance in terms of mean squared error, box plots of the the mean of the MSE of the outputs of all networks for the last 10 validation passes are shown in Fig. 9.4.

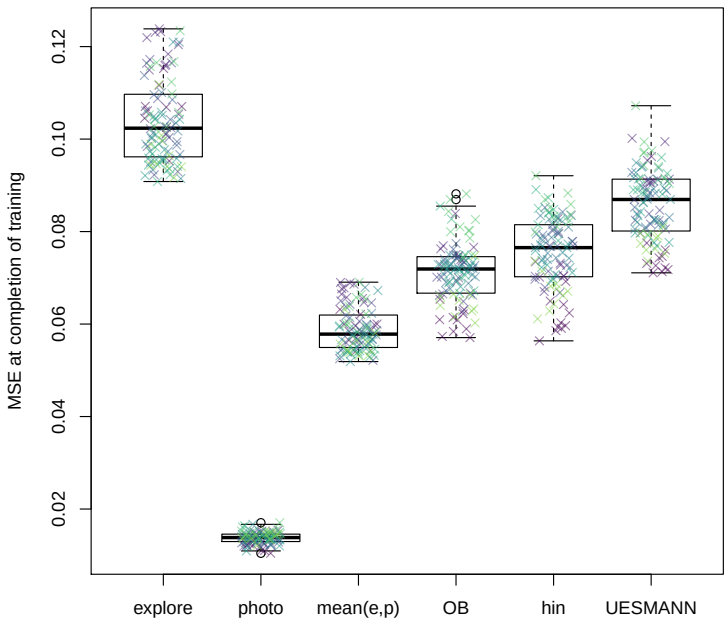
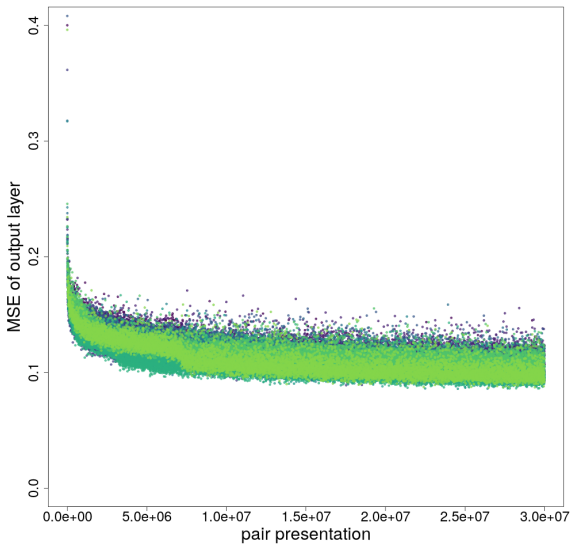


Figure 9.4: The means of the MSEs of all networks of each type over the last 10 validation passes during training, giving an indication of the converged performance. The plots are overlaid with the points, with those from the same network given the same colour. There are ten points from each of the ten networks. Also shown are the means of the *exploration* and *phototaxis* networks.

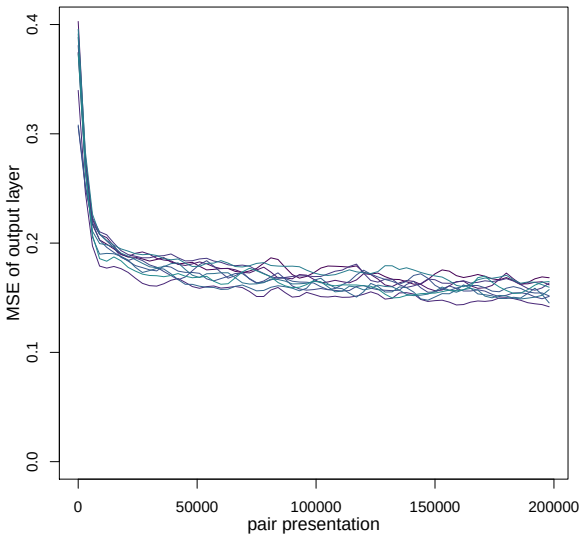
9.3.1 Plain back-propagation

Networks were also trained with plain back-propagation on examples of just the *exploration* and *phototaxis* controllers: these are shown in Fig. 9.5 both as points over the entire training run and splines fitted to the first portion. These show that the networks do converge, and there is no subsequent rise in error as might be seen with overfitting. The noise seen in both plots for each network is likely to originate in the validation process: each validation is on a difference slice (of 10) of the test set.

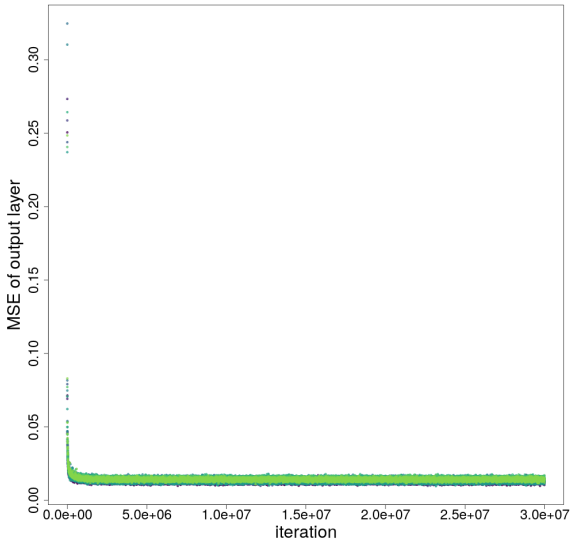
It is clear that the *exploration* behaviour is considerably more difficult to learn, with a much slower convergence and higher final error. The *phototaxis* networks converge to solutions within 200000 iterations. This is as expected: learning to avoid walls with multiple sonars is a much more complex task than simply heading towards the light and then stopping, which can be achieved with a simple Braitenberg vehicle.



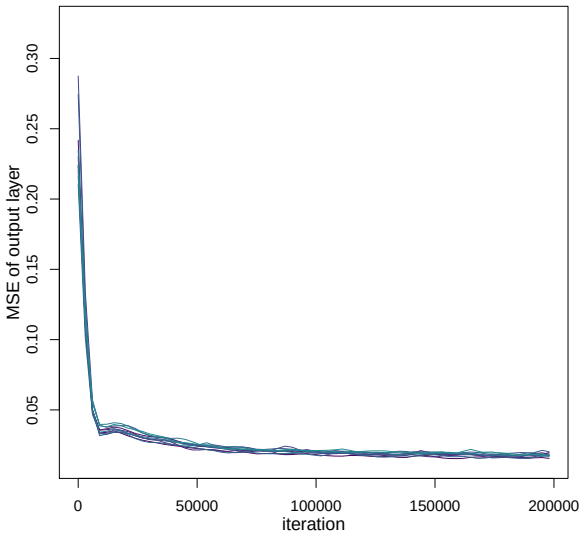
(a) *exploration*, all



(b) *exploration*, partial and smoothed



(c) *phototaxis*, all



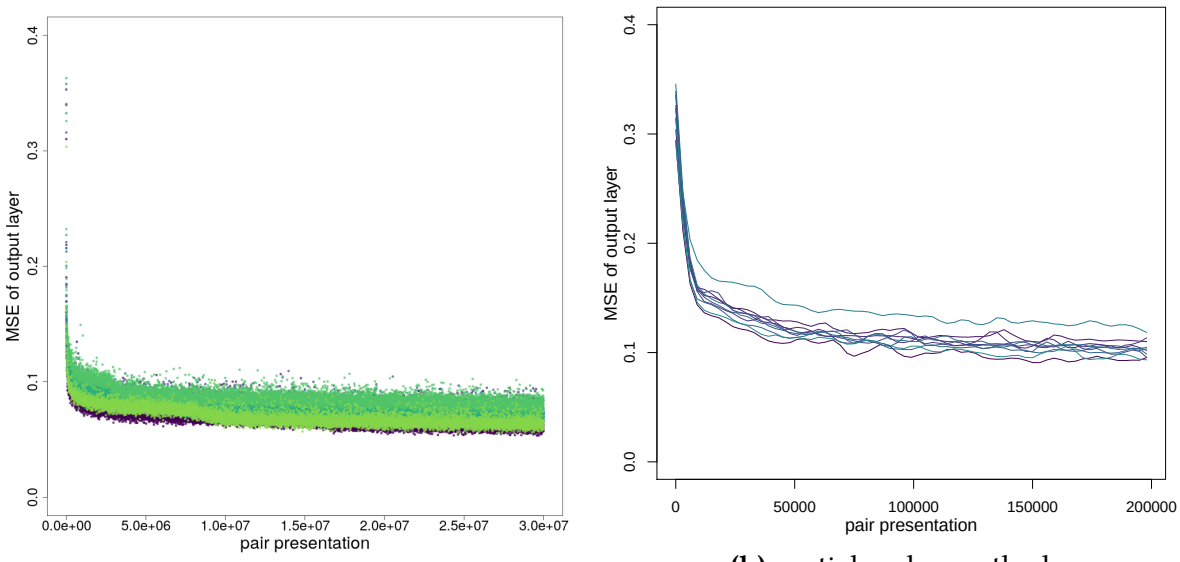
(d) *phototaxis*, partial and smoothed

Figure 9.5: Convergence behaviour for plain back-propagation trained on *exploration* and *phototaxis* data sets. To obtain the error values, slices of a separately generated training set were run through the network and the mean of the mean squared errors at the output found. The point plot shows the different networks in different colours, while the line plot fits a polynomial spline (using R's `pspline` package) to each network's points and shows only the beginning of training.

9.3.2 Output blending

The performance of output blending should be halfway between that of the two separate networks in the previous section, since we are effectively training two

such networks. Given that the final MSEs of *exploration*-only and *phototaxis*-only are approximately 0.1 and 0.01 respectively, we would expect a final performance of about 0.055. The mean is shown in the box plot in Fig. 9.4, and should represent the performances of output blending networks formed from the networks generated for each sub-task. The errors we see for the actual output blending networks are significantly higher than 0.055 ($p < 0.001$, Wilcoxon rank-sum test) for reasons which are not fully understood. It may be that this test of significance is invalid because there are only ten networks, and we are including ten values for each network (the ten final validation passes) in the comparison. Thus the values for each network are not independent of each other. If we were to assume only ten samples (because there are ten independent networks), then the threshold for a significant difference would be much higher.



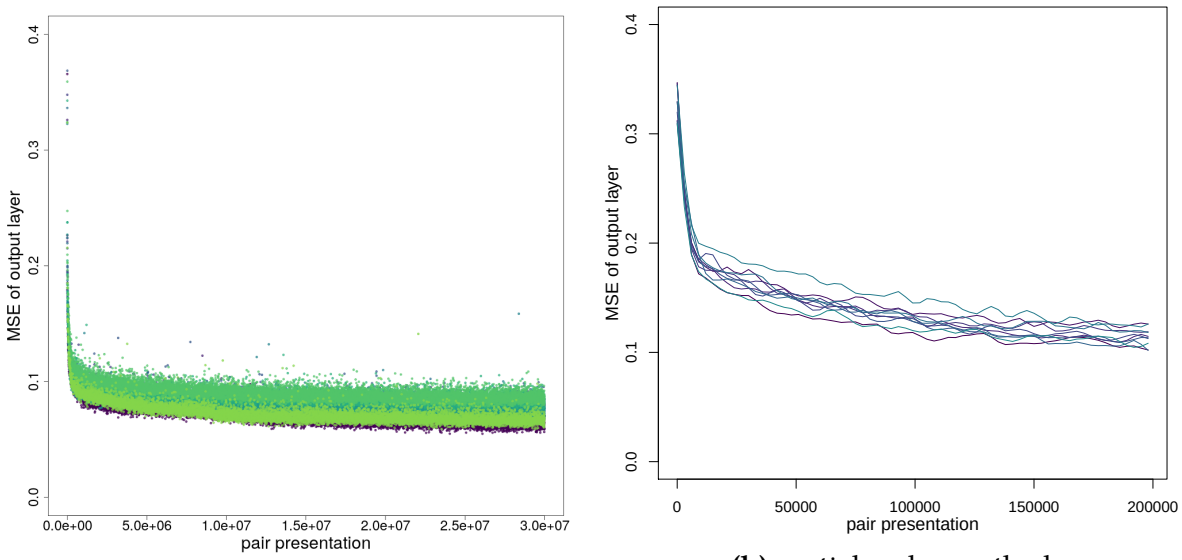
(a) all

(b) partial and smoothed

Figure 9.6: Convergence behaviour for output blending *exploration→phototaxis*. See Fig. 9.5 for details.

9.3.3 *h*-as-input

Fig. 9.7 shows that this network trains more slowly than output blending, and achieves a slightly worse final performance. This is as we would expect: we are now training a single network to perform two tasks, rather than two separate networks.



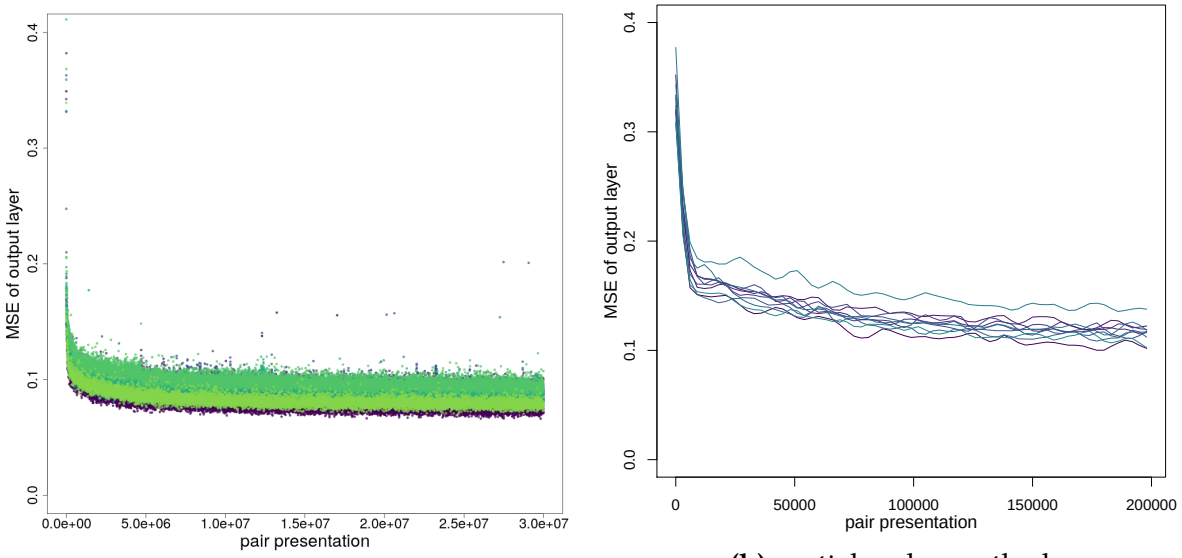
(a) all

(b) partial and smoothed

Figure 9.7: Convergence behaviour for *h*-as-input *exploration→phototaxis*. See Fig. 9.5 for details.

9.3.4 UESMANN

As shown in Fig. 9.8, UESMANN converges faster than *h*-as-input initially (although not as fast as output blending), but converges to a higher mean error. There is still some overlap: some UESMANN networks outperform some *h*-as-input networks. Note, however, the "spikes" in the convergence plots in Fig. 9.8, even after a large number of training iterations: UESMANN has a larger error on certain slices of the validation set, although the MSE here is still only ~ 0.1.



(a) all

(b) partial and smoothed

Figure 9.8: Convergence behaviour for UESMANN *exploration→phototaxis*. See Fig. 9.5 for details.

9.3.5 Discussion

All three network types converge to solutions, with the solutions for output blending better than those for h-as-input and those for h-as-input better than those for UESMANN. However, the differences between the final solutions are small with all networks giving a final MSE between 0.05 and 0.1. This is better than the final MSE on networks trained on *exploration* only, which achieve a final MSE of 0.1: this is clearly a more difficult task to learn than the *phototaxis* task. We will now look at how the final trained networks perform in the simple simulator.

9.4 Simple simulator experiments

In this first set of experiments, the simple simulator was used to run the best networks (in terms of final MSE) trained in the previous section. The runs were analysed qualitatively from the point of view of the paths traversed both in physical space and phase space (distance against charge), and also quantitatively using the metric developed in Sec. 8.1.2.4 above. We will later compare these results with those from both Gazebo simulation and the real robot, to see how the networks cope with the "reality gap."

Because the simple simulator does not incorporate any form of path finding or collision detection, the internal walls shown in Fig. 9.2 were removed. The light source was moved to the centre, and the arena was made much larger to give the robot enough space to move away from the light to lose significant charge: while the training arena is a 6m × 6m square, the test arena is four times larger at 12m × 12m. This new test arena is shown in Fig. 9.9. This arena also differs from the Gazebo and final physical robot arena, which is an irregular shape.

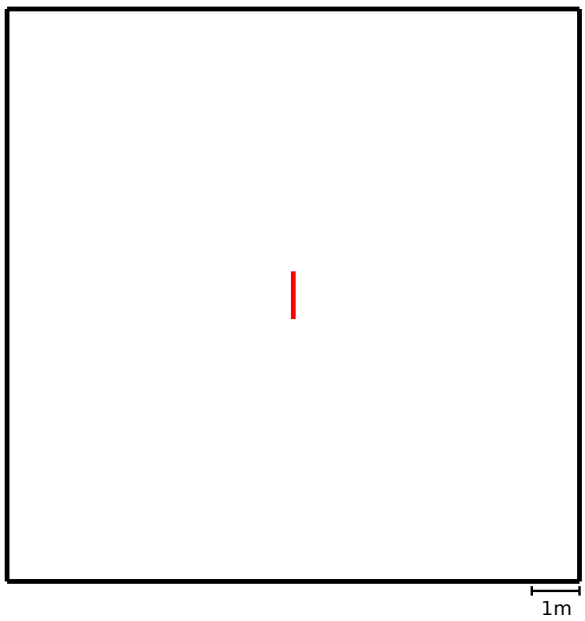


Figure 9.9: The arena used with the simple simulator to test the behaviour of the networks. The red line is light-emitting and not an obstacle.

In these tests, the power model described in Sec. 9.1.4 is active and feeds into the network modulator with $h = 1 - C$ where C is the charge (i.e. Eq. 8.1). The simulation is run for 1000 simulated seconds or until the charge reaches zero, whichever happens first. In each run the robot is started from a random position near the centre ($x, y \in [-2, 2]$) with a random orientation. The constants for these experiments are given in Table 9.1.

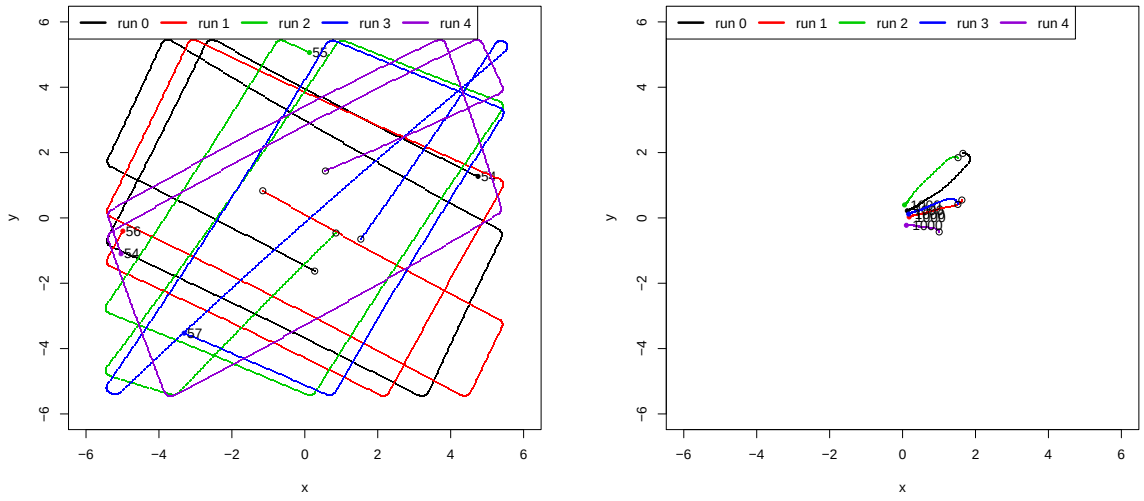
9.4.1 Control experiments: *exploration* and *phototaxis* only

To determine how well *exploration* and *phototaxis* were learned in the plain back-propagation experiments, the networks with the lowest final MSE were taken and run five times from random initial poses. Position plots for the results are shown in Fig. 9.10. The *exploration* runs avoid the borders of the arena, but run out of charge after less than a minute of simulated time. The *phototaxis* runs move towards the light and then stop, surviving the full 1000 seconds by remaining close to the

Table 9.1: Constants used in the simple simulator

constant	description	value
b	wheelbase	0.35m
Δ_t	time step	0.01s
k_m	motor power usage	0.01
k_{base}	base power usage	0.005
k_{power}	light power factor	0.0025
k_{Ls}	light steer factor	0.2
$k_{L\alpha}$	light smoothing factor	0.99
k_{Lmin}	light slowdown brightness	5
k_{Lmax}	light stop brightness	7
k_{Lspeed}	light speed factor	1

power source. Thus the two controllers have been successfully learned by the plain back-propagation networks, and the power simulation works as expected: exploring without regard to charge maintenance leads to a rapid demise.



(a) *exploration* **(b)** *phototaxis*

Figure 9.10: Paths of the best networks trained on *exploration* and *phototaxis* showing different runs in different colours. The start points are marked with an open circle, the end points with a dot and the time the run ended, either due to charge depletion or reaching the maximum time of 1000s (simulated).

9.4.2 Modulatory network results

This section will present the results for the three modulatory network types in the simple simulator, firstly using the combined metric (which will be used to select networks for the robot and Gazebo experiments) and then showing selected runs in more detail.

9.4.2.1 Comparison using the combined metric

Fig. 9.11 shows the values of all metrics for all runs, and box plots combining those results for each network type. The control runs of the previous section are not included here, because those runs are not attempting to achieve homeostasis while exploring, which is what the metric is designed to measure.

In these runs, the performance of output blending is consistent: all runs of all networks achieve roughly the same performance. While all runs achieve homeostasis, surviving the full duration, the amount of distance variation and the edge-weighted travel metric are all low. This indicates that these networks probably circle around the light/power source without much exploration.

The h-as-input experiments show a significantly higher combined metric than output blending, but a good deal of internal variation in performance both between different networks and within individual networks. Consider network 1, which always completes the runs but has a wide variation in edge-weighted travel and distance variation.

There is no significant difference between the mean performances of UESMANN and h-as input in any metric, although UESMANN is more consistent, having a lower variance in most metrics. In order to compare the performance of the networks in the simple simulator, it is necessary to examine individual runs of the networks qualitatively.

9.4.2.2 Output blending results

In Fig. 9.11, all output blending networks have near-identical levels of performance. A pair of typical runs of the best network are shown in Fig. 9.12, with phase (distance/charge) and variable plots for run 0 shown in Fig. 9.13. After an initial long excursion, the robot orbits the emitter in a series of tighter loops until the loops stabilise at a maximum distance of around 0.8m. These loops have a figure-of-eight appearance oriented to the linear emitter's axis, which leads to variation in the power input: when the robot is on the emitter axis (i.e. $x \approx 0$) it receives less power than when it is not aligned, notwithstanding the blurring in the light sensor simulator.

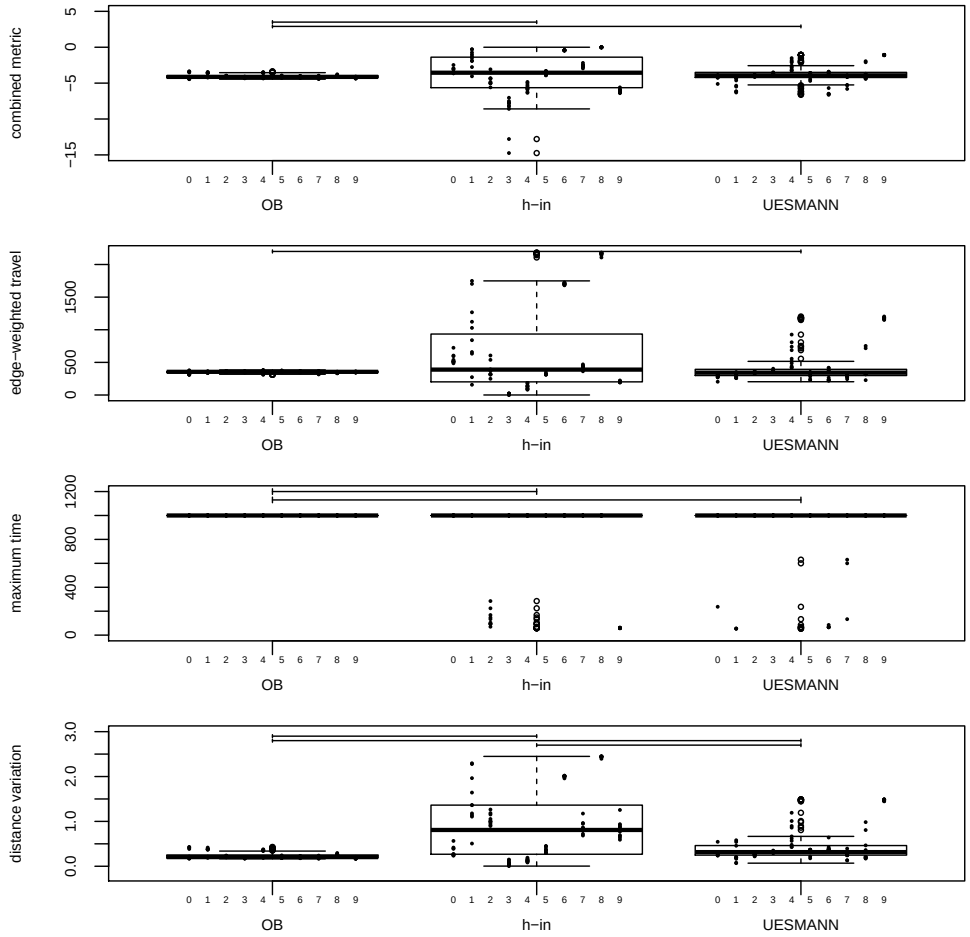
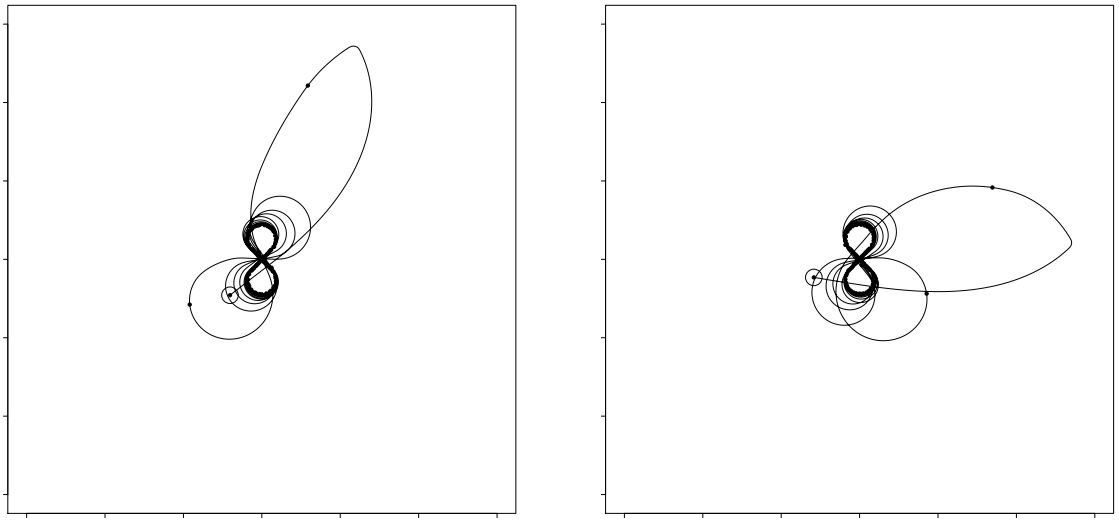


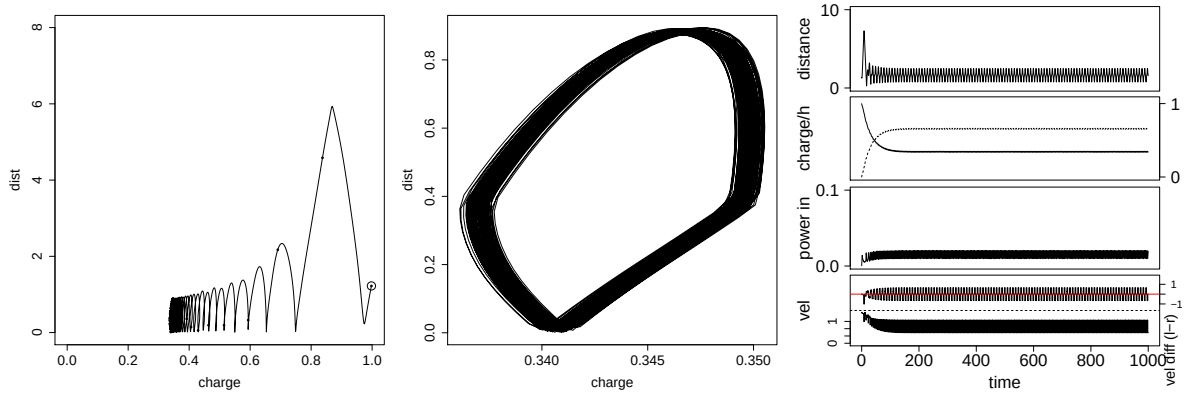
Figure 9.11: The values of the different metrics and the combined metric for all runs in the simple simulator. The box plots are for all networks of each type, while individual dots show the performances of each individual network within the type. The axis labels $[0, 9]$ indicate the network numbers, with each column of dots representing the experimental runs for each network. Near the top of the plot are brackets: these link network types whose means are significantly different (Wilcoxon rank-sum test, $p < 0.05$).

The charge reaches a minimum of 0.34, and the robot never stops moving. The maximum distance from the emitter centre is 5.93m on the initial excursion, but the mean is 0.52m. Although the robot achieves homeostasis, it does so by staying close to the emitter. Because it does not stop (which would lower the power consumption), it does not build up sufficient charge to move far.



(a) Run 0 **(b)** Run 4

Figure 9.12: Two position plots for different runs of the best output blending network according to the combined metric. The start position is shown by a circle, while small dots are equal times apart.



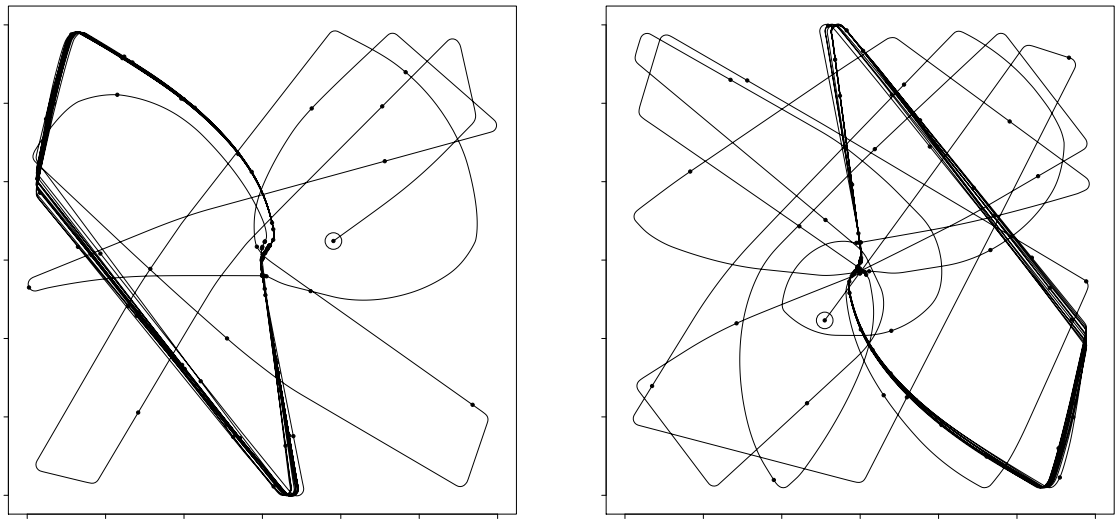
(a) Phase **(b)** Phase detail, $t > 500$ **(c)** Variables

Figure 9.13: Phase and variable/time plots for run 0 of the best output blending network according to the combined metric. In the phase plot, the dots are spaced at equal times and the start position is shown by a larger dot. In the variable plots, the upper part of the velocity plot shows left/right motor difference and the lower part shows the combined motor velocity magnitude; a dotted line separates the two. The dotted line in the charge/h plot is the modulator h.

9.4.2.3 h-as-input results

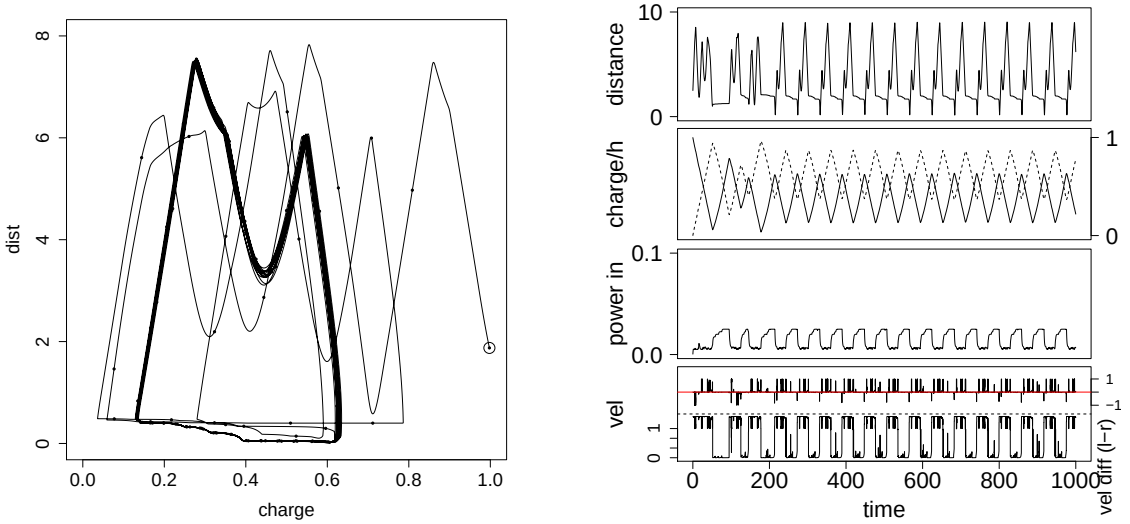
Position plots for two runs of the best network (number 8) are shown in Fig. 9.14, with the corresponding phase and variables plot for one of these runs in Fig. 9.15 (the other is similar). From the position plot, it is clear that the robot is rather more

successful in exploring the arena, although the arena's regular nature and the lack of noise in the simulation leads to it falling into loops. Run 2 appears more successful, starting at an angle which leads it to explore more of the arena before looping.



(a) Run 0 **(b)** Run 2

Figure 9.14: Two position plots for different runs of the best h-as-input network according to the combined metric. The start position is shown by a circle.



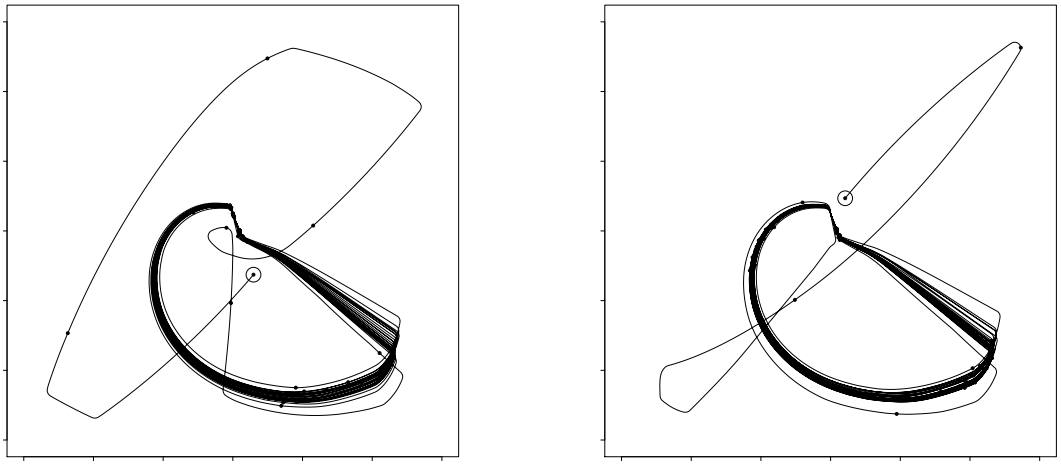
(a) Phase (run 0) **(b)** Variables (run 0)

Figure 9.15: Phase and variable/time plots for run 0 of the best h-as-input network according to the combined metric.

Comparing the velocity plot with the modulator, there is a clear transition: above a certain modulator level the velocity drops to zero. This allows the robot to recharge until the modulator falls again. Other velocity spikes are due to the exploration behaviour reducing one motor velocity in order to turn.

After the first few passes, the phase plot settles into what appears to be a limit cycle, which would likely be triangular were it not for the arena boundaries. This starts at the bottom-left with low charge and distance, the robot remaining stationary and charging until just over 0.6. It then moves away from the emitter, until the charge falls to around 0.3, when it returns to the emitter. Once at the emitter it recharges again.

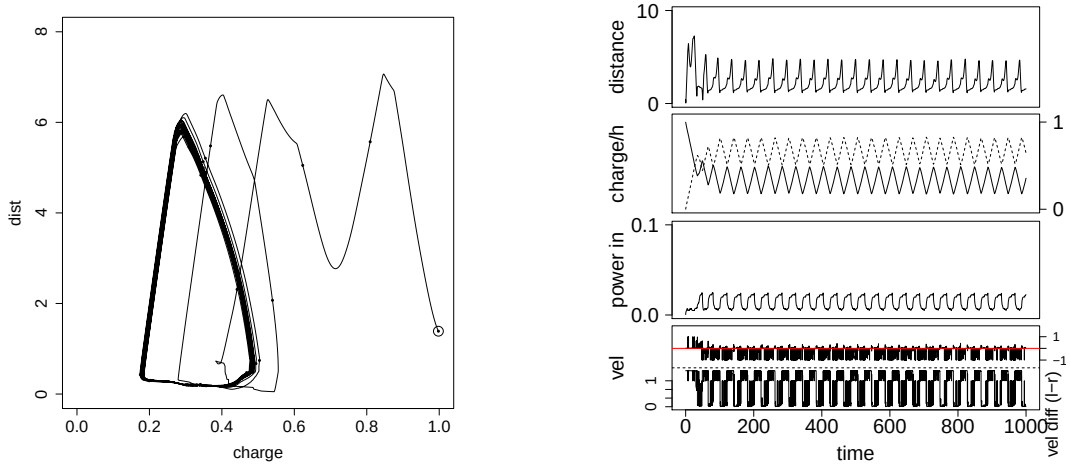
Fig. 9.11 shows that the h-as-input networks are less consistent than the output blending networks. To illustrate this, plots from the second-best network (number 6) are shown in Figs. 9.16 and 9.17. The behaviour here is similar to the best network, but the transition points are different: Fig. 9.17a shows that the system moves away from the emitter once the charge reaches 0.5. This small difference has a large effect on the system: the excursions are shorter because the system does not build sufficient charge to explore far before returning.



(a) Run 0 **(b)** Run 2

Figure 9.16: Two position plots for different runs of the second-best h-as-input network according to the combined metric. The start position is shown by a circle.

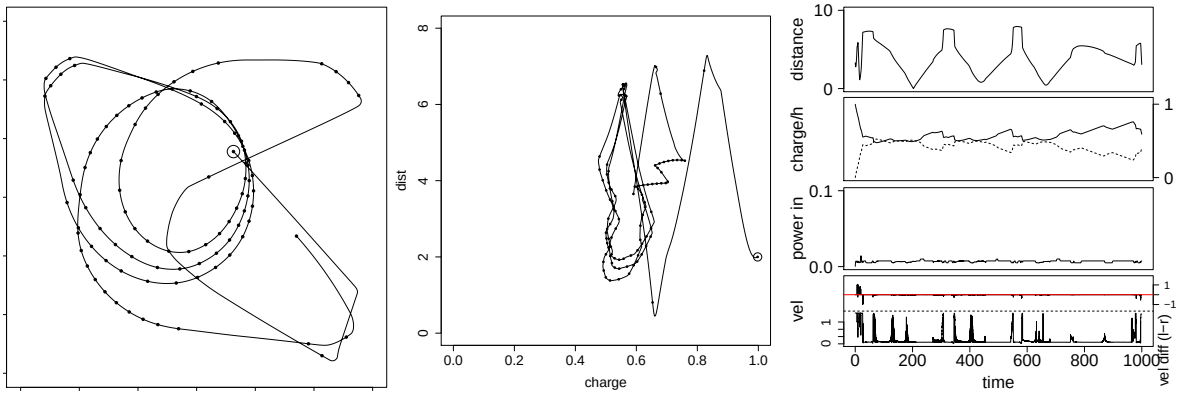
Other more serious failure modes exist. A run of a network near the mean, network 7, is shown in Fig. 9.18. Here, after the charge has fallen below the initial high level, the robot circles slowly around the emitter while still avoiding walls. In network 3, shown in Fig. 9.19, the robot rapidly spirals away from emitter and moves



(a) Phase (run 0) **(b)** Variables (run 0)

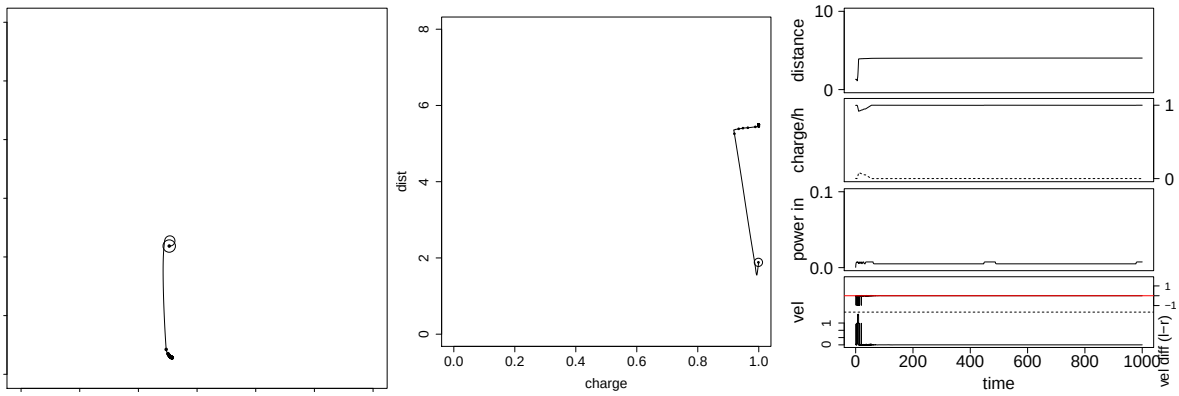
Figure 9.17: Phase and variable/time plots for run 0 of the second-best h-as-input network according to the combined metric.

towards a wall. This is not a smooth motion: consider the velocity plot. Once the charge begins to decrease it stops, surviving by not moving.



(a) Position **(b)** Phase **(c)** Variables

Figure 9.18: A run of network 7 of h-as-input



(a) Position **(b)** Phase **(c)** Variables

Figure 9.19: A run of network 3 of *h*-as-input

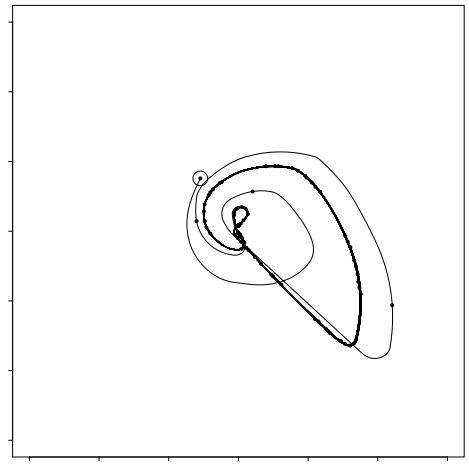
9.4.2.4 UESMANN results

The position plots of two runs of the best UESMANN network (number 9) are shown in Fig. 9.20, with the phase and variable plots for one of the runs.

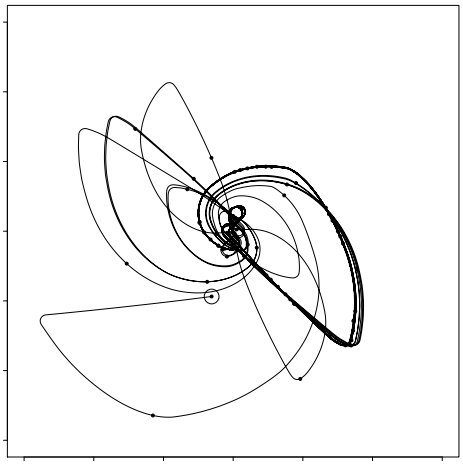
The performance is comparable with that of Fig. 9.16 (the second-best *h*-as-input network), with the edge-weighted distance and distance variation slightly better in that network. In the phase plot, the robot remains static at the emitter until the charge reaches 0.6 when it excurses. The excursion continues until charge 0.5, when the robot begins to return to base resulting in a minimum charge of 0.4. This result is more conservative than both *h*-as-input networks, in which the charge drops to below 0.2.

Two networks whose performances are closer to the overall mean are shown in Figs. 9.21 and 9.22. These both show an initial excursion followed by a tight loop around the centre of the emitter.

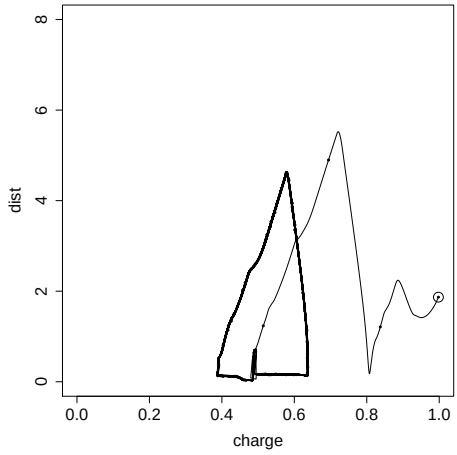
A poor UESMANN network is shown in Fig. 9.23, in which back-propagation has converged to a local minimum. Here, the robot moves at a constant but slow speed with a degree of phototaxis at the start but not towards the end of the run. This is not the behaviour required; we would prefer the phototaxis to be evident when the charge is low. The network successfully avoids the walls, but because there is no phototaxis as *h* increases, the run terminates in less than a minute of simulated time.



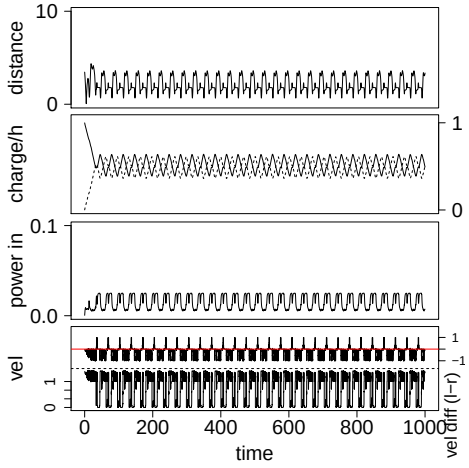
(a) Run 0



(b) Run 2

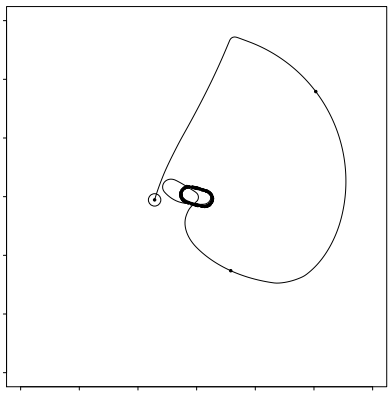


(c) Run 0 phase

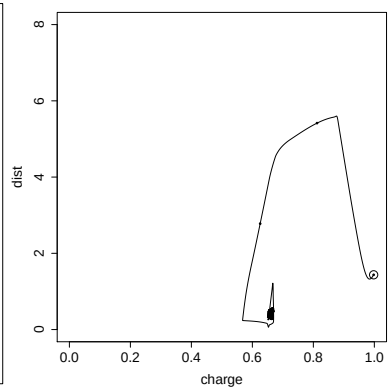


(d) Run 0 variables

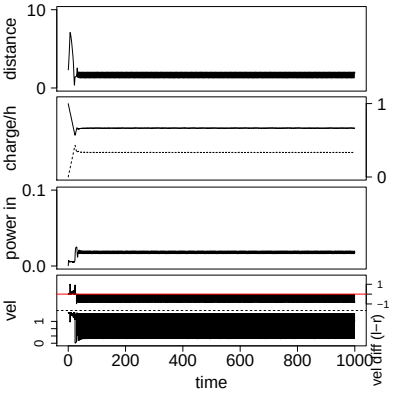
Figure 9.20: Two position plots for different runs of the best UESMANN network according to the combined metric.



(a) Position

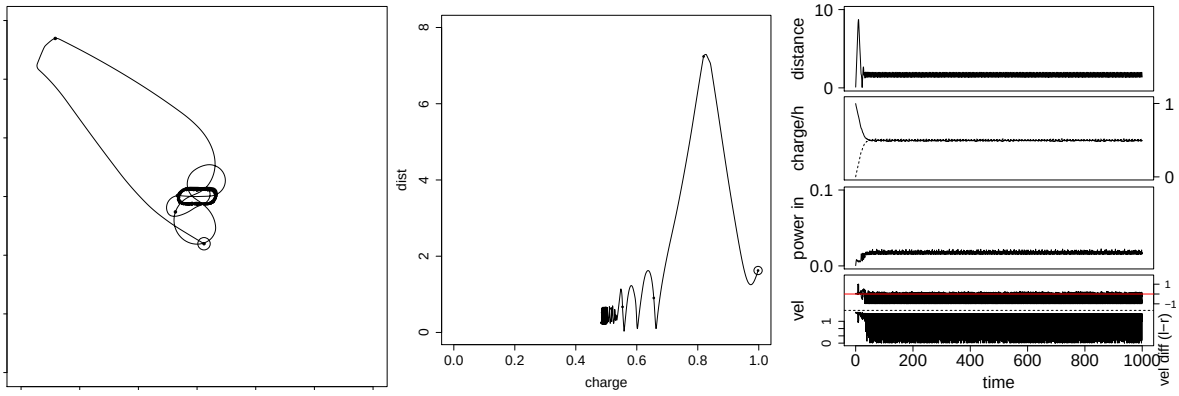


(b) Phase



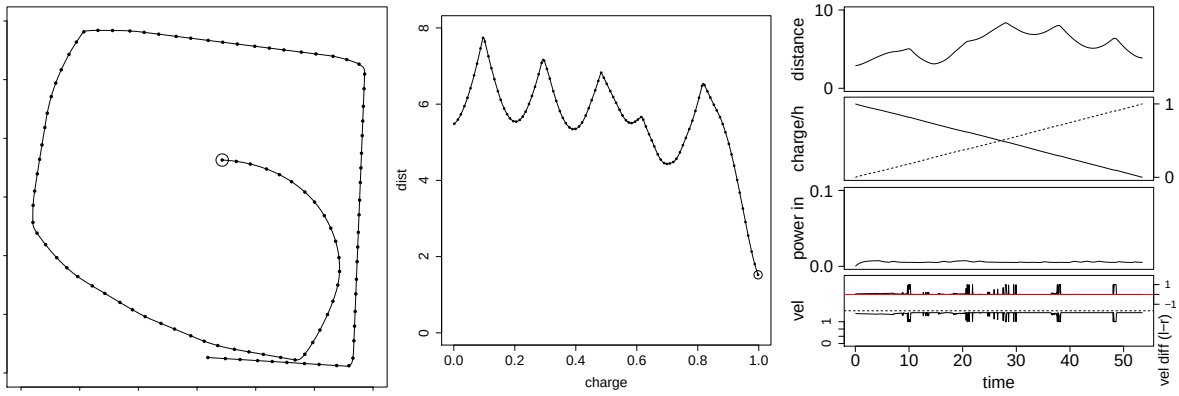
(c) Variables

Figure 9.21: A run of network 2 of UESMANN



(a) Position **(b)** Phase **(c)** Variables

Figure 9.22: A run of network 3 of UESMANN



(a) Position **(b)** Phase **(c)** Variables

Figure 9.23: A run of network 1 of UESMANN

9.4.3 Discussion

The run plots above show that output blending is a poor choice for this problem. Output blending has a smooth, linear transition between the two networks because it is a simple linear interpolation of the networks' outputs. In the previous classification experiments, this leads to a sharp transition under thresholding as has previously been explained (see Sec. 4.5, p. 121).

Here, this smooth interpolation between the two behaviours and the linear relationship between charge and modulator results in a small amount of phototaxis when the charge is not exactly 1. This causes the robot to always curve towards the light even when the charge is high. This can be demonstrated by taking the best output blending network and running it at a fixed modulator level, as shown in Fig. 9.24. Here, even a modest $h = 0.1$ provides enough phototaxis to keep the robot away from the walls in most cases. The $h = 0.2$ case shows a fairly tight loop around the emitter.

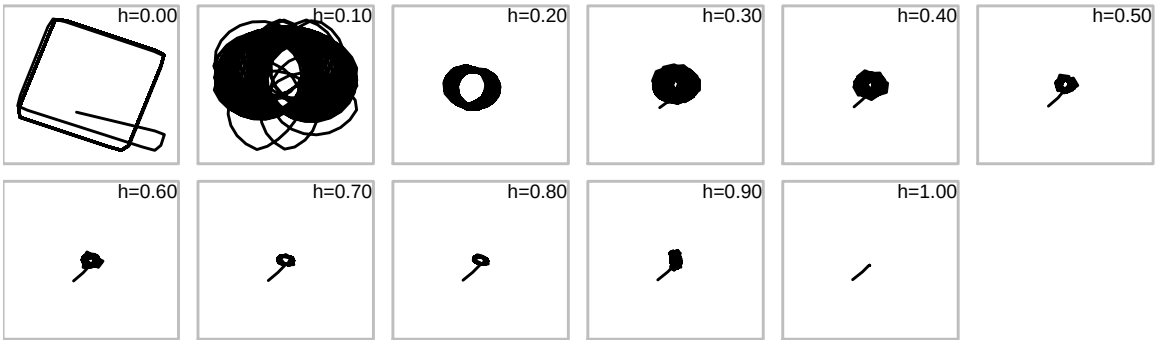


Figure 9.24: Position plots for output blending network 1 run for 1000s with different fixed modulator levels.

The h-as-input and UESMANN networks perform better because they have a non-linear transition in their behaviours: at modulator values close to zero they will (ideally) perform the *exploration* behaviour, while at values close to one they will perform *phototaxis.* The exact nature of this transition constitutes the difference between those networks of each type which are not local minima. In both networks, the transitions between the two behaviours follow the general scheme shown in the distance/charge phase diagram in Fig. 9.25. Here, the initial maximum charge leads to an excursion. Once the charge drops below a certain level, the robot returns to the emitter and a cycle begins, which consists of a stationary recharging phase, followed by an excursion and return.

There are three distinct points of transition in each network:

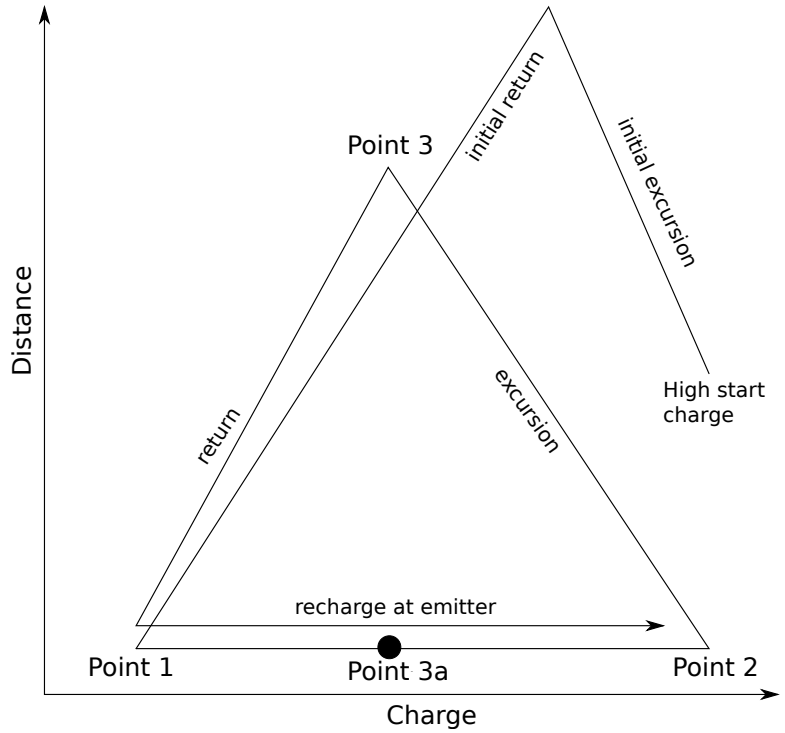


Figure 9.25: Distance/charge phase diagram showing the typical robot behaviour in h-as-input and UESMANN experiments. The labelled transition points are discussed in the text.

- Point 1: transition from moving to stationary at the emitter under phototaxis. This occurs as a consequence of the *phototaxis* behaviour and is not a switch between behaviours: in the rule-based controller from which the network is trained (Algorithm 10) the robot comes to a halt as the total light input rises from k_{Lmin} to k_{Lmax}.

- Point 2: transition from recharging to exploration. This switch is from the *phototaxis* behaviour back to *exploration* and occurs once the modulator drops below (i.e. the charge rises above) a certain level.

- Point 3: transition from exploration to return. This is a switch from the *exploration* behaviour to *phototaxis*, which occurs when the modulator rises above a certain level.

Note that the output blending network does not have this kind of behaviour: here, the behaviour passes smoothly through the four phases shown diagrammatically in Fig. 9.26.

- Phase 1: the charge is relatively high, so *exploration* exceeds *phototaxis* — the robot moves forwards with only a slight tendency to turn to the light.

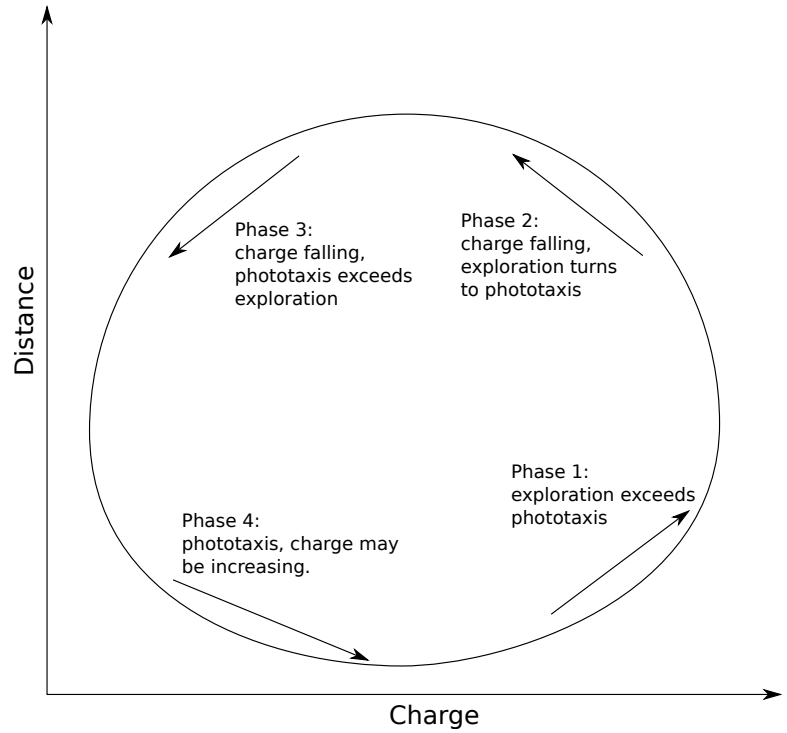


Figure 9.26: Distance/charge phase diagram showing the predicted behaviour in output blending experiments.

- Phase 2: as the charge falls, the robot's tendency to move forwards decreases, and it begins to turn towards the light. The distance is still increasing, however, because the turn will be gradual.
- Phase 3: the charge has fallen sufficiently and the robot has turned enough in the previous phase that it is now moving towards the light.
- Phase 4: the robot will slow as it approaches the light, but it may gather enough charge that *exploration* causes it to pass through the light without stopping. In this case, the transition between phases 4 and 1 will be sharp.

This can be seen in the detail of a run of the best output blending network in the simple simulator, Fig. 9.13b (p. 259), showing a very tight cycle around the emitter.

9.4.3.1 Discrepancy between transitions at different light levels (UESMANN and h-as-input)

We can see from Figs. 9.15a and 9.20c that the charge values are not the same for points 2 and 3 in both h-as-input and UESMANN. If they were, the system would constantly oscillate between the two behaviours once point 3a was reached. Instead, the robot remains stopped at the emitter until the charge is fairly high. This wide

separation between points 2 and 3 on the charge axis is likely to be desirable: a large reservoir of charge built up during this part of the cycle will lead to longer excursions.

It may be that the behaviour "zero motors when light input is very high", which is part of *phototaxis*, dominates the learned solutions such that it manifests when the modulator is further away from the *phototaxis* end of the modulator range than the rest of that behaviour. This may be because it is a simple rule to represent in either network. As such, minima may be learned which perform this rule over a wider range of modulator values than the elements of *phototaxis* which perform phototaxis *per se*. Fig. 9.27 shows this behaviour in schematic form.

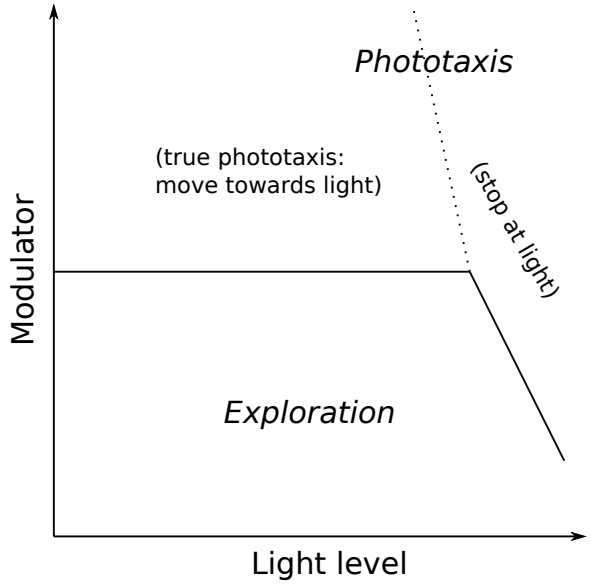


Figure 9.27: A possible explanation for the discrepancy between transitions at different light levels between *exploration* and *phototaxis*. The transition between the two behaviours may not occur at the same modulator level for different light levels, because of the two sub-behaviours of *phototaxis*. The region boundaries are indicative only — their relative positions may be quite different (or indeed non-linear).

This may be a natural outcome of a network transition based on changing the behaviour of the network itself, rather than interpolating between two networks. Consider the MNIST transitions in Sec. 7.3.5, particularly Table 7.19 showing the mode of the transition behaviour for UESMANN (p. 223): we see that certain digits change their output label quickly, while others take longer. Note in particular the difference between digits zero and one: a one is recognised as its nominal label until $h = 0.6$, while the label for most zeroes transitions at $h = 0.25$. Thus in MNIST, the "rule" for labelling "one" as 1 persists through the transition, while the "rule" for labelling "zero" as 0 changes very quickly. Similarly, in the robot, the "rule" for

"stop when the light input is high" may hold for more of the modulator range than the rule "drive as fast as possible avoiding walls."

It is worth noting that the best UESMANN network shows a very small excursion and return during the recharge cycle, at a charge of about 0.48. This may be an attempt to return to exploration at point 3a, as would be the case if the network did not have the transition discrepancy highlighted above. However it is significantly below the charge at point 3, so it is more likely to reflect some underlying complexity in the transition between behaviours.

9.4.3.2 Performance effect of charge at transition

Perhaps the most significant value to affect the relative performances of h-as-input and UESMANN is the charge at point 3: the transition from *exploration* to *phototaxis* at which the robot begins to return to the emitter. In the best h-as-input network this is 0.43, while in UESMANN it is 0.58. Thus h-as-input permits the charge to fall to a lower level before a return to recharge is necessary, allowing more time for exploration. UESMANN is more conservative in its power use.

9.4.3.3 Overall nature of the transitions

It is possible to gain some insights into the transition behaviour for the different networks by forcing them to run at a given modulator level with no charging model (and thus no termination due to discharge). By allowing them to run for a long simulated time and plotting the mean distance from the emitter, we should see a high distance when *exploration* is being performed and a low distance when *phototaxis* is being performed.

In Fig. 9.28, 50 runs (each starting from a random position and orientation) were performed for 1000 simulated seconds over a large range of modulator values. The large variation within each plot is because we are measuring the mean distance of a number of robots which are moving around the arena. Output blending shows a smooth transition between the two behaviours which appears to follow an exponential curve: most of the modulator range shows low distance, resulting in poor performance for reasons detailed above. The h-as-input network shows exploratory behaviour for most of the modulator range, sharply transitioning to *phototaxis* between $h = 0.6$ and $h = 0.7$ (i.e $0.3 < C < 0.4$), which corresponds with the phase plot in Fig. 9.15a. UESMANN follows an irregular curve with a high degree of "non-monotonicity"[6] indicating a complex transition: this is perhaps not surprising, given

[6]We shall use this term in a loose sense, to describe the property some networks have of moving between the two behaviours several times throughout the modulator range rather than performing

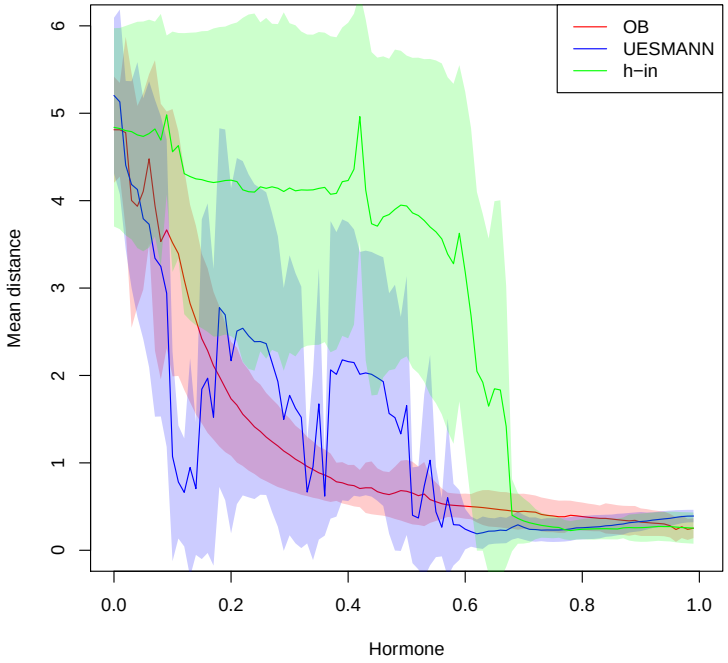


Figure 9.28: Mean distance from the emitter over 50 runs of the best network of each type (according to the combined metric) with the modulator fixed at various levels, with no charging model. The shaded area shows the mean ±1 standard deviation.

the underlying complexity of the transition function shown in simpler problems (see, for example, Sec. 6.5). The mean distance is slightly higher over the modulator range than output blending, but lower than h-as-input as we would expect from the relative performances.

It is worth noting that much of the difference in performance between the networks is due to the overall shape of the transition curve, which affects the modulator levels at which the behavioural transitions occur. Naturally, this is easy to modify by passing the modulator through a suitable non-linear function before applying it to the network. It would be straightforward, for example, to pass the modulator through a sigmoid such that the transition region for UESMANN occurred over a lower modulator (higher charge) range, which would increase the performance significantly[7].

one behaviour at $h = 0$ and moving smoothly to the other at $h = 1$. Strictly speaking, a behaviour is a complex function and not a value, so "monotonicity" is perhaps not a term which should be applied to it.

[7]It should be borne in mind that this would not affect the training or the network solution: this procedure would be applied to the modulator during network operation, and does not refer to the sigmoid activation function of the network nodes themselves.

However, if this were applied to output blending to remove most *phototaxis* from the low modulator range and *exploration* from the high modulator range, we would lose the potentially useful difference on the charge axis between the charge leaving the emitter and the charge arriving at it, described in Sec. 9.4.3.1, which seems to be unique to *h*-as-input and UESMANN.

The correctness of the behaviours themselves (both at the extrema and in transition) is more of interest here, and the nature of the transitions. The simulations have shown that UESMANN and the other network types are capable of learning the rule-based controller behaviours with enough accuracy to perform well in a perfect simulation (i.e. the same simulation with which the controllers were trained). Also, while it has been shown that the natures of the transitions differ – UESMANN's transition is considerably more non-monotonic and "spiky" than the other networks – the simple simulator does not show the effect this might have on a real robot. The next chapter will address this.

Chapter 10

Robot and Gazebo experiments

The previous experiments demonstrate that all three networks are capable of learning the two behaviours well and transitioning between them. If we compare the transition regions with those for the MNIST experiments in Sec. 7.3.5, we see similar behaviours with UESMANN producing a wider transition than h-as-input (output blending produces a sharp transition in those experiments because of the thresholding at 0.5 for classification).

In these simple simulator experiments, h-as-input shows a higher level of performance due to the overall shape of its transition curve. UESMANN is more conservative in nature. As stated above, it would be trivial to change the charge/modulator mapping from the current linear function to a sigmoid, to optimise the performance of each network. However, we will continue to use the mapping $h = 1 - C$, while recognising that the transition shape is less important than the correct functioning of the network throughout the modulator range.

The simulation used in the above experiments is extremely simple: the sensors and actuators are "perfect" and there are are no complex physical interactions with the world. Robotic controllers developed in simulation often fail when incorporated into real robots: the so-called "reality gap" discussed in Sec. Sec. 2.4 (p. 41). To study how well the various networks developed bridge this gap, the most performant networks were carried over into experiments using the Gazebo simulator for initial evaluation, before being moved onto a real robot.

10.1 The robot and the arena

The real robot used in the experiments was a Pioneer 2-DX, as described in Sec. 8.1.1 above. The experiments were performed indoors in a large, irregularly shaped arena formed by a sonar-opaque mesh, which is shown approximately in Fig. 8.2 (p. 235).

Note that this image does not accurately show the positioning of the mesh during the run: this is shown in plan form in Fig. 10.1. Modifications were required for practical and logistical reasons. The light source was an incandescent lamp suspended from the ceiling, as shown in Fig. 10.2. The lamp illuminated an area of the floor which was clearly visible in the omnidirectional camera. The red LED on the camera is a visual tracking aid, as will be discussed later.

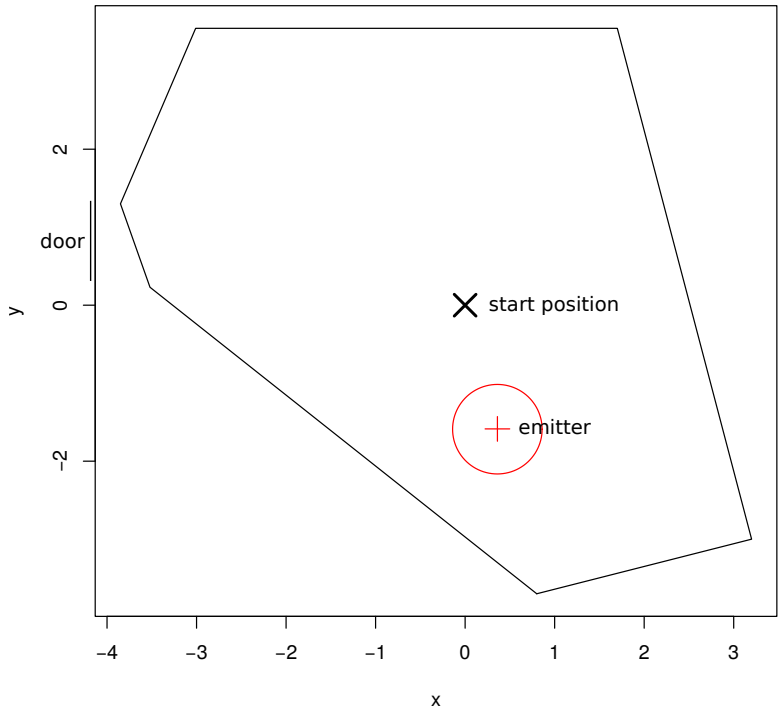


Figure 10.1: A plan of the final real-world arena. A red circle and cross mark the position of the lamp and the rough extent of the circle of light it casts, while a black cross marks the robot starting position. The text refers to the positive y axis as "north." Axis units are metres.

10.1.1 System architecture

The system uses the Robot Operating System (ROS) [225] to run the neural network and communicate with either the robot or the Gazebo simulator. This allows the same software to run the network in both environments, while using different sensor inputs and actuator outputs. Full details of the system are given in Appendix A. In brief, the neural network is always run on a host computer (not the robot) and communicates with either Gazebo (also running on the host) or the robot.

Figure 10.2: The robot and its light source during an experimental run. The red LED is used for tracking.

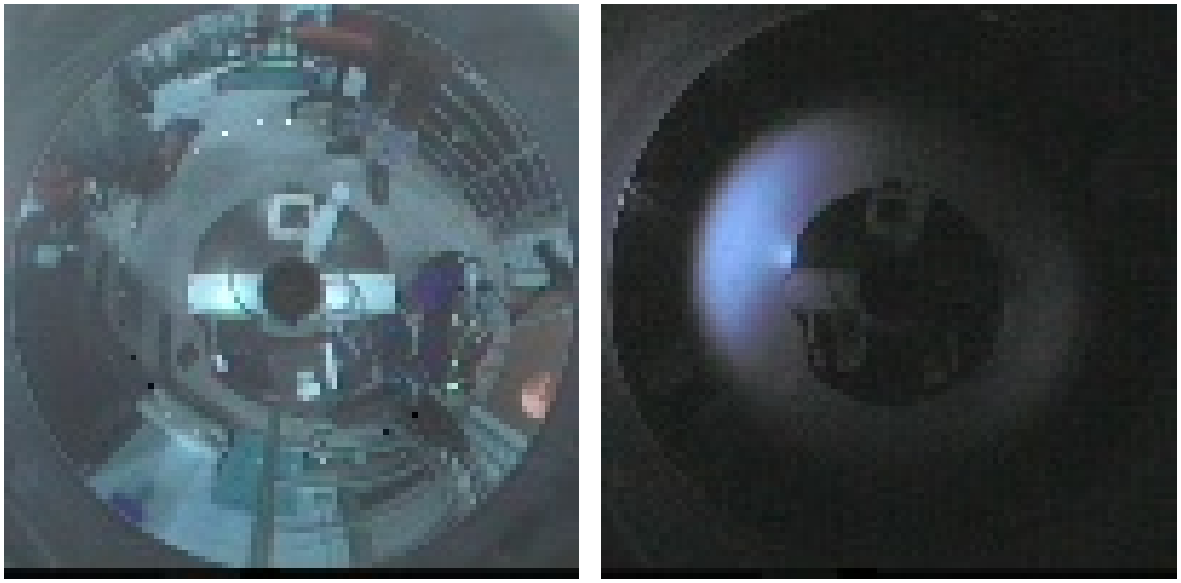
10.1.2 Light sensor

The networks have been trained using a simulated emitter as their light source which subtends an angle in the sensor's view. All pixels within the angle are white, all others are black. In the experiment, the real sensor should produce similar values. This is done by taking the image from an omnidirectional camera and summing pixels along radii in the image. To generate hard black and hard white values in the resulting vector, and to permit the "pool of light" projected by the lamp to subtend at the camera similar angles to the simulated emitter, a series of Gaussian blurs and threshold operations are used. The design was chosen to keep the light sensor system generic and flexible for future use, while providing the required light pattern to the network. In robot experiments components of this system run on both the Pioneer and the host as described in more detail below.

10.1.2.1 Light sensor components on the robot

The light sensor used by the robot converts an omnidirectional camera view into an array of pixels. On the Pioneer, this is from a camera pointing upwards at a hyperbolic reflector. The frame grabber used (an Arvoo Picasso) is very slow, reading a 550×550 pixel image every ~ 0.3s, but adequate given the slow speed of the robot (for safety reasons). Using OpenCV, this image is resized to 100×100 and a Gaussian blur applied. This is to ensure that the next stage – radial summing of

pixels to produce a vector of pixels around the camera – will capture bright regions of the image which are between two sampling radii (see below). Two typical images (before blurring) are shown in Figure 10.3.



(a) Image of the lab with ambient lighting **(b)** Experimental image near the light source

Figure 10.3: Images from Pioneer omnidirectional camera (dots in (a) added indicate the light value along each of 32 radii)

RGB values are then sampled along 32 radii from the centre of the image, one for each element of the output array (these are downsampled later to 8 for input to the network). The samples are summed, and then normalised to the range [0,1] (each colour channel being processed separately). The three colour channels are multiplied by 255, clamped and composed into a vector of 32 8-bit RGB triples which is then sent from the robot to the host. The process is shown in Algorithm 11.

10.1.2.2 Light sensor components on the host

Once on the host computer, each colour channel in the resulting linear array of RGB values – representing a circle around the robot – is processed according to the algorithm in Algorithm 12. This blurs each channel and renormalises it independently, giving a similar effect to the morphological dilate operator, "spreading out" the lit region. This is necessary because the angle subtended by the light source is smaller than in the simulation. The result is then thresholded to reduce noise, because the network was trained on clean images.

These 32 pixels are then sent to the ROS "node" which runs the network (see Appendix A). In that node, the RGB channels are averaged into a single monochrome

Algorithm 11 Light sensor algorithm used on Pioneer. The $clamp(x, a, b)$ operation limits x to the interval $[a, b]$.

Require: Img = image from camera
Ensure: $(\overline{red}, \overline{green}, \overline{blue})$ = three channel 1D image
  $Img \leftarrow GaussianBlur(Img)$ {kernel size=31, $\sigma = 2$}
  $w \leftarrow 100$ {w is image size}
  $step \leftarrow 2\pi / OutputSize$ {OutputSize = 32}
  $\theta \leftarrow 0$
  for $i = 0$ **to** $OutputSize - 1$ **do**
    $red_i \leftarrow 0, green_i \leftarrow 0, blue_i \leftarrow 0$
    {use pixels 0.16 to 0.5 times the image width along each radius}
    for $r = 0.16w$ **to** $0.5w$ **do**
      $x \leftarrow r \sin\theta + 0.5w$
      $y \leftarrow r \cos\theta + 0.5w$
      $\bar{p} \leftarrow Img(x, y)$ {read pixel from image}
      $red_i = red_i + p_r$
      $green_i = green_i + p_g$
      $blue_i = blue_i + p_b$
    end for
    $\theta \leftarrow \theta + step$
  end for
  $\overline{red} \leftarrow clamp(255 \cdot normalise(\overline{red}), 0, 255)$
  $\overline{green} \leftarrow clamp(255 \cdot normalise(\overline{green}), 0, 255)$
  $\overline{blue} \leftarrow clamp(255 \cdot normalise(\overline{blue}), 0, 255)$
  return $(\overline{red}, \overline{green}, \overline{blue})$

channel and the resulting monochrome pixels are decimated down to 8 by applying a further Gaussian blur ($\sigma = 1$, kernel size 5) to the vector and downsampling. These are then sent to the neural network under test, having been converted from [0,255] to [0,1] in range. This is shown in Algorithm 13[1]. Figure 10.4 shows the various stages of signal processing.

10.1.2.3 Gazebo simulated light sensor

The purpose of the simulated light sensor is to match as closely as possible the actual sensor on the robot, rather than the very simple simulation used in training. However, by necessity it uses a similar strategy to the simple simulator. The sensor is written as a plugin for Gazebo which traverses the objects of the simulated world at 10Hz. If the plugin encounters an object with a name of the form `lightrgbXXX` where `XXX` is a hex triple, it converts the triple into RGB values and calculates the angle in

[1]Note that this was later found to introduce a bug, in that the ceiling and floor operators, intended to expand the emitter when linear (as in the training arena), reduce the possible set of outputs from the sensor ring to a discrete set of integer rotations. See Sec. 10.2.2.4 for details of the effects.

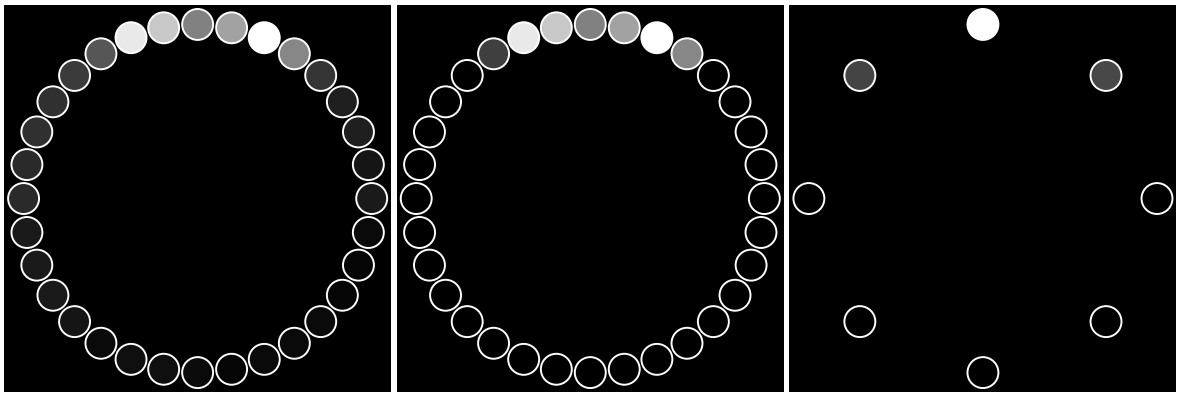
Algorithm 12 Light sensor algorithm in the bridge client node on the host.

Require: Img = linear image vector from server, as 3 channels of RGB
Ensure: Img = blurred, normalised and thresholded linear image
  $thresh = 0.3 \times 255$ {assign threshold level}
  for all $\overline{c} \in Img$ **do** {for each channel}
    $\overline{c} \leftarrow GaussianBlur(\overline{c})$ {kernel size 7, $\sigma = 1.2$}
    $\overline{c} \leftarrow normalise(\overline{c})$ {normalise to range [0,255]}
    for all $x \in \overline{c}$ **do** {for each pixel}
      $x \leftarrow \begin{cases} 0, & x < thresh \\ x, & \text{otherwise} \end{cases}$ {threshold the values, replacing them in $\overline{c}$ and thus in Img}
    end for
  end for
  return Img

(a) Raw camera image, down-sampled to 100×100

(b) Image after Gaussian blur with $\sigma = 2$, with dots showing sampled values

(c) Sensor values received by bridge (channel sum)



(d) After Gaussian blur with $\sigma = 1.2$

(e) After thresholding at $\lfloor 0.3 \times 255 \rfloor = 76$

(f) After decimation: input to neural network

Figure 10.4: Light sensor processing

Algorithm 13 Light sensor algorithm on the host, producing a vector of neural net inputs.

Require: $outsize$ = output buffer size (8), Img = linear input image from bridge node

Ensure: out = output buffer, decimated to $outsize$ pixels, $total$ = total brightness of image

{Create mono image $\overline{m}$ by averaging channels and converting range to [0,255]}
$total \leftarrow 0$
for all $p(i) \in Img$ **do** {for each pixel}
$m(i) \leftarrow (p(i)_r + p(i)_g + p(i)_b)/(255 * 3)$
$total \leftarrow total + m(i)$
end for
{Decimation consists of Gaussian blur followed by linear interpolative down-sampling}
$\overline{m} \leftarrow GaussianBlur(\overline{m})$ {kernel size 5, $\sigma = 1$}
$step \leftarrow \text{size}(Img)/outsize$
$pos \leftarrow 0$
for $i \leftarrow 0$ **to** $outsize - 1$ **do**
$i_1 \leftarrow \lfloor pos \rfloor$, $i_2 \leftarrow \lceil pos \rceil \bmod \text{size}(Img)$
$t \leftarrow pos - i_1$
$out(i) \leftarrow (1 - t)m(i_1) + tm(i_2)$
$pos \leftarrow pos + step$
end for
$\overline{out} \leftarrow \overline{out}/255$
return $\overline{out}, total$

the xy-plane subtended by the object's axis-aligned bounding box at the camera. It then adds these RGB values to a circular array of size 100, and applies a Gaussian blur ($\sigma = 1$, kernel size 5) before sending the array to the control node. Once on the host, the array is decimated down to 8 pixels in the same manner as on the real robot. While the array is larger than the 32 element array from the robot, more granularity here provides smoother edges after the blur: the edges which result from the initial pass through the objects are very crisp.

In the simulated arena used, the emitter was simulated as a white cylinder roughly the same size as the circular pool emitted in the real arena. Given the use of the axis-aligned bounding box, this becomes a cuboidal emitter in the sensor code.

10.1.2.4 Simulated power input from light

In the real robot, the power input value is calculated from the sum of the light inputs. It proved difficult in the simulator to use this data to generate values to provide to the charging model: the simulator generates a simple blurred image of

a circular emitter, while the real emitter is a diffuse reflection of a point light with some specular elements. To solve this, a number of runs were made on the real robot, recording the light sensor sum and the distance from the emitter. These were used to construct a lookup table of emitter distance to light sensor sum. The simulator plugin uses this to generate a "bright" value which it publishes to the control node, which (when running in simulator mode) uses it to generate the power input.

10.1.3 Sonar sensors

In initial testing on the real robot, it was found that the networks behaved erratically due to occasionally receiving extremely long distances from the sonars. Because the sonar return times are fed directly into the network, these very high values had a disproportionately large effect on networks which had been trained on completely enclosed arenas.

It might have been more principled to apply an inverse transformation to the values (e.g. $v' = \frac{a}{v+b}$). This would mean that short distances would produce a high value, while values given by longer distances would asymptotically approach zero, thus minimising the noise at longer distances. It would also fit the problem: short distances are those at which avoidance should be performed, so a non-linear mapping in which close distances occupy more of the mapping's range would be useful. However, this would have required completely retraining the networks and repeating all the simple simulator experiments. Instead, therefore, a 5m cap was imposed on all sonar readings from the robot.

10.1.4 Actuators

The left and right motor values are taken from the network outputs and clamped to the range $[-1, 1]$ for safety. They are then multiplied by a constant due to the simulated robot speeds being different in the two simulators, and different from the motor speeds on the robot. On the robot, this constant was 0.05 while on the simulator it was 0.499. These values were established by finding a safe speed for the robot and then calibrating the simulator to move at the same speed for the same motor output values.

There is no guarantee that the motor control values coming from the network are smooth: it may be that a small variation in input or modulator produces a large change in output. These were found to cause stalls in the low-level motor control systems which required a manual reset. To avoid this, a degree of smoothing using

an exponential weighted average is applied separately to each motor channel:

$$m_t = (1 - \alpha)x_t + \alpha m_{t-1}, \tag{10.1}$$

where m_t is the motor control value at time t, x_t is the output from the network at that time, and α is the smoothing constant. This smoothing runs every 0.2s in a separate thread on the bridge server node, which with the smoothing constant $\alpha = 0.8$ provides a half-life of around 0.6s.

10.1.5 Localisation

Robot localisation was performed visually using a commodity webcam connected to the host computer, with a view of the entire arena. The camera is in an enclosure bolted to a high beam to the bottom right of Fig. 10.1. An OpenCV application detects a bright red blob in the image (produced by a diffused red LED on top of the robot), finds the centroid, and performs a perspective transform to get world coordinates. These are then published and read by the ROS node running the network and incorporated into its logs. The tracking system is described in detail in Appendix B. Near the camera the tracker is accurate to 1cm, while far from the tracker the accuracy falls to 10cm. There is a "dead zone" in which the LED is occluded by the lamp hood; this is towards the top left of Fig. 10.1.

10.2 Experiments

The networks which achieved the highest combined metric in the simple simulator experiments of Sec. 9.4 were initially run using the Gazebo simulator, in order both to confirm that the assumptions made in developing the architecture described above were correct, and to explore the behaviour of the networks with a more realistic physics model in an arena the same shape as the real arena.

The behaviour of the system should be similar in both simulators, but the smaller size of the arena used in Gazebo was expected to cause differences which should be accounted for in the results. For example, a long excursion in the simple simulator may lead to failure of homeostasis because the robot starts the return to the emitter too late to recharge; while in Gazebo the robot may start to return during exploration because of a fortuitous "bounce" from the closer arena walls. However, the positive y direction from the start point (see Fig. 10.1) provides a long run into darkness, sufficient for failure.

10.2.1 Methodology

In all experiments, runs were made starting from the point given in Fig. 10.1 with the robot facing "north" (along positive y) or "south" (along negative y). The constants were as in Table 9.1, but k_{power} — the amount of simulated power from the light source — was set to two different values: 0.0025 (as in the simple simulator experiments) and 0.003.

Thus for each network we have four experiments, using different starting orientations to show the effect of different encountered topography, and different charging levels to show the effect of available power on homeostasis. Five runs were made of each experiment on the robot, because the complexity of real-world physics creates differences between runs with identical starting conditions. This was not the case in Gazebo (notwithstanding the tiny amount of Gaussian noise in the simulated sonar sensors), where only a single run was made of each experiment.

Due to the low number of runs and the variation between them within each experiment, the results of the robot runs were compared qualitatively with the corresponding Gazebo run and with the simple simulator runs. The analyses concentrate on whether homeostasis was achieved, but more importantly (given that the network transition region could be moved by conditioning the input as discussed above) how well the networks performed the end-point behaviours and how they transitioned between them.

Each run was performed for at least 1000 seconds if the robot did not discharge, collide or come to a stop before that time. This was an arbitrary time limit, determined by the experimental time available and the need to maintain the charge in the real battery.

10.2.2 Output blending

The results of the experiments for the best output blending network (according to the combined metric in simulation) are shown in Figs. 10.5 (for $k_{power} = 0.0025$, the lower charging level) and 10.6 ($k_{power} = 0.003$). The constant k_{power} determines how efficiently the simulated charger converts light to charge (see Sec. 9.1.4).

It is immediately obvious that the relatively small difference in k_{power} — a factor of 1.2 — makes a significant difference to the outcome. All "north-facing" runs (i.e. those which start with the robot moving along positive y away from the light source) run out of charge rapidly at the lower charging level, while at $k_{power} = 0.003$ all but one achieve at least 1000s. Of the south-facing runs, only the simulator and one

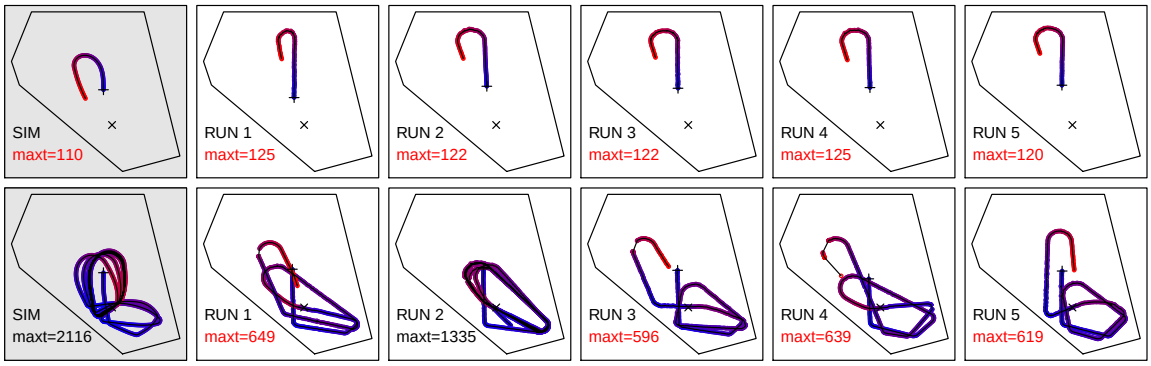


Figure 10.5: Output blending best network runs at $k_{power} = 0.0025$. The simulator runs are shown on the left, the remaining runs are robot runs. The position of the robot over time is shown, with the robot starting at the + symbol. The emitter position is marked by the × symbol. The colour indicates the charge: blue is high and red is low. The $maxt$ value is the maximum time, with experiments terminated either by zero charge or due to time constraints. In the zero charge case the $maxt$ value is given in red.

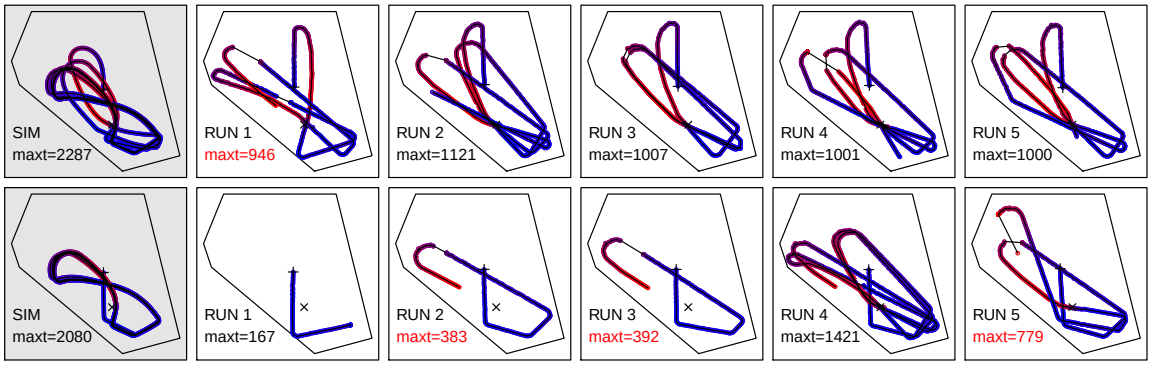
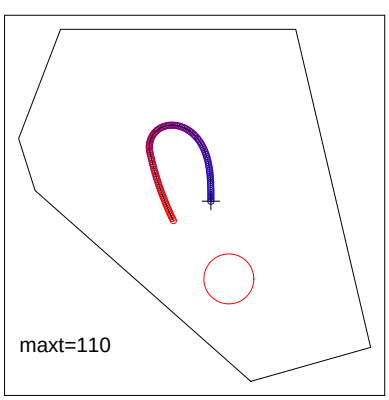


Figure 10.6: Output blending best network runs at $k_{power} = 0.003$. South-facing run 1 stops indefinitely facing into the wall (see Sec. 10.2.2.3).

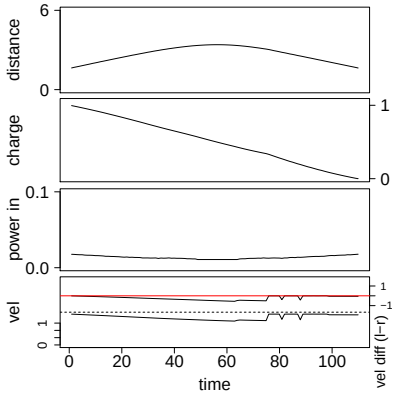
robot run achieve a time of 1000s at the lower charging level. Run 2 succeeds, but loops tightly around the illuminated, narrow southern end of the arena.

10.2.2.1 Runs at $k_{power} = 0.0025$

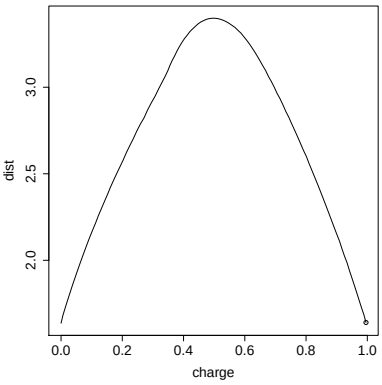
The north-facing Gazebo simulation at $k_{power} = 0.0025$ is shown in more detail in Fig. 10.7. As stated above, this is a non-homeostatic run, with charge exhausted after 110s. As expected, the robot smoothly begins to increase its turn as the charge falls, but the turn is still too late to permit survival. Robot run 3 is shown in Fig. 10.8. Here, the position plot shows the robot moving in a straight line, despite the network outputs requesting a slight turn. The robot does not turn until a time of 50s, despite a significant difference between the left and right motor outputs (as shown by the motor velocity difference in the variables plot). We will examine this behaviour later.



(a) Position plot

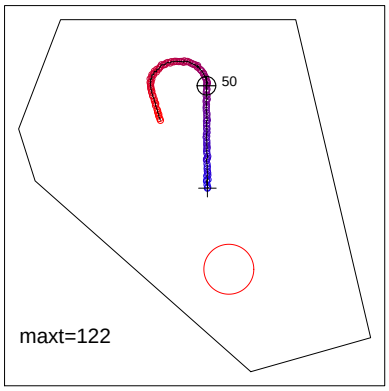


(b) Variable plot

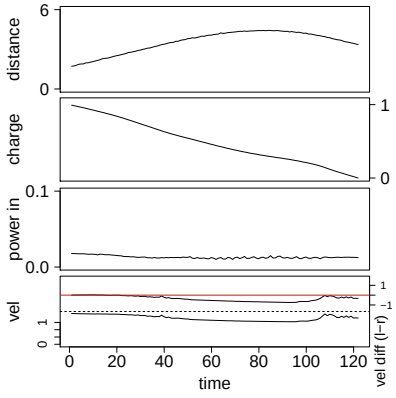


(c) Phase plot (distance/charge)

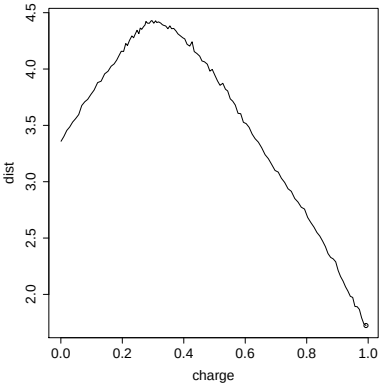
Figure 10.7: Simulated north run of output blending, $k_{power} = 0.0025$.



(a) Position plot



(b) Variable plot

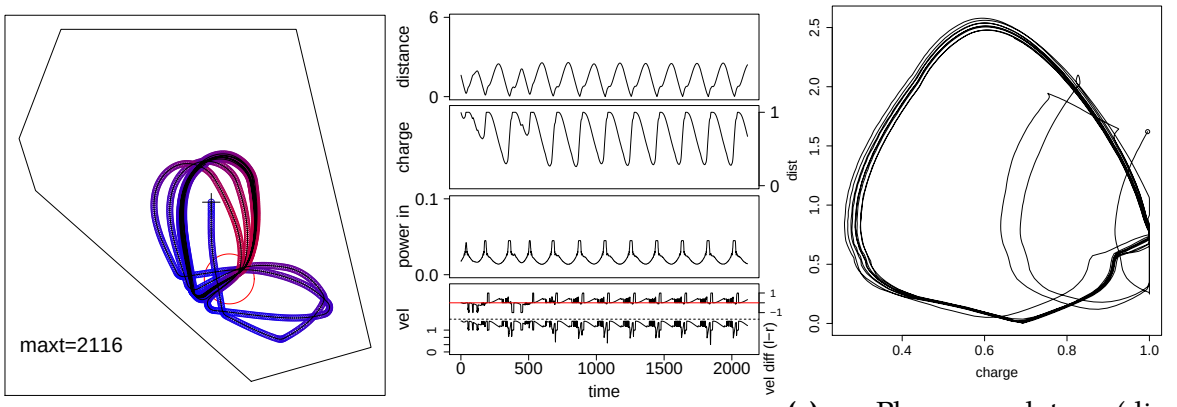


(c) Phase plot (distance/charge)

Figure 10.8: Robot run 3 (north) of output blending best network, $k_{power} = 0.0025$. Here, the point at time 50s is labelled in the position plot: see the text for details.

The south-facing simulator run at $k_{power} = 0.0025$ achieves homeostasis, as shown in Fig. 10.9. Once again, the robot's behaviour moves smoothly between *exploration* and *phototaxis* over the entire modulator range, as expected. In this run, however, the first excursion is into the narrow end of the arena. The robot loses less charge due to the proximity of the emitter, and performs two loops through this area due to "bounces" from nearby walls. The next "bounce" takes it into the wider arena, but the return phase starts earlier than in the corresponding north run because the charge is now lower. The robot is able to return, and settles into a loop roughly following the pattern in Fig. 9.26.

This behaviour is at first sight different from the simple simulator results shown in Fig. 9.12. However, both exhibit a long initial run (curtailed in the new arena by



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.9: Simulated south run of output blending best network, $k_{power} = 0.0025$.

the walls) followed by tight loops. In both the Gazebo simulation and robot runs this long initial run is often too long for the robot to return to recharge. If the robot does return, however, it settles into a tight loop similar to that seen in the simple simulator, where it appears much tighter because of scale: the simple simulator arena is much larger. The tight loops are caused by the constant presence of *phototaxis* behaviour across the entire modulator range, as discussed above.

10.2.2.2 Runs at $k_{power} = 0.003$

These are shown in Fig. 10.6. Here, the north-facing runs perform rather better, achieving several loops with all but run 1 achieving 1000s. However, these runs may not be indefinitely homeostatic: the runs have sections with very low charge, and it is possible that the robot's course will eventually cause it to fail.

The simulator run settles into a tight loop, similar to that in Fig. 10.9, but with straighter segments because the robot's charge is higher. However, in the robot experiments the course is typically straight, which was not predicted: as before, there should be an element of *phototaxis* unless the charge is at maximum.

North-facing runs 2 and 3 on the robot are nearly identical, with both resulting in a long run through the bright region which (due to the robot not slowing down) does not recharge the robot sufficiently. The robot then turns too late and fails. The long, straight sections here are unexpected and similar to the brief run seen in Fig. 10.8: again, given the smooth nature of the network transition the robot will be performing some degree of *phototaxis* and so should be turning.

Run 4 achieves homeostasis, fortuitously arriving on a looping path which passes into the dark northern region with a sufficiently low charge to permit it to return.

Run 5 nearly achieves this, but ends on a long run into the north, with similar results to those seen previously.

In the south-facing runs, the simulator quickly settles into a homeostatic figure-of-eight pattern similar to the simple simulator experiments of Fig. 9.12. In the robot, however, the performance is varied. Run 1 stops indefinitely, facing into the wall at full charge; we will look at this anomaly in the next subsection. Runs 2 and 3 again show the distinctive straight runs, which appear to cause the robot to turn too late. Run 4 achieves homeostasis, having taken a slightly different route from the previous runs such that it is low on charge but still close enough to the emitter to return. Run 5 takes a route somewhere between the extremes of runs 2 and 3 and run 4, which allows it to complete two loops, but then moves into the dark part of the arena and turns too late.

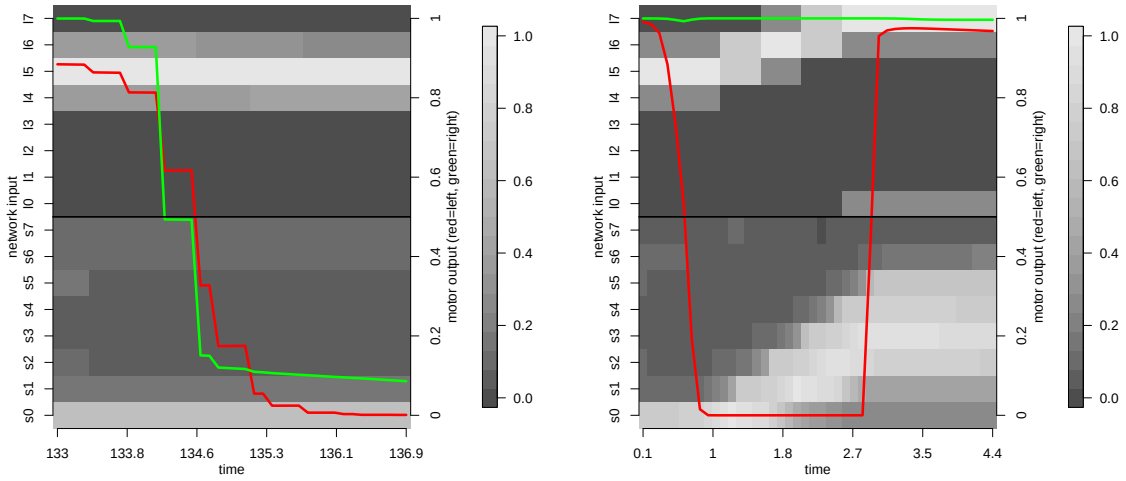
10.2.2.3 Anomalous stopping behaviour

South-facing run 1 at $k_{power} = 0.003$ is an anomaly: the robot simply stops moving, facing directly towards the mesh. Both network outputs are near zero, and the charge remains high due to the proximity of the emitter. All the networks were trained with examples provided by the simple simulator, which has perfect sonars with no noise: here, it may be that noisy sonars are providing an input for which the system has not been trained and for which it has not generalised. However Fig. 10.10a, showing heat maps of the network inputs over time for the period leading up to the stop, shows that when the robot stops it is receiving close and clean echoes for all but the far left sonar: the robot should turn left. Note that at the stop the charge is at maximum, so only the *exploration* network is being used.

Similar situations can occur in the training arena, so examples with similar sonar values should be present in the data set. When a simple simulator using the same network is started with similar values in an emulation of the final arena, the results are correct: the robot turns left, away from the walls.

Consider the network inputs at at $t = 134.6$ in the robot and at $t = 1.0$ in the simulator: they are similar, as is shown in more detail in Fig. 10.11. However, the network output is different at these points: both motors are slowing down on the robot, but only the left motor is slowing in the simulation. There is a slight difference in the modulator level, but this would result in a very small difference in the outputs given that we are using output blending.

The only significant differences are that sonar s_0 is larger in the simulation and that s_0 to s_5 in the robot show more variation, with the corresponding values almost



(a) Robot **(b)** Simple simulator

Figure 10.10: Network inputs leading to the erroneous stop in output blending robot south run 3, and the corresponding situation in the simple simulator. The heat map shows the sonar inputs (scaled down by 8) and light inputs, while the superimposed lines show the network outputs (red is left, green is right).

equal in the simulation. It may be that these differences constitute a situation for which the network has not been trained.

10.2.2.4 Why does the robot turn the wrong way?

One obvious feature not described above is the wrong direction of the *phototaxis* turn in Fig. 10.7 (and all other north-facing runs). This is due to the bug in the simple simulator light sensor (Algorithm 13) described in a footnote on page 277, in which the angles visible to the light sensor were inadvertently discretised. Here, the emitter appears to be directly behind the robot, which is interpreted as a right turn. This influenced the training data provided to the networks. The bug was detected at a late stage, and there were insufficient resources available to retrain the networks and re-run the experiments. However, the problem only manifests when the light is nearly directly behind the robot. While it would have an effect on the robot paths, it should not have an effect on the overall performance.

10.2.2.5 Why are the courses straight?

Output blending should produce curving courses when not facing directly towards the emitter, because there is always an element of *phototaxis* present when the robot is not fully charged (see Sec. 9.4.3 and Fig. 9.24). However, the robot shows many

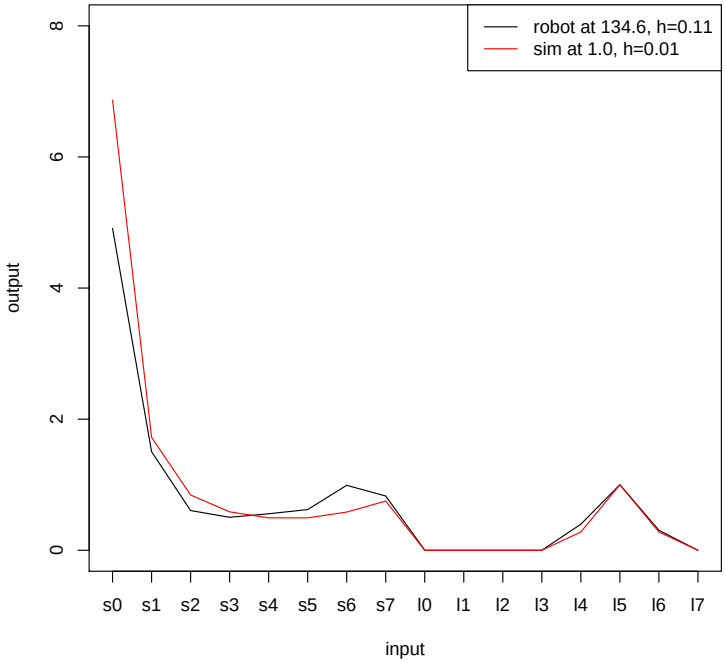


Figure 10.11: Plot of network input values in the robot and simulator networks for the anomalous stopping run in output blending.

straight runs. For example, in Fig. 10.8, the robot does not actually turn until 50s into north run 3, despite receiving different motor velocities for some time before this.

To examine this behaviour, several short runs were made with test values replacing the neural network outputs: the right motor was held at 1, while the left motor also started at 1, then after 10 seconds ramped down to a lower level with the ramp taking a further 10 seconds, as shown in Fig. 10.12. Robot experiments

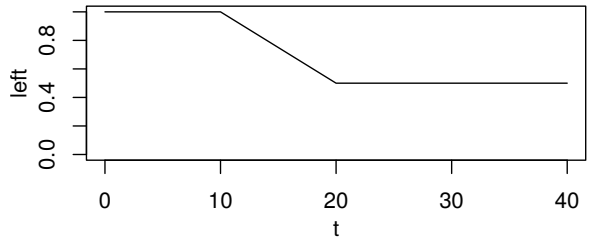
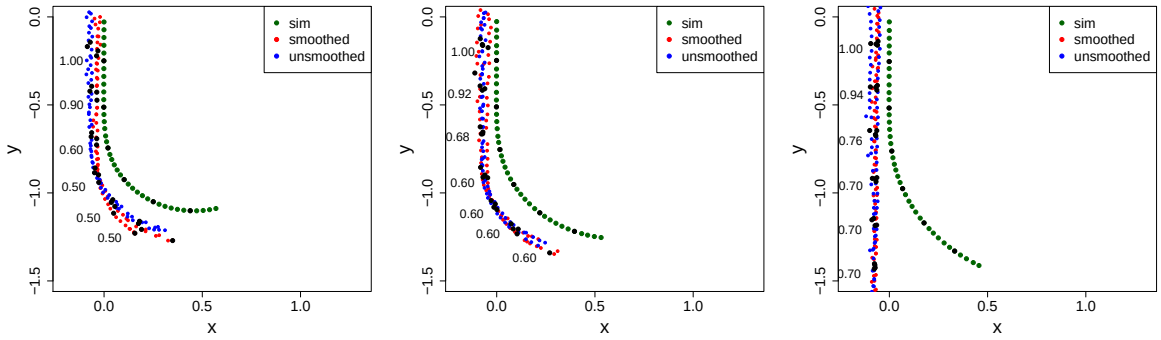


Figure 10.12: Output for left motor over time for differential drive tests, with the lower level at 0.5.

were performed with and without the smoothing described in Sec. 10.1.4 to ensure this was not the cause of any difference. We would expect to see a gradual curve starting at 10 seconds for both simulator and robot if there were no problems. What we actually see is shown in Fig. 10.13 for three different levels. It is clear that the

difference between the left and right motors needs to be high before the robot will start to turn. When the motors are commanded to (0.7,1) the robot continues in a straight line.



(a) Ramp down to 0.5 **(b)** Ramp down to 0.6 **(c)** Ramp down to 0.7

Figure 10.13: Differential drive test results. Each plot shows the position of the robot as the left motor output is ramped down, with the robot starting at (or close to) the origin. The green dots show the path of the simulated robot. The red and blue dots show the path of the real robot, smoothed and unsmoothed respectively. The text shows the motor output at a subset of points (marked with black dots) in the smoothed runs.

This may be a fault with the elderly Pioneer robot used for the experiments: the robot uses a rear caster wheel, which, if stiff, would cause this problem. Alternatively the PID control of the motors may be at fault. However, it serves here as an example of the "reality gap": differences between the simulation and reality will cause the robot to behave in grossly different ways. Here, the output blending network generates outputs which should produce gradual turns, but the physical robot simply ignores these.

10.2.2.6 Summary

In summary, output blending performs fairly well in the robot if given sufficient power to work with. In simulation it is conservative, due to the constant presence of *phototaxis*. This behaviour does not carry over onto the robot, however: the differential drive problem causes the robot to ignore gradually increasing turns, following a straight line until the charge is significantly lower than the maximum. This causes many robot runs to fail or approach dangerously low charge levels, but helps it explore.

Another instance of the "reality gap" causing problems is south run 1 at $k_{power} = 0.003$, where the robot simply stops while facing the mesh at full charge. This appears to be a failure to generalise: the network has been presented with a situation which

is sufficiently different from those for which it has been trained to cause it to generate an output which does not follow the original controller-based rules.

10.2.3 h-as-input, best network

The results for the best h-as-input network are shown in Figs. 10.14 (for $k_{power} = 0.0025$), the lower charging level) and 10.15 ($k_{power} = 0.003$). Only two robot experiments were performed at each power level because the network performed so badly on the robot that it caused damage to the arena bounding mesh, and was in danger of damaging itself. However, this network performed better than both UESMANN and output blending in the simple simulator experiments (See Fig. 9.11, noting the performance of network 8 of h-as-input).

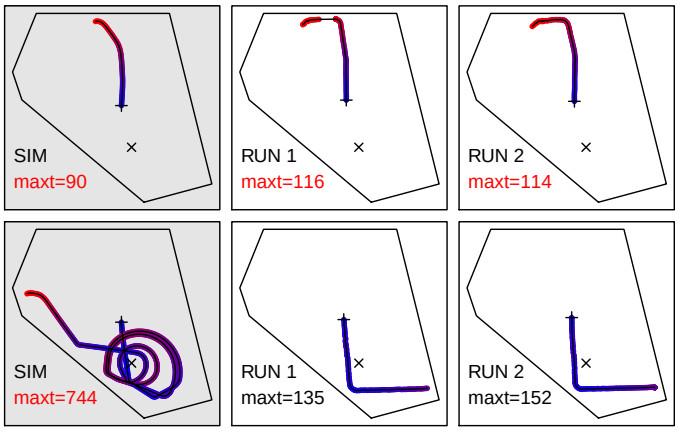


Figure 10.14: h-as-input best network runs at $k_{power} = 0.0025$.

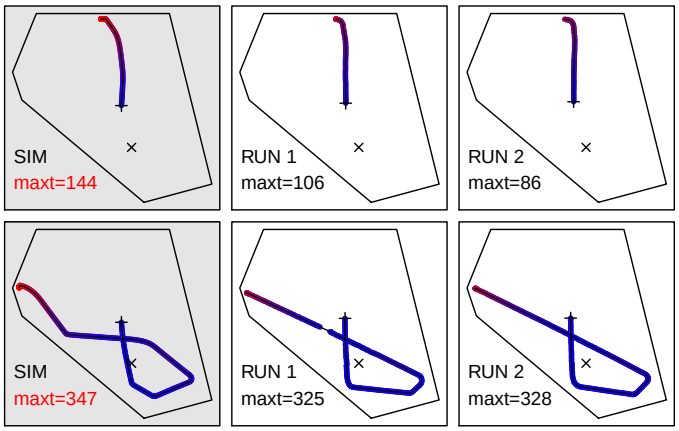
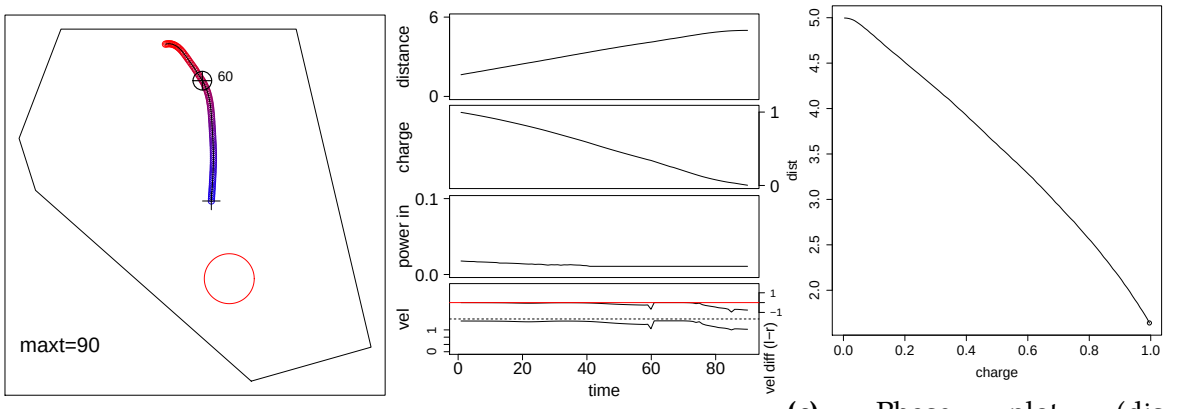


Figure 10.15: h-as-input best network runs at $k_{power} = 0.003$.

No run survives for 1000s, although the simulated south-facing run at $k_{power} = 0.0025$ performs several cycles before turning into the dark area and failing to return. Robot runs either turn too late or not at all (whereupon the experiment was aborted).

10.2.3.1 Runs at $k_{power} = 0.0025$

The north-facing simulation and second robot run are shown in more detail in Figs. 10.16 and 10.17 (the second was chosen because it has an uninterrupted position trace). In simulation, the robot shows very little *phototaxis* until the charge is about 0.5. At this point it begins to turn. However, at $t = 60$ there is a sudden return to straight driving (i.e. *exploration*): the robot continues to drive straight until it is almost at the north wall. Here it turns sharply, consistent with both behaviours, but runs out of charge.



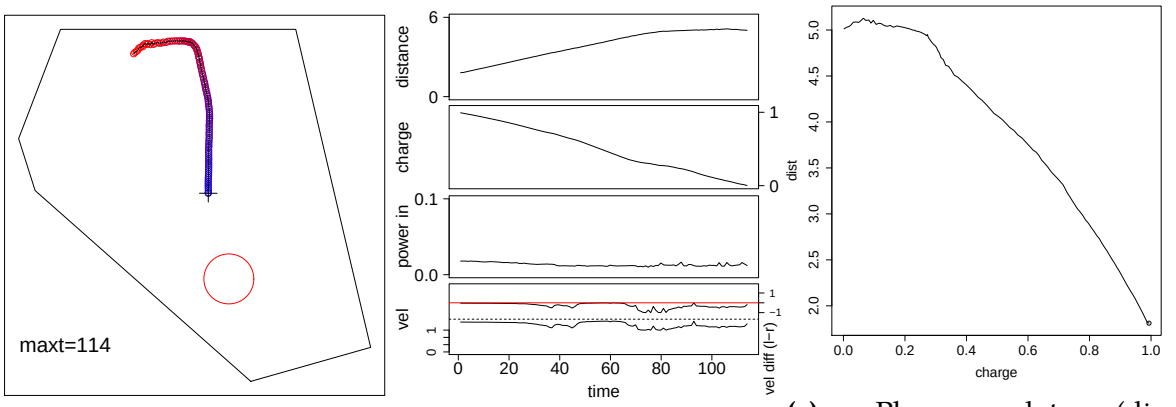
(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.16: Simulated north run of h-as-input best network, $k_{power} = 0.0025$. Here, the point at time 60s is labelled in the position plot: see the text for details.

The first part of this behaviour is consistent with the plot in Fig. 9.28: *exploration* is the dominant behaviour until $h = 0.6$ (and charge is 0.4), whereupon a rapid transition to *phototaxis* occurs. This can also be seen in the phase plot for the simple simulation in the simple test arena, Fig. 9.15a. However, the return to exploration (or at least straight driving) at $t = 60$ is more difficult to explain.

The robot run in Fig. 10.17 shows similar behaviour: a turn followed by a straight, followed by a turn which follows the wall. Towards the end of the run there is a short period of *phototaxis*, but far too short to enable a return to the emitter before the simulated battery discharged.

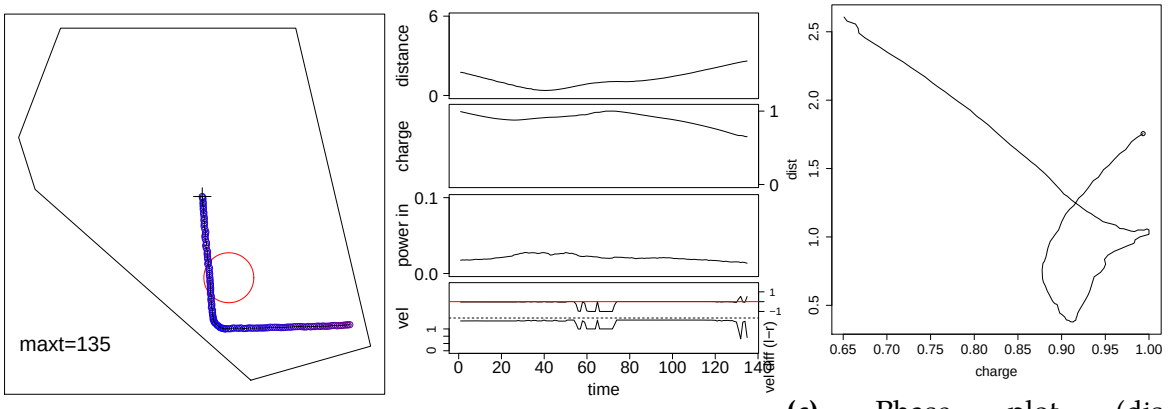
In the simulated south-facing run the robot orbits around the emitter, aided by the geometry of the arena. The orbits are tight, which suggests that *phototaxis* is appearing while the modulator is still low, in contrast to the north-facing results. The robot finally leaves the emitter with a high charge and begins exploring. However, now the *phototaxis* behaviour does not begin until the robot is unable to return.



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.17: Robot run 2 (north) of h-as-input best network, $k_{power} = 0.0025$.

South-facing run 1 is shown in Fig. 10.18. The robot shows no *phototaxis* as expected due to the high charge and the nature of the transition (again, see Fig. 9.15a), but also shows considerable flaws in *exploration*: the run was aborted at 135s because the robot drove into the mesh wall, although it was turning intermittently as it did so. The same behaviour occurred in the other run, and no more runs were performed to avoid damage.



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

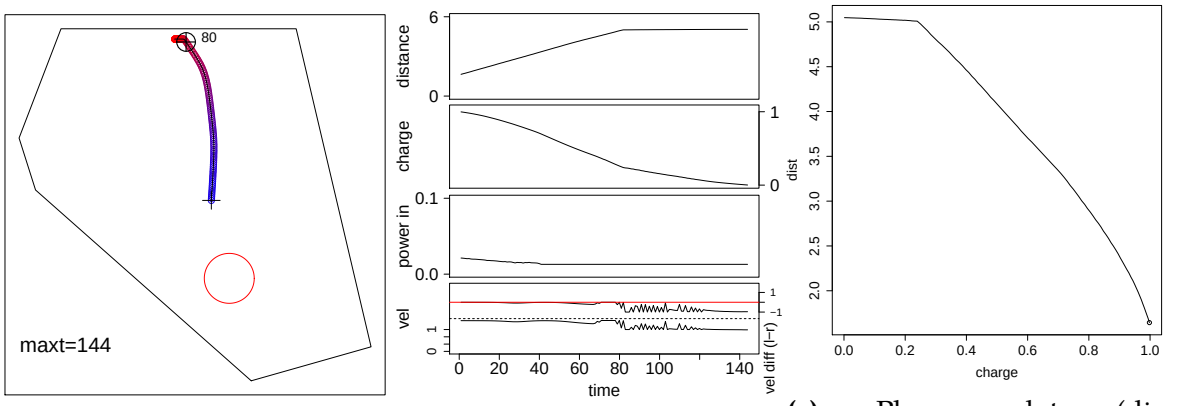
Figure 10.18: Robot run 1 (south) of h-as-input best network, $k_{power} = 0.0025$.

10.2.3.2 Runs at $k_{power} = 0.003$

Here, the simulation performed similarly to that at $k_{power} = 0.0025$, although the details are a little different, as shown in Fig. 10.19. There is a small amount of *phototaxis*, which abruptly stops at $t = 70$ in a similar manner to the previous simulation — this is much smaller, however, because of the higher power availability. At $t = 80$, the

robot begins to turn sharply away from the wall, but this turn is intermittent as can be seen from the velocity difference plot. This turn could be *phototaxis* or *exploration*, both behaviours are appropriate here. Following this the robot tries to turn sharply and runs out of charge.

The north-facing robot experiments show similar behaviour to the simulation, with the turns so late that the robot hit the mesh and the runs were aborted.



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.19: Simulated north run of h-as-input best network, $k_{power} = 0.003$. Here, the point at time 80s is labelled in the position plot: see the text for details.

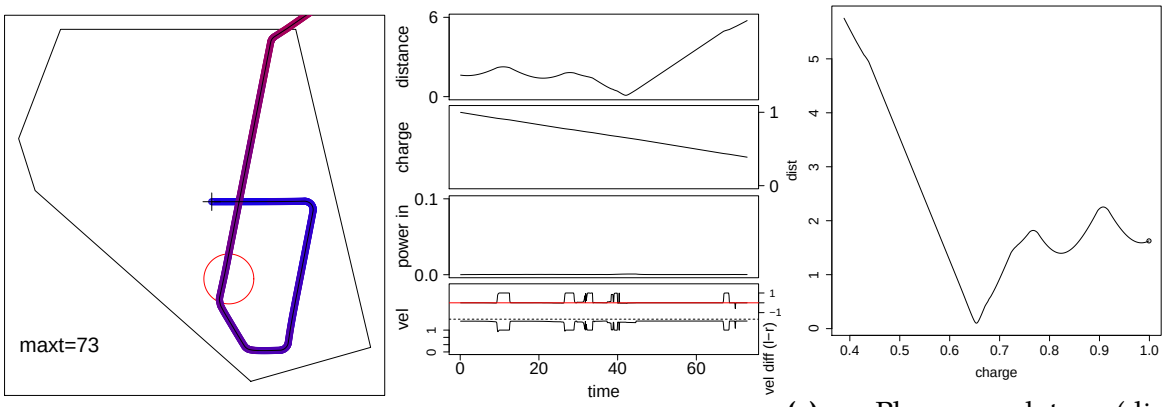
The south-facing simulation is similar to that at $k_{power} = 0.0025$, although the higher power level permitted the robot to explore without cycling around the emitter. The run proceeds in a similar fashion with a run into the dark part of the arena. It ends with an similar attempt to turn to that of the north-facing simulation, during which the robot runs out of charge. The robot experiments are similar to this until the end, where they were aborted because the robot failed to turn and hit the mesh wall.

10.2.3.3 Collisions

The best h-as-input network clearly has some issues. The most obvious of these is that turns to avoid the wall either happen too late or not at all, a behaviour not seen in the simple simulation using the simple test arena. However, this behaviour can be reproduced if the simple simulation is used with a map based on the robot arena, as shown in Fig. 10.20.

There are three, possibly interrelated, causes of this problem:

- Collisions occur when the charge is below the maximum i.e. when the network is in a transitional state, reasonably far from the extrema which should



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.20: Simple simulator running the best h-as-input network on a simulacrum of the final arena.

perform either *phototaxis* or *exploration*. It may be that the transitional state is compromised, such that it neither performs phototactic turns nor obstacle avoidance.

- It may be that the network has overfitted on the training data, causing it to fail when presented with certain combinations of light, sonar and modulator input which were not represented in the data.
- Additionally, this network was selected partially because it achieved a high weighted distance score, which may be because it only responds to very close sonar echoes. This constraint, applied to the networks post-training, may have selected a network whose sonar component is marginal.

10.2.3.4 Behaviour changes

The simulated south run at $k_{power} = 0.003$ shows a tight cycling around the emitter at the start of the run, which would appear to indicate *phototaxis* begin performed. This is unexpected, because the charge is moderately high. However, towards the end of the run the robot is heading straight, while the charge is low, which appears to indicate *exploration*. A similar unexpected relationship between modulator and behaviour occurs in Fig. 10.16, the simulated north run. Here, the robot performs a little *phototaxis* but switches back to a straight line when the charge falls further.

Thus the relationship between modulator and behaviour appears to have some undesirable features, including a strong non-monotonicity. Some non-monotonicity is expected for both h-as-input and UESMANN (see Fig. 9.28) but the result here is to cause a nearly discharged robot to appear to return to *exploration*.

Some changes may also be due to the network learning stronger weights for some sensors than others due to irregularities in the training set. If the weight for a particular light sensor is low, when *phototaxis* rotates the robot such that this "weak" sensor is directly in line with the emitter, the response will decrease. This would give the appearance of a return to *exploration*.

10.2.3.5 Summary

While it performs well in the simple square testing arena, this particular h-as-input network fails to perform in the final arena, either in simulation or on the robot. The performance is worse than output blending, against which this network type scored favourably during testing. The network does not deal well with the final arena's more complicated geometry, possibly due to overfitting to the small, square training arena.

Additionally, there is an apparent tendency to return to *exploration* at low charge levels (or at least to drive without turning back to the light), which is undesirable but possibly resulted in a high metric at the test stage, which has a less stringent power regime. As stated above, this may be due to overfitting, a flawed transitional behaviour, or perhaps the effect of different weight strengths from the light sensors into the hidden layer.

On the robot the problems appear to be worse, which may be due to the inputs being different from those encountered during training. Only two robot runs were made in each direction, due to all runs resulting in emergency stops.

10.2.4 h-as-input, second-best network

As has been noted, the best h-as-input network may have gained an advantage through behaviours which do not perform well on the actual robot. The second-best network according to the combined metric was therefore run in the same way, to determine whether it suffers from the same problems. The runs are shown in Figs. 10.14 and 10.15.

Fig. 9.11 (p. 258) shows that both networks achieve homeostasis in the simple simulator within the test arena (at least for the length of the test run), but that the second-best network (6) has a lower score than the best (8) in the edge-weighted travel and distance variation metrics. This is because it did not stop long enough at the emitter before resuming exploration (see Sec. 9.4.2.3, p. 259).

We would therefore expect the second-best network to be more conservative than the best, with *phototaxis* occurring when the charge is higher, and this is indeed what

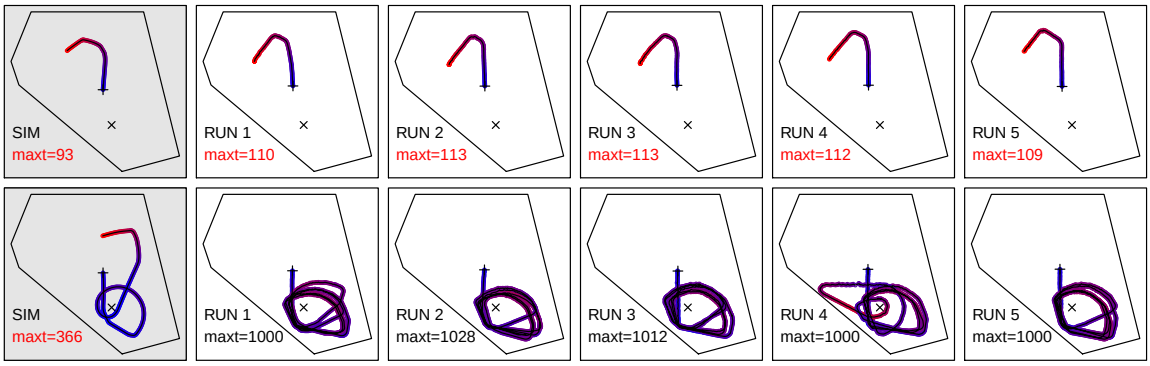


Figure 10.21: h-as-input second best network runs at $k_{power} = 0.0025$.

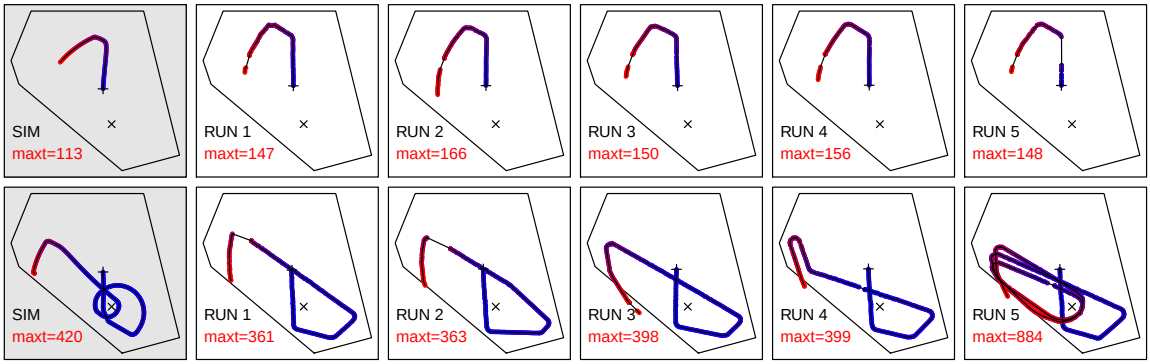
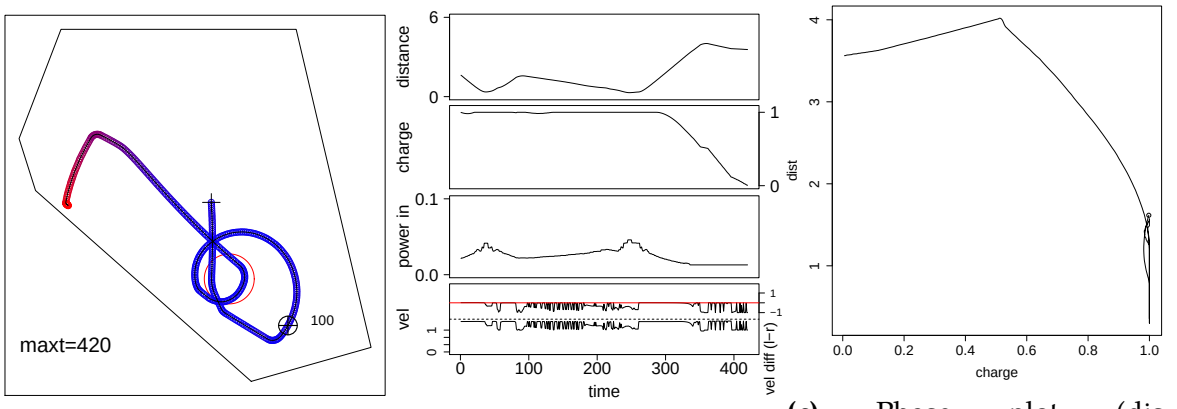


Figure 10.22: h-as-input second best network runs at $k_{power} = 0.003$.

the plots show. This is particularly clear at the lower power level, where the north-facing runs turn earlier (but still too late) and the south-facing runs complete the experiment, at least on the robot.

The behaviour of the simulator in south-facing runs at both power levels is problematic, with a large amount of *phototaxis* while the charge is still very high. This also occurs to a lesser extent in the south-facing low power robot runs. As can be seen in the velocity plot of Fig. 10.23b, the phototactic loop starting at $t = 100$s in the simulated high-power south run occurs while the charge is close to or at the maximum, and takes the form of intermittent spikes in the velocity difference throughout the turn. This suggests a similar problem to the behavioural shift in the best network: while that network showed inappropriate *exploration* at high modulator levels, this network sometimes has inappropriate *phototaxis* at low modulator levels[2]. It seems unlikely that different weights from the light sensors into the network would cause this effect, because there should be no effect from the light sensors at a low modulator (high charge) level.

[2]This is a generalisation, in that the best network also demonstrated inappropriate *phototaxis* at low modulator levels (in the south simulator run at $k_{power} = 0.0025$).



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.23: Simulated south run of h-as-input second-best network, $k_{power} = 0.003$. Here, the point at time 100s is labelled in the position plot: see the text for details.

While this particular behaviour is not apparent in other runs, there is still some clear non-monotonicity in the behaviour as the modulator varies. For example, all north-facing runs show three straight segments with two turns, both well away from the walls. These suggest *exploration* with two brief forays into *phototaxis*.

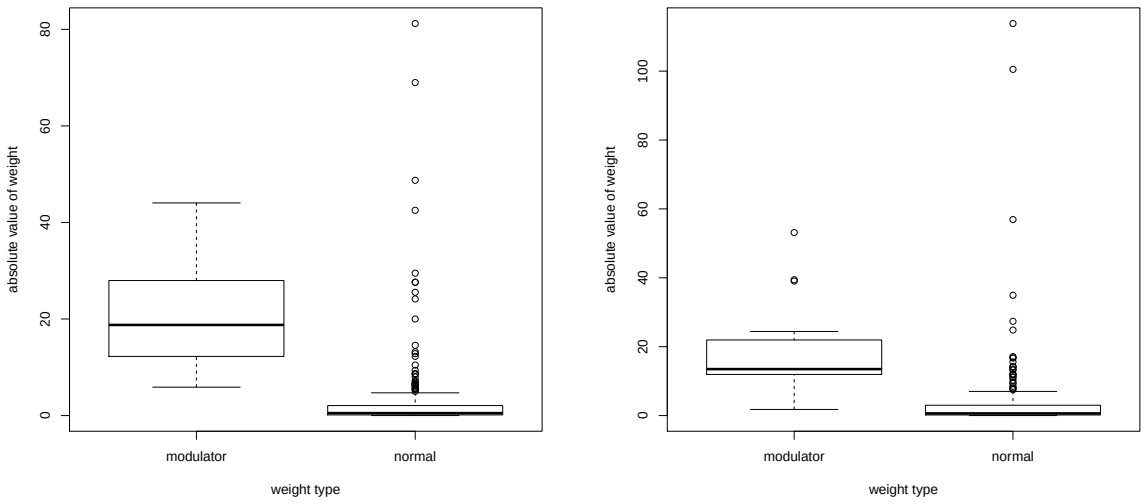
10.2.4.1 Summary

Both h-as-input networks appear to suffer from a tendency to use the wrong behaviour for a given modulator level in an unpredictable way. In the second-best network, this manifests as a tendency to perform *phototaxis* when the modulator is low (seen in the two simulator runs), while in the best network, there is the opposite tendency to drop back to *exploration* when the modulator is high — although inappropriate *phototaxis* also appears in the south simulator run at $k_{power} = 0.003$. The second-best network is general is conservative, suggesting that *phototaxis* is performed over the greater part of the modulator range.

Although only two networks have been studied, it is possible that this pattern extends over all the h-as-input networks. One possibility is that these networks are trained using a single input for the modulator, and thus any signal from the modulator is just another input. Therefore, if the network is given an input stimulus very close to a set of training examples except for the modulator, it may produce an output close to those for the examples, if few examples with the modulator at the given value exist. If this were the case, it could also be the cause of the collisions in both networks. Consider a situation where the modulator is high (i.e. *phototaxis* should be performed), but the network contains examples similar to the current

inputs in which the modulator was low. In this case, outputs similar to those for the examples at the low modulator level might be generated.

This is unlikely, however: the modulator input to the network has a large effect on the behaviour, so it should have a much larger magnitude in its weights going into the hidden layer. To confirm this, plots were made of the magnitudes of the modulator weights compared with the other weights in both best and second-best networks, as shown in Fig. 10.24. These show that the modulator weights are much larger, by almost an order of magnitude. Because of this, the modulator's effect across the hidden layer will be stronger than the effect of other inputs.



(a) Best network **(b)** Second-best network

Figure 10.24: Box plots of the magnitudes of weights entering the hidden layer from the modulator input and from other inputs.

Alternatively, it may be that the transitional behaviour is such that a "blend" of the two behaviours is present, in which the robot drives forwards (as in *exploration*) but ignores the sonar data (as in *phototaxis*). This problem appears in both the best and second-best networks (as can be seen from the south-facing runs of the second-best network at $k_{power} = 0.003$ in Fig. 10.15). Given the conservative nature of the latter network, this would discount the possibility discussed in Sec. 10.2.3.3 that marginal sonar networks were selected from the test set.

10.2.5 UESMANN

The results for the best UESMANN network according to the combined metric are shown in Figs. 10.25 and 10.26. Our first impression is of comparable performance to output blending, with the exception of most north-facing UESMANN runs at $k_{power} = 0.003$ which run out of charge. UESMANN shows more variation across the runs, with long runs following different paths.

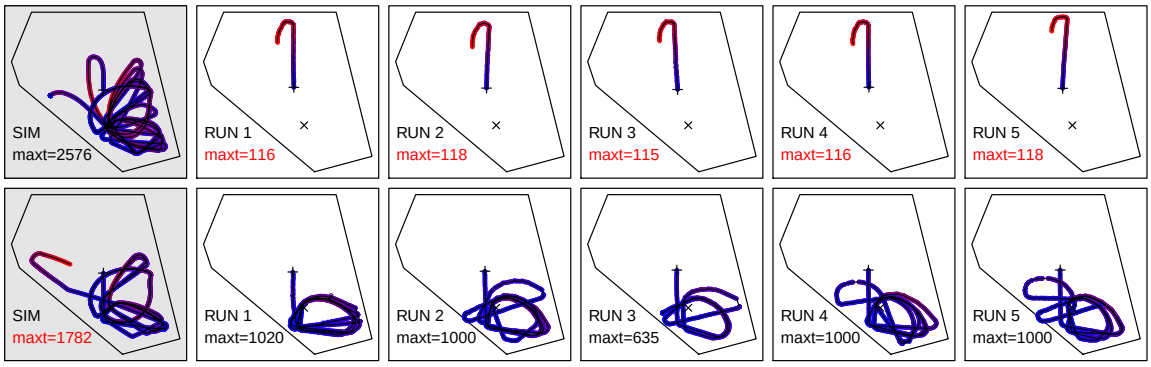


Figure 10.25: UESMANN best network runs at $k_{power} = 0.0025$. South-facing run 3 stops to face the mesh indefinitely.

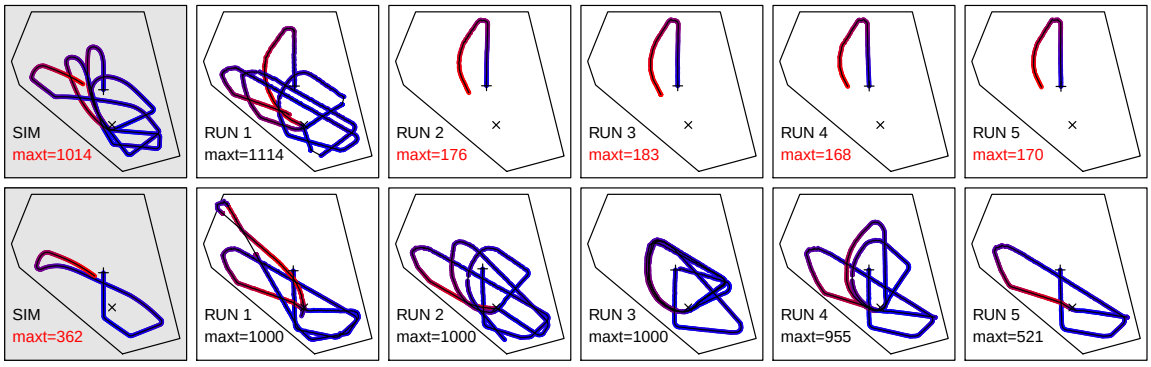


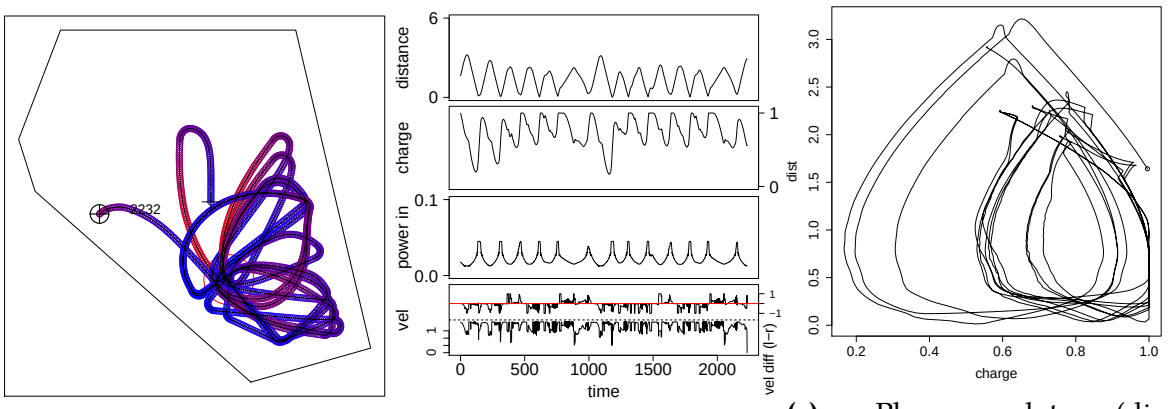
Figure 10.26: UESMANN best network runs at $k_{power} = 0.003$. South-facing runs 4 and 5 aborted due to wall collision.

10.2.5.1 Runs at $k_{power} = 0.0025$

In general, performance is more conservative than output blending but similar in form, with the phase plots showing similar loops (compare Fig. 10.27c and Fig. 10.9c). However, these loops have more of the characteristics of Fig. 9.25, with a triangular shape, than the more circular Fig. 9.26.

The north-facing simulated run at this power level is shown in Fig. 10.27, which performs considerably better than its output blending counterpart in Fig. 10.7 (which

fails after 110s). However, the run comes to a dead stop at full charge, facing the wall at around 2230 seconds. Note that this is well after the normal 1000s termination time for the robot experiments.

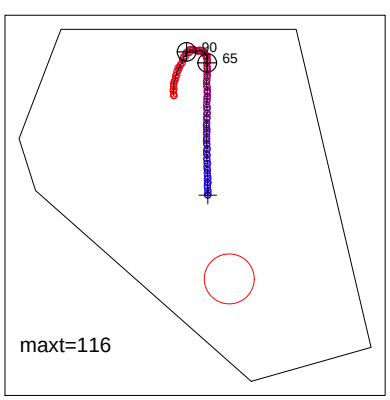


(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

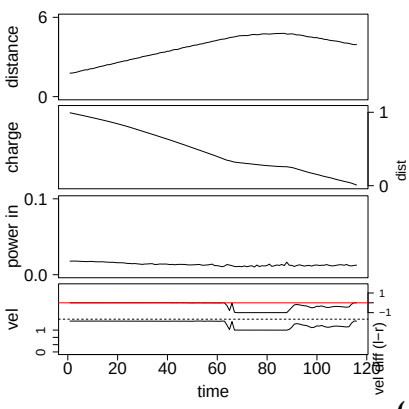
Figure 10.27: Simulated north run of UESMANN best network, $k_{power} = 0.0025$. In this plot the time of the stop is shown — the run continued until 3000s with the robot stationary..

Of the north-facing robot runs, all fail with the robot again turning too late. Run 1 is shown in Fig. 10.28. At 65s, the robot performs a strong turn, which could be caused either by *phototaxis* or *exploration*. However, subsequent runs with turns further from the wall occur at a similar or higher charge level and have a similar character (e.g. Fig 10.30), suggesting this is a *phototaxis* turn. At 90s the robot stops turning sharply, and begins to curve gently towards the emitter. This suggests firstly that the transition into *phototaxis* is crisp at $h \approx 0.7$ (as can be determined from the charge plot), and secondly that the phototactic behaviour is less marked than might be expected when the angle to the emitter is small. Indeed, the velocity difference plot shows considerable variation during this turn, where the *phototaxis* controller should give a gradual reduction in difference. This is likely due to the modulator continuing to rise through a non-monotonic transition.

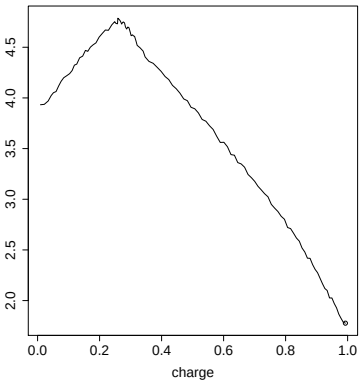
The south-facing simulator run shows a similar set of loops to its north-facing counterpart when constrained by the walls, but upon escaping the robot again moves into *phototaxis* too late and fails to return. All the robot runs behave similarly, looping conservatively around the narrow lit area, and most complete the experiment. It is possible that given more time, these would eventually suffer a similar fate to the simulator run. Run 3 behaves similarly to the north-facing simulator run, stopping dead facing the wall at full charge. This is shown in Fig. 10.29. While difficult to see in the plot, this happens at roughly the same place in the arena as the simulator run.



(a) Position plot

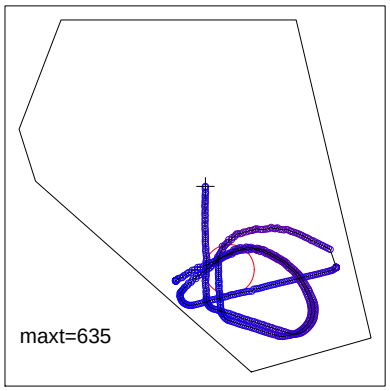


(b) Variable plot

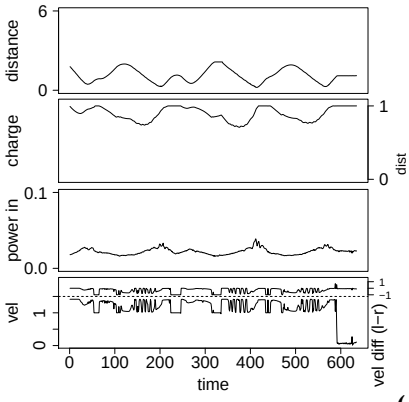


(c) Phase plot (distance/charge)

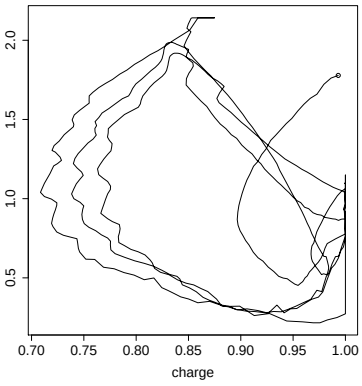
Figure 10.28: Robot run 1 (north) of UESMANN best network, $k_{power} = 0.0025$. Here, the points at times 65s and 90s are labelled in the position plot: see the text for details.



(a) Position plot



(b) Variable plot



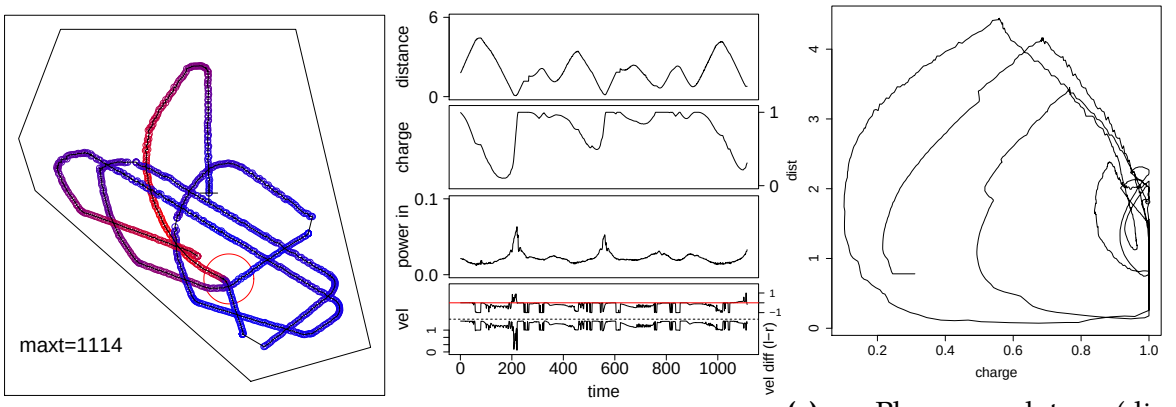
(c) Phase plot (distance/charge)

Figure 10.29: Robot run 3 (south) of UESMANN best network, $k_{power} = 0.0025$.

10.2.5.2 Runs at $k_{power} = 0.003$

At the higher power level, we see the expected behaviour: the loops are more open, and the robot travels further. The simulated run shows the common failure mode of a long "north-west" run which travels too far to return safely; this does not occur at the lower power level because *phototaxis* keeps the robot within the narrow area. All north-facing robot runs fail in a similar manner to the lower power level runs, except run 1, which is shown in Fig. 10.30.

Here, the robot turns slightly earlier and so has enough power to return to the emitter, falling into a loop of long runs. Note the general similarity of the shape of the phase plot with the simulated run at the higher power level in Fig. 10.7.



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

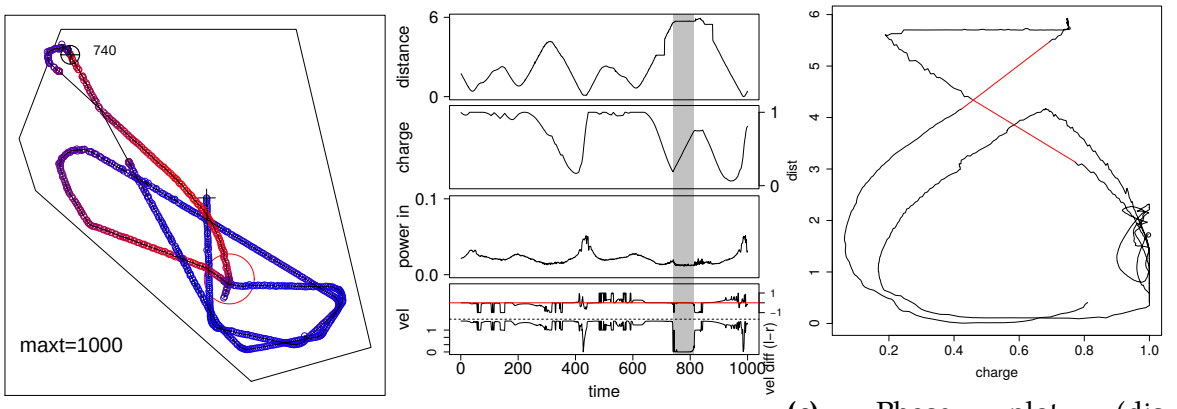
Figure 10.30: Robot run 1 (north) of UESMANN best network, $k_{power} = 0.003$.

The south-facing simulator run fails with the robot falling immediately into a north-west run. While there is an attempt at phototaxis towards the end, the robot is bounced away from the wall (showing enough *exploration* to do so), taking it further into danger. Runs 1-3 complete the experiment. Runs 4 and 5, however, do not turn away from the wall. In both these cases the experiment was halted.

Run 1 is an interesting case, and is shown in Fig. 10.31. In this run, the robot travels north-west towards the far corner of the arena — this single traverse is close to the longest possible. In most cases, this should fail. However, at 740s, the robot stops to recharge for nearly 100s, performs a tight turn, and returns to the emitter. Although the robot is performing *phototaxis* at the stop, it is not receiving the stimulus for which stopping is the correct response: a large amount of light. Nevertheless, this "error" has a beneficial effect.

10.2.5.3 Behaviour changes

UESMANN consistently fails to turn until too late on north-facing runs, which appears to be due to a transition to *phototaxis* at $h \approx 0.7$, as can be seen in the low power run of Fig. 10.28. This corresponds to a charge level of 0.3, which is too low to return to the emitter safely. However, as with h-as-input, the nature of the transition between the behaviours is complex. Consider the low power south-facing in runs in Fig. 10.25, which show a considerable amount of *phototaxis* at relatively low modulator (high charge) levels. Another example is Fig. 10.31, which shows a large amount of *exploration*, when *phototaxis* should have turned the robot. However, near the wall the robot moves into *phototaxis* (albeit the stopping behaviour).



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

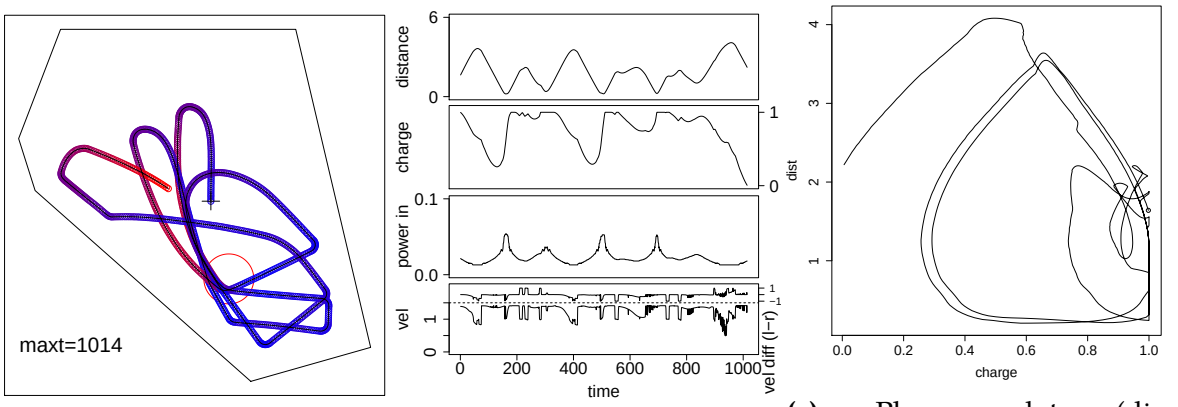
Figure 10.31: Robot run 1 (south) of UESMANN best network, $k_{power} = 0.003$. Here, the point at time 740s is labelled in the position plot, and the recharge which takes place after this time is marked by a grey rectangle in the variable plot: see the text for details. Note that there are discontinuities in the positional data due to the robot LED being obscured by the emitter lamp hood; these manifest as steps in the phase plot, and are marked in red.

10.2.5.4 Summary

In general, UESMANN performs comparably to output blending with fewer runs failing due to discharge, although it has two collisions and an unexplained stop in the robot runs. It performs considerably better than h-as-input, and shows a larger degree of variation in the runs than both methods. UESMANN appears to be more conservative than output blending, which causes more runs at $k_{power} = 0.0025$ to survive. However, the behaviour performed across the modulator range varies non-monotonically, as it does in h-as-input. This can be seen in all the north-facing runs, and is particularly clear in the successful run in Fig. 10.30. Both westernmost turns in this run show a distinct set of stages, with the robot turning sharply, proceeding perhaps 2m, then turning again. While this behaviour may be emphasised by the robot's problems with differential steer (Sec. 10.2.2.5), it is still evident to a lesser extent in the simulator runs: see the westernmost turn in Fig. 10.32 for an example.

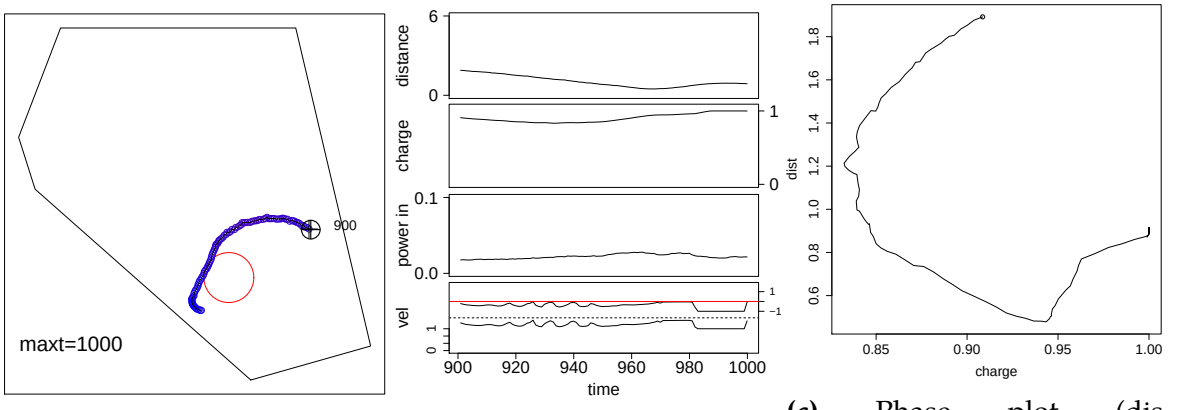
This non-monotonicity is less problematic than in h-as-input, which contributes to its success compared to both h-as-input networks. However, like that network, it sometimes turns too late (such as in the north-facing runs) and sometimes too early. Compare Fig. 10.33, which shows *phototaxis* too early, with Fig. 10.34, which shows *phototaxis* too late (or rather, not at all).

In the first case, the robot is showing a wheel velocity difference while the charge is still at around 0.8, while in the second case the charge falls to around 0.3 without any velocity difference.



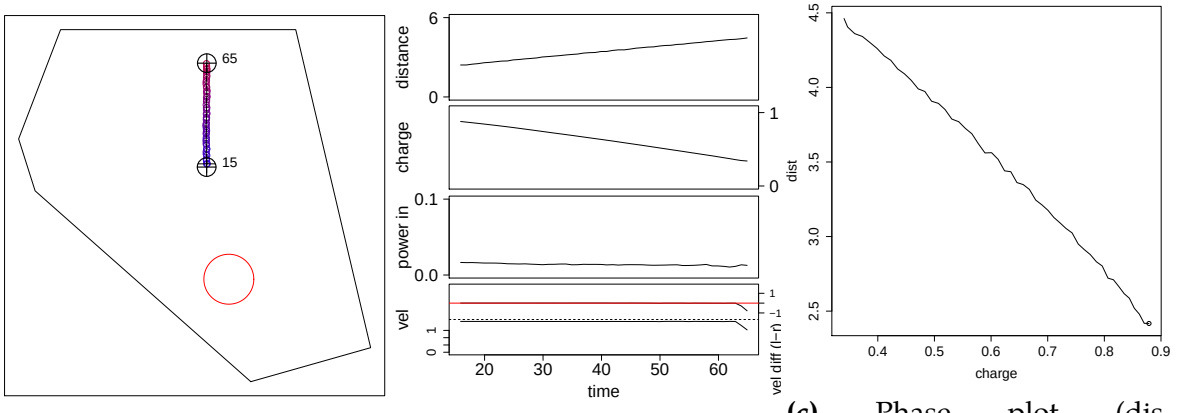
(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.32: Simulated north run of UESMANN, $k_{power} = 0.003$.



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.33: Robot run 2 (south) of UESMANN at $k_{power} = 0.0025$, $t > 900$.



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 10.34: Robot run 1 (north) of UESMANN at $k_{power} = 0.0025$, $15 < t < 65$.

Like h-as-input, UESMANN suffers from collision problems, although these are only seen twice and in similar circumstances (south-facing runs 4 and 5 at $k_{power} = 0.003$). Here, this occurs during *exploration* with the robot at high charge, near a distinctive corner of the arena unlike any part of the training arena. Another failure mode is a dead stop facing the mesh, seen twice: once in simulation and once in a robot run. These occurred at nearly the same place in the arena, facing south-west into a wall. This is similar to the stopping behaviour seen in output blending (Sec. 10.2.2.3, p. 286), and happens at roughly the same place in the arena. This suggests that the problem lies in the training data, or rather the system producing the training data (since each network is trained from a separately generated set of examples).

Chapter 11

Conclusions

The robot experiments show that all three network architectures are able to learn a simple homeostatic robot task, given examples of simple controllers performing two behaviours at different modulator (and therefore charge) levels. This is true even when the simulator from which the examples were generated has simple sensors and actuators, which behave rather differently from reality, and when the simulation arena is different from the final arena.

11.1 Quantitative differences

Before summarising the qualitative differences described in detail above, we will show some brief quantitative analyses of the robot runs. The simplest possible metric for these experiments is how many survive the arbitrary time limited selected during the experimental design. The results are shown in Fig. 11.1, which appears to show that UESMANN performs "best", in that it has the highest number of surviving runs. Another simple metric is survival time, which is shown in Table 11.2. This

Table 11.1: Number of robot runs for each network type and power regime which survive at least 1000s without stopping indefinitely.

Network	good runs ($k_{power} = 0.0025$)	good runs ($k_{power} = 0.003$)	total
output blending	1	**5**	6
h-as-input best	0	0	0
h-as-input second-best	**5**	0	5
UESMANN	4	4	**8**

shows a high survival time for output blending, with UESMANN a close second. The discrepancy with the previous table is due to some output blending networks surviving for some time before failing, while some UESMANN runs fail quickly.

Table 11.2: Total survival times of all five robot runs for each network and power regime, with each run capped at 1000s. All figures rounded to the nearest second.

Network	total time ($k_{power} = 0.0025$)	total time ($k_{power} = 0.003$)	total
output blending	4119	**7668**	**11786**
h-as-input best	519	846	1364
h-as-input second-best	**5559**	3175	8733
UESMANN	5220	6175	11394

While output blending tends to survive at high power, it does not perform as well as UESMANN at low power due to the latter network's more conservative behaviour. At low power, the second-best h-as-input network appears to perform well, but this also is achieved by extremely conservative behaviour. At a higher power level, however, h-as-input does not survive long. A final obvious metric is the combined metric used to select the networks for these experiments from those used in the simple simulator. The results are shown in Fig. 11.1. This clearly shows the difference between north and south runs, with most north runs performing poorly, except in output blending. However, the overall picture shows no statistically significant differences between the three scores ($p > 0.05$, Wilcoxon rank-sum test).

Fig 11.2 shows the same data with the north and south facing runs separated. In this analysis, output blending is significantly better than h-as-input in north-facing runs, and UESMANN is significantly better than output blending in south-facing runs ($p < 0.05$, Wilcoxon rank-sum test). This is mainly due to differences in survival times, as can be seen from visual inspection of the runs.

11.2 Qualitative differences

Detailed descriptions of how each network behaves have been given above at some length, with each network producing a complex set of behaviours. We will attempt, however, to summarise the differences between them.

It was predicted that output blending would survive in the real robot but not explore the arena well because of the constant presence of phototaxis at all modulator levels except zero. This was not the case, because the robot did not turn until the difference between the motor outputs was large (see Sec. 10.2.2.5). In fact, output blending explored further than the other network types, with the long, slowly increasing curves generated by the motor outputs being realised as long straight runs, until a threshold in the motor output difference was reached. However, it is possible

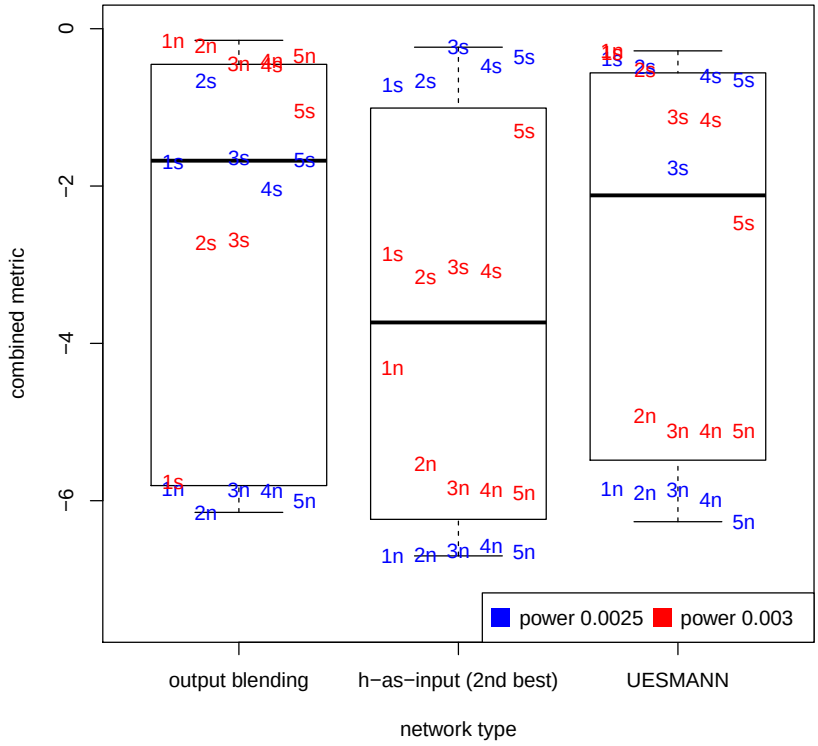


Figure 11.1: Combined metric for all robot runs. Each run is shown numbered with "s" or "n" indicating whether it is a "north" or "south" facing run. The few runs made for the best *h*-as-input network have been omitted.

that this good performance is a consequence of the peculiarities of this particular robot's steering mechanism.

Output blending was also less prone to the other problems suffered by the networks, notably random stops and collisions with the mesh (although one random stop did occur). Because of this, it has the longest survival time at high power: long runs in other networks often ended in a stop or collision. This may be due to each behaviour being isolated to a single network, resulting in less "crosstalk" between the behaviours.

This may be a particular problem in *h*-as-input, which seems to show a strong tendency to use the wrong behaviour for a given modulator level, and to sometimes perform only aspects of that behaviour. For example, the best network drives straight into walls at high modulator levels: a behaviour consistent with elements of *phototaxis* and *exploration* being mixed, showing the straight driving of the latter and the "sonar blindness" of the former. Aside from crosstalk, it may simply be that the training data does not contain examples for this situation given the different shapes of the training

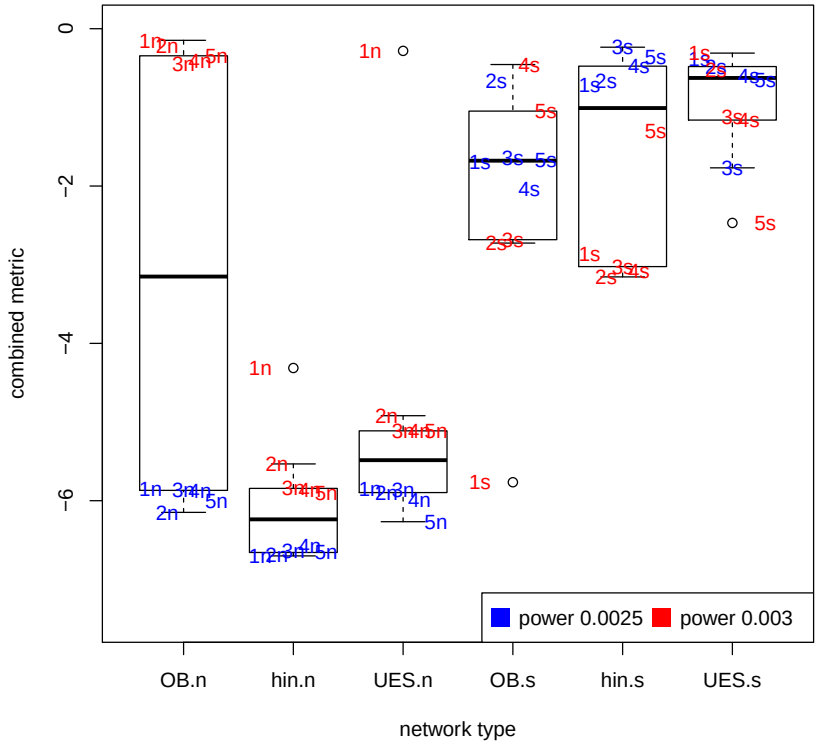


Figure 11.2: Combined metric for all robot runs, with separate box plots for north and south. Each run is shown numbered with "s" or "n" indicating whether it is a "north" or "south" facing run. The few runs made for the best *h*-as-input network have been omitted.

and final arenas. In this situation, the network may produce a "corrupt" version of a correct behaviour. More work is needed to study the cause of the problems: if and how crosstalk occurs, and how limitations in training data affect the network.

When *h*-as-input works — as it generally does in the second-best network — it appears to be more conservative than output blending, which can be seen particularly clearly in the $k_{power} = 0.0025$ runs. The transition itself is non-monotonic and complex, with *phototaxis* sometimes occurring at high charge (low modulator) levels, and *exploration* at low charge (high modulator), depending (as it must) on the input. Despite this, the behaviour is quite consistent, with the robot traversing roughly the same pattern in each group of runs.

In UESMANN, however, different runs from the same experiment can follow very different paths. There is also less of a tendency for the robot to settle into a repeating loop — something commonly seen in other network types. It seems likely that UESMANN is more sensitive to differences in the inputs and modulator, perhaps

with the response these being less smooth at small scales. Certainly Fig. 9.28 suggests that UESMANN is sensitive to changes in the modulator at all but the highest values.

Thus it may be that UESMANN demonstrates a form of sensitive dependence on initial conditions, given that a slight variation in the course will change the light input and affect the modulator, which will cause a change in behaviour, which will affect the course. If so, this may be useful in an exploratory robot; UESMANN covers more of the arena from different angles. Naturally, because it finds more situations in which to fail, it may also appear to fail more often.

UESMANN's failures consist of discharge (although these occur a little less than in output blending), two collisions which occur at roughly the same place in the arena, and a single dead stop at full charge. The discharging runs are all north-facing, and UESMANN has fewer than the other network types. The other failures may be caused by deficiencies in the training data, since they all occur in parts of the arena unlike the training arena. It is possible that the small, square training arena does not provide examples for this situation, but insufficient resources were available to repeat the experiments with modified training data.

The nature of the transition in UESMANN is complex, and perhaps rather more complex than *h*-as-input given the possible sensitive dependence. For the paths to diverge from the same approximate initial conditions as much as they do, it is likely that small variations in the modulator cause larger differences in the motor outputs than they do in the other networks. Indeed, the behaviour of the system may be truly chaotic, and future work might look at measuring the Lyapunov exponent spectra [304] of a large number of runs to determine if this is the case (see Sec. 12.5.2.1, p. 329).

11.3 Limitations in the training data?

We have mentioned above that some problems, particularly with *h*-as-input and UESMANN, may be due to limitations in the training data which arise from the nature of the training arena. The training and final arenas are shown in Fig. 11.3. They are roughly the same size, but the training arena is a simple square, so all angles are right angles. All corners will therefore present identical sonar profiles when viewed from equivalent relative locations. In contrast, the final arena is a more complex shape where each corner's profile is unique. Additionally, the final arena is much more open than the training arena, which tends to produce much closer sonar echoes (however, the simple test arena generates more distant echoes than than either the training or final arenas and does not suffer from sonar problems to

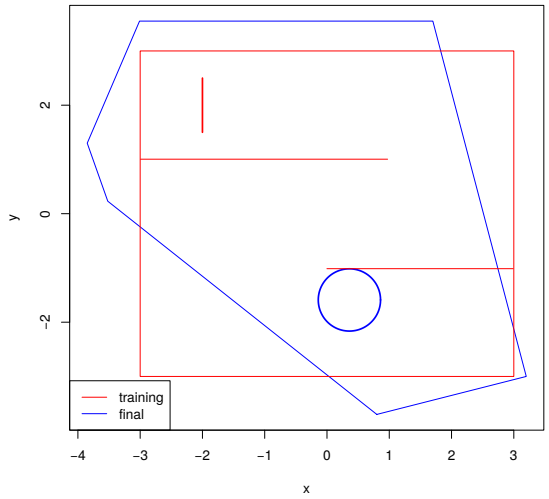


Figure 11.3: The training and final arenas, with the emitter positions and sizes (the final arena shows the approximate size of the pool of light cast by the emitter lamp).

the same extent). Finally, the emitters are differently positioned and are different in nature: the training emitter is linear and crisp, while the final emitter is circular and diffuse. It is likely to be the case that the patterns generated in the input vector by the emitter are different in the two arenas, despite some care being taken to ensure the robot's light sensor behaved like the virtual sensor.

Thus it is possible that there will be input vectors generated in the robot (and Gazebo simulation) which are not adequately represented in the training data, and may result in incorrect behaviour. Naturally, a neural network is intended to generalise from the training data to recognise unfamiliar inputs, but in some cases the input may simply be too unfamiliar.

This problem seems to manifest most in h-as-input, and to some extent in UESMANN. It is least apparent in output blending, although the single unexplained stop at full charge may be connected. The reasons for the different prevalences of the problem are not currently understood, but may be related to crosstalk from examples with similar inputs at different modulator levels, given that output blending separates the modulator levels into different networks.

Nevertheless, UESMANN is able to perform the behaviour in a real robot rather better than h-as-input and with comparable performance to output blending. The h-as-input network, despite working well in the simple simulation, does not deal well with the reality gap — even the relatively modest reality gap from the simple simulator to Gazebo. Output blending works well on the robot, but does so because

of the reality gap: the steering behaviour of the real robot counteracts the gradual nature of the behavioural transition.

11.4 Emergent behaviour

The behaviour which emerges from the system in the real world is interesting and perhaps useful. We note that output blending may only be able to perform well because of the physical peculiarities of the steering on this particular robot, but of more interest is the more complex emergent behaviour of the two other networks.

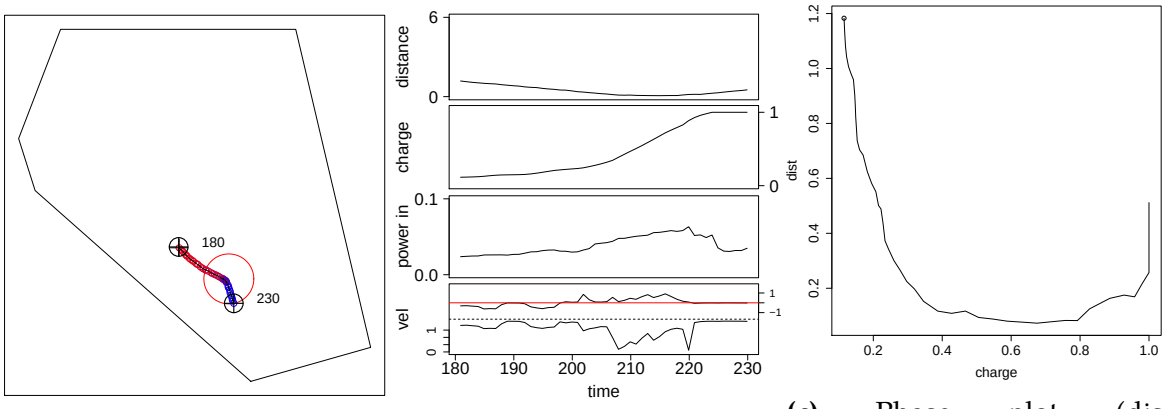
Both h-as-input and UESMANN have complex transition regions, which oscillate between the two trained behaviours as the modulator increases. In UESMANN's case, the non-monotonic sensitivity of the behaviour to modulator appears to be more finely-grained (from the apparent sensitivity to initial conditions described above), leading to a good deal more variation in the paths traversed. This is very useful behaviour in a robot designed to cover an area as much as possible, and in general may provide a useful source of variation. In h-as-input this behaviour is larger, with the "incorrect" behaviour being used more often, and appears less finely-grained, with the second-best network showing less variation in the paths traversed (although this may also be due to the tightness of the loops).

One behaviour which was expected to be helpful from the simple simulation runs in h-as-input and UESMANN is the extended stop at the emitter for recharging, seen in the phase diagram in Fig. 9.25 (p. 267) and shown in Figs. 9.15a and 9.20c. An example from the robot experiments is north-facing UESMANN run 1 at $k_{power} = 0.003$, a detail of which is shown in Fig. 11.4. This is exceptional, however: the robot rarely stopped at the emitter for any length of time, either having picked up a sufficient charge while travelling to the emitter or simply never venturing far in the first place, often being bounced back toward the emitter by the arena walls.

A final, minor example of interesting emergent behaviour is that of the run shown in Fig. 10.31, where the robot stops and recharges despite the low light values. This only occurs in a single run, but would be worth analysing in future work. However, it is possible that this behaviour, while helpful in this particular case, may be deleterious in others.

11.5 Issues with multiple training sets

Each network in Sec. 8.1.3 (p. 239) is trained from different initial weights and values, but also from a different set of training data generated by the simple simulator. This



(a) Position plot **(b)** Variable plot **(c)** Phase plot (distance/charge)

Figure 11.4: Robot run 1 (north) of UESMANN at $k_{power} = 0.003$, $180 \leq t \leq 230$.

was intended to provide an extra source of variety. If one set of training data were used, and it had problems, these would be reflected across all the networks. Using multiple training sets helps avoid this possibility: at least one network should have a reasonable set of training data and can be carried forward into the Gazebo and robot experiments. However, it does result in confounding factors: the success of the network depends on both the training data and the initial weights of the network.

The effect can be investigated by training a number of networks from different training data sets and different initial weights and biases. A possible procedure might be:

- Generate N sets of training data $T_1, T_2 \cdots T_N$
- For each set of training data, train M networks initialising the weights and biases by initialising the PRNG seeds to the values $S_1, S_2 \cdots S_M$.
- We now have $N \times M$ networks: one for each pairing of training data set and initial weights and biases.
- Make multiple runs of each network and record their performances according to the metric in Sec. 8.1.2.4 (p. 238).

If the networks with the same initial weights but trained using different data have very different performances, this would indicate a problem with the training data. All sets of generated training data should have representative data points for each possible interaction with the environment. If there is indeed a problem, it would also have ramifications for Sec. 11.3 because it would indicate clearly that the training data is inadequate, even within the simple training arena.

However, Fig. 9.4 (p. 250) and Fig. 9.11 (p. 258) (box plots showing the final mean squared error after training, and the combined metric on all trained networks respectively) appear to show clear differences between the three network types. This appears to indicate that randomly sampling networks from the 2D space of training data and initial parameters is sufficient to expose the differences between the network types.

If the experiments were to be repeated with a single training data set, this could be made much larger, and generated in a more methodical way than the current sets. One possibility is simply positioning the robot at all points in a regular grid within the training arena, and at a range of rotations, and recording the outputs from the rule-based controllers. Such a larger training set would impact the training times considerably.

Part V

Conclusion

Chapter 12

Discussion

12.1 A global, uniform neuromodulator

Our first research question was:

> Is it possible to build an extremely simple neuroendocrine system whose response to a neuromodulator (a global parameter) can be trained such that the network performs qualitatively different functions at different modulator levels?

UESMANN is such a system, inspired by the Neal/Timmis artificial endocrine system [209]. Like that system, it uses a simple multiplicative modulation of the weights; however, where the NTS has

$$y = \sigma\left(b + \sum_i w_i x_i h s_i\right) \qquad \text{(2.15 revisited)}$$

UESMANN adds one to the modulator to make $h = 0$ the value at which the weights take their nominal values, and removes the sensitivity value s_i:

$$y = \sigma\left(b + (h+1)\sum_i w_i x_i\right). \qquad \text{(3.2 revisited)}$$

In both these equations, y is the node output, w_i and x_i are the ith weight and input, b is the bias, h is the modulator and σ is the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$. It therefore modulates all the weights uniformly: when $h = 0$ the weights are nominal, and when $h = 1$ the weights are doubled.

Intuitively, it might seem unlikely that such a simple system would be able to perform two qualitatively different functions: indeed, the initial design of the

system used by Sauzé and Neal [243] suggests that they believed that increasing the modulator would simply create more of the behaviour for which the network was trained. However, this is not the case: Sauzé and Neal instead found that their network behaved unpredictably when the modulator was used, and so switched to modulating the output layer only. We wished to study the possibility of controlling this behaviour, beginning with simple functions.

The simplest functions studied in neural networks are typically the binary boolean functions. Sec. 3.1 shows that a UESMANN network with two hidden nodes can perform all possible pairings of such functions in the same number of parameters required to perform a single function, and establishes the size of the volume occupied by each pairing in the parameter space.

In Chapter 4 a simple algorithm for training UESMANN networks through the back-propagation of errors is described, based on finding the gradients with respect to both the weights and doubled weights for each problem respectively, and alternating between each in the update steps. This algorithm successfully finds solutions for all binary boolean functions, having problems with some combinations (e.g. $x \wedge y \rightarrow \neg(x \vee y)$) but outperforming two rival techniques in others (e.g. $x \oplus y \rightarrow x \wedge y$).

The networks are analysed in some depth in Sec. 4.5 demonstrating that the modulation has the effect on a single node of raising the bias and narrowing the unsaturated part of the sigmoid activation function's output. A multi-layer network of such nodes either functions with all the nodes in saturation, in which case solutions are easy to find; or finds a careful balance between the weights which may be significantly harder to find. This difficulty is sometimes unpredictable: consider the pairing $x \wedge y \rightarrow \neg(x \vee y)$, which has by no means the smallest solution volume, but is hard to train. In this case, it is possible that the solution lies through a narrow "pass" in the parameter space which is difficult to find from the initial weight range used in the experiments — increasing the weight range increased the number of solutions found. Other problem domains are likely to have similar difficulties.

Part III tests UESMANN against two other modulatory techniques in two simple classification tasks: switching between horizontal and vertical line recognition and recognising handwritten digits. The two other techniques were *output blending*, which trains two networks separately for the $h = 0$ and $h = 1$ functions and uses h to interpolate linearly between the outputs; and *h-as-input*, which supplies h as an extra input to a single network.

UESMANN was able to find solutions to both problems which were comparable with the other methods, particularly in line recognition. Interestingly, while

the average performance of UESMANN was worse at line recognition, some individual networks outperformed all networks of both h-as-in and output blending (Fig. 6.10, p. 183). In recognising handwritten digits UESMANN was less able, but the performance was still comparable at higher node counts (Fig. 7.7, p. 209).

Finally, Part IV tests UESMANN against the two other methods in a control and regression problem, attempting to achieve both homeostasis and area coverage in a robot with a simple charging model by supervised learning of two behaviours and switching between them. UESMANN performed well here, and much better than h-as-input which failed to learn solutions which crossed the "reality gap" between simulation and reality. Output blending also performed well, but this may again have been due to the reality gap: the steering on the robot failed to turn when the velocity differential between the wheels was small, counteracting the wide transition which would normally cause this network type to perform poorly. Because these experiments took place in the real world with its concomitant complexities, each network's runs varied a great deal. This made analysis of the behaviours difficult, and exposed a number of problems in each network type which may have been due to problems with the training data.

Perhaps the most interesting aspect of UESMANN is the transition behaviour, when the modulator is between the extrema 0 and 1. The transition region, within which the function being performed is notably different from the extrema, is typically wide. The functions produced by the networks at these points are usually (but not always) some compromise between the extrema. The transitions regions were compared with both h-as-input and output blending: in h-as-input the behaviour was similar but with a narrower transition, while output blending produced a smooth linear interpolation of the outputs across the entire region. In classification problems this resulted in an almost zero-width transition due to thresholding, and a transition occupying the entire range of h in regression and control problems. In the robot control problem, UESMANN demonstrated a complex non-monotonic transition which created useful variation in the traversed paths.

The above is a brief summary of the findings of this thesis; we will now discuss some of the issues arising from the findings in more depth from the point of view of our initial research questions.

12.2 Simplicity

Our second question was:

> How simple is it possible to make such a system, so that we can make the fewest possible prior assumptions about it?

UESMANN is a simple modulatory system for an MLP: it multiplies all the weights by a global parameter, which is a single operation. That we multiply by $h+1$ instead of h is just convenience: the weights have their nominal values at 0 and 1 rather than at 1 and 2. Also, in a node defined by $y = \sigma\left(b + h\sum_i w_i x_i\right)$, setting $h = 0$ would always produce zero output.

It might be considered that an additive modulation is simpler yet, but networks of this type have not been tested. Rather than introducing a completely novel form of modulation, it was decided to build upon (or rather simplify) the existing NTS. Whether addition is a "simpler operation" than multiplication is an argument in number theory[1].

It can be argued that modulating both weights and biases, i.e.

$$y = \sigma\left((1+h)(b + \sum_i w_i x_i)\right) \qquad \text{(3.81 revisited)}$$

is simpler. Nodes of this type modulate by narrowing the unsaturated region of the activation function, which is certainly a simpler operation. This form of modulation was briefly tested in Sec. 3.2 (p. 82) and found to be fairly capable on the boolean functions, but unable to find solutions for all. It was therefore considered to be below the (somewhat arbitrary) threshold for "usefulness."

Another possible modulation scheme involves modulating the bias, rather than the weights:

$$y = \sigma\left((1+h)b + \sum_i w_i x_i\right). \qquad \text{(3.82 revisited)}$$

This modulation doubles the intercept on the node when $h = 1$. While this was also investigated in Sec. 3.2 (p. 82) and found to have a similar performance on the boolean functions as UESMANN, it had a slightly worse distribution of probabilities of solutions. However, it is still above our arbitrary threshold in that it can perform all boolean pairings, and perhaps should be investigated further. Whether it is

[1]Briefly, the real numbers form a field, and multiplication is a necessary operation on a field. Addition could exist without multiplication, but the reals would not then be a field, so on the reals at least, multiplication is as fundamental as addition.

"simpler" or not is a matter of opinion: it is still a single multiplication, but does not have the same effect on the node output as it leaves the width of the unsaturated region unmodified.

There are many possible enhancements to the basic back-propagation algorithm which could have been used: momentum, regularisation (including weight decay) and so on. Indeed, momentum is considered part of the core gradient descent algorithm, appearing in Rumelhart, Hinton and Williams [237]. These were not used: while straightforward to implement, the aim was to investigate the basic behaviour of the algorithm, keeping things as simple as possible. It is also possible that certain enhancements would also have required careful tuning to achieve a good performance. For example, momentum in particular might have made it more likely that narrow "passes" through the topography of the error surface would have been overshot. However, these should certainly be tested in the future, not least because they may provide insights into the nature of the learning process.

No batching was used in the UESMANN training algorithm; instead, the paired examples are presented first at $h = 0$, an update is made, then at $h = 1$ with another update being made. There are two points to be made here: first, and most obviously, that batching is another enhancement which was not implemented for the sake of simplicity. Secondly, while this is the simplest possible form of the algorithm from an implementation point of view, it is perhaps more mathematically complex than finding the mean gradient of the two results and applying it. The result would still be stochastic gradient descent given that we do not batch the means of the gradient pairs and apply them as a single update.

Thus we have analysed the behaviour of a simple system, but it could be argued that it is not the most simple. Perhaps additive or bias-only modulation would be simpler, or a combination of the two. Perhaps combining the two gradient vectors into one mean vector would be simpler. Nevertheless, UESMANN is a simple network which has the strength of being built upon an existing system (the Neal-Timmis AES).

12.3 Is it useful?

Our third question was:

> What engineering advantages might such a system have?

On simple classification problems UESMANN does not perform the end-point functions as accurately as the two other methods under test, although the performance

was still good. However, the complex behaviour of the transition region gave a wide gradual shift between one classification and another. This is potentially useful behaviour, although the use-case is somewhat esoteric.

In the robot problem, UESMANN performed surprisingly well, but much of the desired behaviour could have been achieved by other means. For example, the transition region could be controlled by using an output blending system with a sigmoid imposed on the modulator to precisely set the centre point and width. However, the "chaotic" (to use the term loosely, but see also Sec. 12.5.2.1) nature of the transition behaviour is harder to replicate with a traditional system. This appeared to provide the network with a source of variety which allowed it to cover more of the arena. It might be thought that a small amount of noise would produce the same behaviour; however, the real robot is in such a noisy environment (due to sensor and actuator irregularities) but does not show the same degree of dependence in h-as-input or output blending.

Thus the complex transition of UESMANN might be its most directly useful trait, providing a gradual shift in classification problems and a source of variety in control problems. The latter has applications in problems where the system might otherwise form undesirable loops, but also in human/machine interaction. In initial work on their AES, Neal and Timmis [209] found that people responded emotionally to the fairly simple behaviour of the system:

> Requests to "leave the poor thing alone" and other such comments are not uncommon. Indeed it is surprising how people are very willing to project emotions onto a small autonomous robot which exhibits even very rudimentary displays of "distress" and "fear."... Although the mechanism used is extremely simple, the behaviour generated is both functional and emotionally appealing. ([209], p. 8)

The network described in the cited paper used the modulator to increase the weights of the entire network, leading to unpredictable behaviour rather than "more of the same," as described above. UESMANN could provide such a system with an alternative behaviour into which the system could transition in a slightly unpredictable, naturalistic way, which might be of use in situations where it is desirable for humans to form relationships with robots. One possible example is support robots for the elderly or disabled [294]. This would be particularly useful in combination with a Neal/Timmis (or similar) release/decay hormone model, thus forming a full AES.

Most neuromodulatory AESs in the literature have either pre-designed neural networks, or use some form of evolutionary algorithm to determine the parameters. UESMANN uses stochastic gradient descent, and thus is a supervised learning

technique requiring a large number of examples of the "correct" behaviour at the modulation extrema. While this is not ideal, it is often easier to define the required behaviours by example rather than designing the network by hand.

One existing problem noted in the literature review (Sec. 2.5.6.5, p. 51) was the hexapod of Henley and Barnes [118]. This walking robot used an neuromodulator to vary the leg lift height, but suffered from undesirable translations under modulation. This was resolved by setting modulator sensitivities by hand, but it is possible that training for the modulator extrema using a UESMANN network might produce better results. Of course, it is also possible that in such problems output blending and h-as-input could provide useful (or better) solutions, and this thesis also demonstrates that these two simple neuromodulatory techniques have their place.

One possible advantage of UESMANN over output blending and h-as-input has not yet been discussed, and has only been tested in preliminary work: the possibility of extending the algorithm to train for h at multiple points, not just the extrema $h = 0$ and $h = 1$. Work on line recognition with a third function at $h = 0.5$ suggests that this is possible, particularly if the third function is in some sense "between" the functions at the extrema. For example, an intermediary function which detected any line (whether horizontal or vertical) worked, but an intermediary which detected blanks did not. Values outside the range $[0, 1]$ might also have show interesting behaviour.

Finally, UESMANN is limited by only having one modulator: the Neal/Timmis system can have any number of modulators. It is difficult to see how it could be extended to have this capability, although this possibility will be discussed below in Sec. 12.6.

Nevertheless, UESMANN is an interesting network. It can be trained using gradient descent (which makes it a possible candidate for deep learning), performs reasonably well at the extrema, provides a wide — but not too wide — transition region, and the transition has irregularities which may be useful in certain cases. It is also simple to understand and implement, and provides an "existence proof" that is is possible to construct an simple neuromodulatory model which can solve a large range of problems.

12.4 What can it tell us about biology?

Our final question was:

> If such a simple system can be built, can we learn from it anything about the nature and evolution of the biological systems which inspired it?

UESMANN is a biologically inspired system; it is not bio-mimetic. That is, it is not an attempt to directly model a biological process, but is instead an attempt to apply the general principles of operation of such a process. This means it is unlikely to provide many insights into the workings of biological systems. The biological neuron is far more complex than the MLP neuron in its response to neuromodulators. In biology, the intrinsic nature of a neuron can be changed in different ways by neuromodulators; for example, a "tonically firing" neuron can change into a "bursting" neuron – these are two fundamentally different types of activity [182]. The UESMANN MLP neuron has no variable intrinsic properties; instead, modulation is performed entirely at the synaptic level.

However, it does provide a potentially useful "existence proof" of a certain form of modulation. In UESMANN, a modulator acts in a uniformly excitatory manner on a group of units which have a particular learned behaviour. Rather than simply performing more of the same behaviour when the modulator is increased, it can instead generate an entirely different learned behaviour. This might come as a surprise to those without a deep familiarity with neuromodulation because naïve descriptions of neuromodulator action sometimes describe neuromodulators as increasing or decreasing *behaviours* of large groups of cells, rather than the synaptic connections between them.

Consider the behaviour of the first Neal/Timmis systems [209, 243]: both these systems up-modulated the entire network in an attempt to control the "amount" of behaviour elicited, and both performed in unexpected ways when the modulator was such that the nominal weights were not used. This was controlled in [243] by modulating only the output layer, which produced the desired behaviour.

UESMANN demonstrates that even in a system in which all synaptic connections are strengthened by a modulator in exactly the same way, the behaviour when the modulator is increased can be a qualitatively different behaviour which may also be a learned behaviour. Thus UESMANN shows how an extremely reductive model of excitatory neuromodulation can result in more complex behaviour than is often realised. While this may be obvious to those who have studied neuromodulation in some depth, systems like UESMANN may be useful for introducing such complexities to students.

With regard to the second part of the research question, it is unlikely UESMANN can provide information about the evolution of neuromodulatory systems. While it is possible that UESMANN itself can be trained by an evolutionary algorithm, the biological neuron is a different and far more complex system. Any data gathered

on the evolvability of UESMANN-style systems would not be relevant to biology, beyond the existence proof that such modulatory methods can be evolved.

12.5 Future work

There is a large amount of work left undone in the study of this network due to time and scope limitations. While many possible avenues for future study have been noted in the course of the thesis, it is useful to summarise them here and introduce a few more.

12.5.1 Classification and boolean functions

12.5.1.1 Is there a preference for $h = 0$?

We note in Sec. 4.5.7.3 (p. 153) that the boolean pairings $x \wedge y \rightarrow \neg(x \vee y)$ and $x \oplus y \rightarrow x \wedge y$ appear to have a "preference" for the $h = 0$ function; that is, the transition is skewed such that this function is performed for the larger part of the modulator range: see Fig. 4.32 (p. 139) and Fig. 4.43 (p. 153). This also appears to occur in the line detection transition (Fig. 6.18, p. 194). However, it is not apparent in the MNIST classification transitions of Fig. 7.13 (p. 222) and Tables 7.19 and 7.20 (p. 223). It is difficult to judge if it is present in the robot control solutions, but Fig. 9.28 (p. 271) would suggest not.

Therefore if there is such a preference, it is only seen to manifest in binary classification problems. Further investigation is required to see if this bias is real, and if so, how it is caused and how it can be either used or ameliorated.

12.5.1.2 How few hidden nodes for line classification?

Another open problem is how few hidden nodes are required for a good solution in line recognition. Solutions were obtained with as few as three nodes (Sec. 6.5.3, p. 187), but it was found that a two node solution can exist (since the three node solution has a redundant node). It is therefore likely that such a solution exists, but it may require a large number of random initial weight trials.

In MNIST handwriting recognition this is not an issue: the performance degrades as expected below 10 hidden nodes, although it is possible that good solutions exist with fewer.

12.5.1.3 Why are vertical lines harder to recognise?

In Sec. 6.3.2 (p. 174) it was noted that vertical lines appeared more difficult to train an output blending system to recognise than horizontal lines, which should not be the case. It is likely that this is due to a problem with the code generating the training data. This needs to be investigated.

12.5.1.4 Training more functions at more modulator values

As stated above, preliminary work suggests that it might be possible to train additional functions at other h values, such as $h = 0.5$, or even at values outside the range $[0, 1]$ such as $h = 2$. Algorithm 1 (p. 97) is easily modified to accept such values. We would predict that a third function at $h = 0.5$ can be trained for, provided it is close to one of the end-point functions or forms a "natural intermediate" between the two functions — if the outputs are binary, this would suggest a short Hamming distance between each end-point and the intermediate.

Training more functions could also work in h-as-input without requiring any modification. Output blending would require a more complex interpolation technique, perhaps using splines to interpolate between more than two networks.

12.5.1.5 Why does h-as-input outperform output blending sometimes?

We note from Fig. 7.7 (p. 209) that at high node counts h-as-input consistently outperforms output blending. While unrelated to UESMANN, this is strange considering that output blending uses two separate networks trained for each function, while h-as-input only uses one (with an extra input for the modulator). It would be interesting to investigate this further.

12.5.2 Control

There are a few anomalies in the robot control results which require further investigation, notably the "anomalous stops" (Sec. 10.2.2.3, p. 286), the fairly common collisions and the "serendipitous stop" of Fig. 10.31 (p. 303). It may be that these are due to problems in the training data, which was generated from the simulated arena which was square with a linear emitter in a corner and some internal walls, all at 90° degrees. This needs further investigation, which may only be possibly by retraining the networks in a more irregular arena with a central emitter to generate more representative data, and re-running the experiments.

We also noted that much of the unusual behaviour of h-as-input and UESMANN appeared to be due to a notional "crosstalk" between the two behaviours. This

manifested as the expected behaviour with an element of the other behaviour, such as *exploration* with no obstacle avoidance. This may have been an illusion of the kind often seen in naïve analysis of robot behaviour and needs verification.

12.5.2.1 Is UESMANN on the edge of chaos?

It was found that the paths of the robot had more variation within the runs of each UESMANN network than within the networks of other types. As discussed in Sec. 11.2 (p. 308), this suggests that UESMANN may be sensitive to small differences in the input and/or modulator values, perhaps to the extent of exhibiting the "sensitive dependence on initial conditions" characteristic of chaotic systems. Thus UESMANN networks may operate on the "edge of chaos," (i.e. near the critical line which separates ordered and chaotic behaviour) than the other modulatory methods examined.

Systems which operate in this region have been found to be capable of complex behaviour (including computation) [164, 215]. The *criticality hypothesis* states that "systems which are in a dynamical regime between order and disorder have the highest level of computational abilities and achieve an optimal trade-off between robustness and flexibility" [235]. Indeed, theorists such as Kauffman [146] believe that living systems evolve towards the edge of chaos.

Whether the dynamical system formed by a UESMANN network together with its environment operates within this critical region can be experimentally tested by recording a large number of time series from different experiments and measuring the Lyapunov exponents and exponent spectra [304]. Naturally, these should also be examined for the other two modulation methods. A system which is on the "edge of chaos" should show Lyapunov exponents which are positive, but close to zero [144].

This should also be tested in other problems: it may be that UESMANN is operating near the critical line in this particular problem, but this does not necessarily imply that it will do so in others. Should it appear that UESMANN does operates on the edge of chaos in multiple scenarios, further analysis should be done to investigate why this is so.

12.5.3 Enhancements

As discussed above and in Sec. 4.1 (p. 88), no modifications to stochastic gradient descent (momentum, regularisation, etc.) were used. These should be tested because they may either improve or detract from the performance, as described in Sec. 12.2 above. Batching should also be tested because it may be conceptually "simpler" than stochastic gradient descent, and also because it may improve results – although the

loss of the "jitter" provided by alternating between the two function gradients may also be detrimental. These methods would also improve the results from output blending and h-as-input, perhaps more than they would improve UESMANN.

12.5.4 Alternative modulation schemes

As discussed above and in Sec. 3.2 (p. 82), two alternative modulation schemes (weight and bias, and bias-only) performed slightly worse than UESMANN: bias-only in particular was rejected on the basis of a slightly worse probability distribution in the boolean pairing solution counts. These should be tested more thoroughly because they may perform better on more complex tasks, and because it could be argued (as stated above) that they are simpler than UESMANN. Therefore the line detection and handwriting recognition experiments should be repeated with these schemes, and the better of the two carried over into more robot control experiments.

12.6 More modulators?

One of the main drawbacks of UESMANN as a modulatory system is that it only works with a single modulator. This is in contrast to the original Neal/Timmis system, which incorporates multiple "hormones" with a sensitivity for each applied to each weight in the network.

It may be possible to extend UESMANN to multiple modulators by assigning neurons to be modulated by different modulators, changing the core equations (Eqs. 4.13 to 4.16, p. 96) accordingly. However, training such a network would be difficult because examples must be provided for every possible high/low combination of modulators. Nevertheless, the attempt should be made.

Alternatively, a system could be composed of multiple UESMANN networks, each separately trained to respond to a different modulator. Again, the difficulty is the training data: the entire system can no longer be trained as a "black box": each sub-network would require its own set of examples.

12.6.1 Reinforcement learning

UESMANN is currently problematic from the point of view of learning adaptivity and homeostasis because it is trained using a supervised learning method. There are several possible avenues for reinforcement learning, of which the most obvious is some form of evolutionary algorithm, perhaps based on the NEAT algorithm of Stanley and Miikkulainen [264]. Other techniques are available, for example

Riedmiller [231] adapts Q-learning for neural networks, while the more recent deep Q-network of Mnih et al. [198] extends this idea to deep convolutional networks.

12.6.2 Recurrent networks

It may be possible to use UESMANN to modulate parts of a recurrent network, such as the "readout" layer of a reservoir system (see Sec. 2.3.7.6, p. 33), for example, replacing the learned weights with a UESMANN readout layer. This would allow the network to be trained to recognise different temporal patterns at different modulator levels.

12.6.3 Alternative activation functions and deep learning

There is no inherent reason why UESMANN layers cannot appear in deep networks, provided a suitable activation function can be found for gradient descent (or for the Q-learning system described in the paragraph above). It would be straightforward to rewrite the core UESMANN equations for a rectified linear unit (ReLU), for example. Simple networks of this type should be tested, and if successful, incorporated into a test deep learning problem. It would be interesting to construct a deep convolutional binary image classifier with a UESMANN layer (or layers) which could transition to another classification, and it may provide useful functionality as well as being the first example of a deep AES.

Additionally, if it were possible to modify a deep reinforcement learning system such as the deep Q-network (described above) to contain UESMANN layers, a truly adaptive, deep artificial neuroendocrine system would finally be possible.

Appendix A

The robot system architecture

The architecture was designed so that the same software runs the neural network code whether the experiment is a Gazebo simulation or on the real robot. Sensor emulations were designed where necessary, and a visual tracking system was built to give positional data during the robot runs. To this end, the neural network software was written in the Robot Operating System (ROS) [225], which interoperates well with Gazebo and has a mature and straightforward communications architecture.

ROS executables are known as "nodes", which are able to publish and subscribe to named "topics." These hold structured messages of arbitrary complexity. A ROS application is made up of several nodes, each of which can be running on different machines (although in our application the nodes all run on the "host" machine, a laptop). The neural network software acts as such a node, subscribing to sensor topics and outputting motor control topics. In a Gazebo experiment, Gazebo also acts as such a node. Because the robot does not natively run ROS, robot experiments instead run a "bridge node" which communicates over TCP/IP with a server on the robot, which both controls the robot via its native ARIA interface and communicates sonar and light sensor readings back to the host. This occurs at a rate of 10Hz, although the light sensor data is updated at ~ 3.3Hz (see Sec. 10.1.2.1, p. 275).

Additionally, a localisation topic is subscribed to by the network node to allow position to be included in the experimental logs. The data for this either comes from the localisation system described in Appendix B or from Gazebo. In both cases, the data is sent via the "Diamond Apparatus" simple publish/subscribe system[1]. An overview of the architecture is given in Fig. A.1.

Sensor data is sent from the robot and motor data received in return, at a frequency of 10Hz.

[1] `https://github.com/jimfinnis/DiamondApparatus`

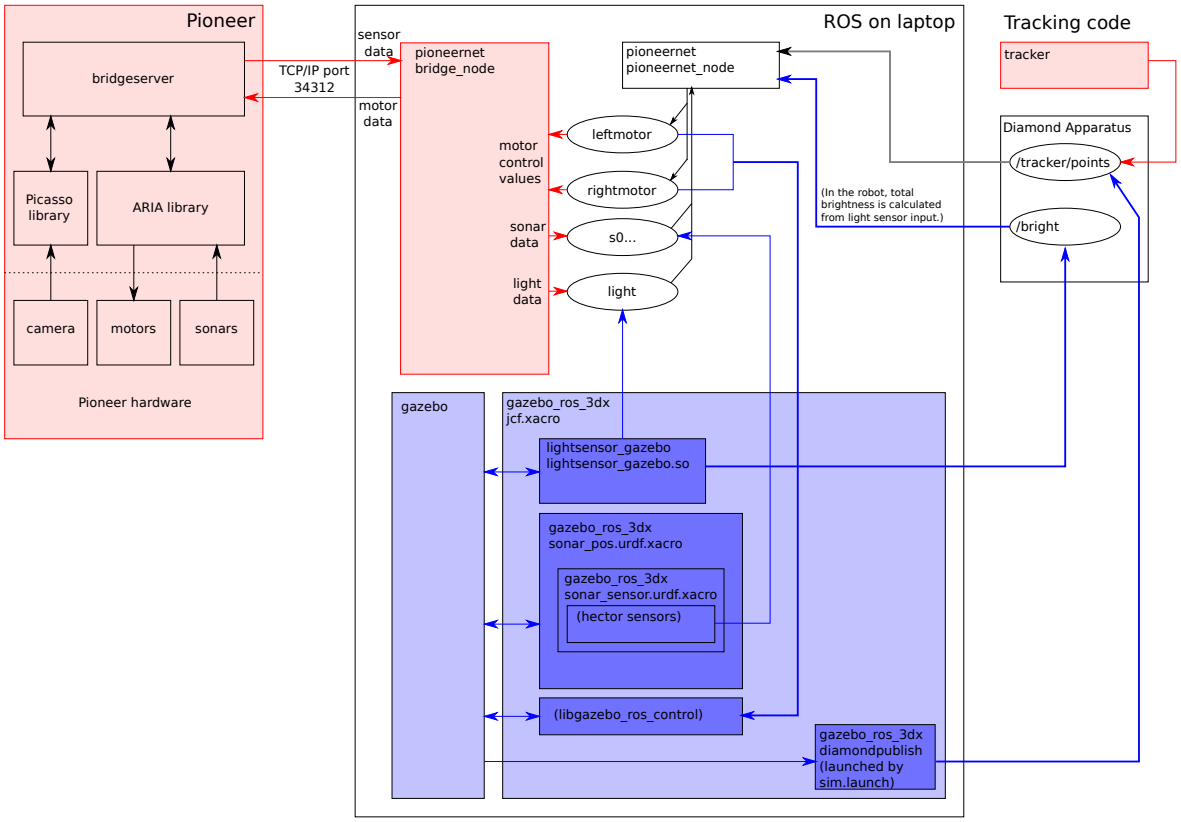


Figure A.1: Architecture of Pioneer/Gazebo controlled by ROS. Components used only in the robot are in red, those used only in the simulation are in blue. "Diamond Apparatus" is a simple publish/subscribe communications system which runs outside ROS.

In the Gazebo simulation the sonar sensors are simulated using the Team Hector sensor plugins [152]. These permit a degree of Gaussian noise to be added to the sonar distance reading, and a value of $\sigma = 0.005$ was selected. This small value was chosen because the robot sonar noise was as yet uncharacterised; it only has a small effect on the repeatability of the paths.

A.1 Safety on the robot

Once the first message has been received, the robot will measure the interval since the previous message. If no message has been received for 0.5s, the bridge program will stop the motors and quit. Additionally, the low level ARIA system will stop the motors if the battery is low or the bumpers encounter a persistent obstruction.

Appendix B

The robot tracking system

This appendix describes the localisation system used in the robot experiments, described briefly in Sec. 10.1.5. As stated in that section, a commodity webcam with a view of the entire arena is used to track the robot by locating a red LED in the image and performing an inverse perspective transform. The steps are:

- Take image.
- Apply a Gaussian blur to make the diffused red LED clearer and reduce noise.
- Filter out a given range of hue, separation and value, turning all other pixels black.
- Use OpenCV's `simpleBlobDetector` to detect blobs, returning a list of key points (centroids of candidates).
- Remove blobs whose centroids are dark (OpenCV has a bug whereby the "colour filtering" in the blob detector does not work correctly[1]).
- Remove blobs which are in areas masked by hand to remove specular highlights from the floor and spurious blobs produce by the lamp hood holes.
- Transform blobs to world space, where world space is a 2D plane parallel to the floor, within which the LED moves.
- If we are operating in "locked" mode, remove blobs which are not within 0.2m of the last robot position found. This is a "strong" frame coherence assumption that the robot will not move more than 0.2m in a frame. The user engages locked mode once the tracker has successfully found the stationary robot.

[1] `https://www.learnopencv.com/blob-detection-using-opencv-python-c/`

- Sort the list by increasing distance from the previous 10 top blobs (if any).
- Find the 10 top blobs by removing all but the first 10 from this list. This is a "weak" frame coherence assumption that the correct blob is probably close to a blob found in the previous frame.
- The next point to be published – the position of the robot – is the head of this list.

This is given a little more formally in Algorithm 14. Fig. B.1 shows the image from the camera before and after manual exposure adjustment, showing the noise present when auto adjustment is used (setting the exposure very high) along with the blown-out pixels of the robot's LED. In the manual exposure image the LED is clearly a trackable red blob. The need for masking is clear, however: the lamp and its specular highlight are still too easily confused with the LED. It should also be noted that there is a fairly large area in which the LED is obscured by the lamp hood: the robot cannot be tracked here.

(a) Auto exposure **(b)** Manual exposure

Figure B.1: Images from the tracking camera, with auto and manual exposure settings

Algorithm 14 Tracking algorithm.

$M \leftarrow$ image to world transformation matrix
$prevKPDists \leftarrow$ empty list
$lastPoint \leftarrow (0, 0)$
repeat
  $I \leftarrow$ image from sensor (RGB)
  $I \leftarrow gaussianBlur(I)$ {Gaussian blur with kernel size 11, $\sigma = 1$}
  $I \leftarrow HSV(I)$ {Transform image into HSV space}
  for all $p \in I$ **do**
    if p_H outside range **or** p_S outside range **or** p_V outside range **then**
      $p \leftarrow (0, 0, 0)$
    end if
    $L \leftarrow blobDetect(I)$ {use OpenCV's `simpleBlobDetector` to get a list of keypoints}
  end for
  clear list $KPDists$
  for all $p \in L$ **do**
    if $p_r + p_g + p_b > 3 \times minBright$ **then** {point is bright at the centre (avoids dark blob detection)}
      if p **not in** a mask **then**
        $\vec{t} \leftarrow M \times (x_I, y_I, 1)^T$ {perform transform to world}
        $\vec{w} \leftarrow (t_x/t_z, t_y/t_z)$ {perform perspective division}
        if not locked **or** w within 0.2m of $lastPoint$ **or** $lastPoint_x < -99$ **then**
          $d \leftarrow$ minimum distance from points in $prevKPDists$
          append $(\vec{w}, d)$ to list $KPDists$
        end if
      end if
    end if
  end for
  sort $KPDists$ by d
  truncate $KPDists$ to first 10 points
  $prevKPDists \leftarrow KPDists$
  if $length(KPDists) > 0$ **then**
    $lastPoint \leftarrow head(KPDists)$
    publish $lastPoint$ to `/tracker/points` in Diamond Apparatus
  end if
until done

B.1 Calibration

Calibration consists of several phases:

- find the transform to convert image coordinates into world coordinates, done by marking known world positions in the image to give corresponding 2D world plane coordinates — this was done using a pole with a marker the same height as the LED to compensate for the height difference;
- calibrate the camera exposure so the LED is always detectable;
- find the optimal values for the HSV filtering;
- find the optimal values for the blob detection;
- establish which areas in the image need to be masked to avoid false positives – necessary because of specular highlights from the lamp on the floor, and holes in the lamp's hood emitting large spots of light.

Bibliography

[1] D. H. Ackley, G. E. Hinton and T. J. Sejnowski. "A learning algorithm for Boltzmann machines". In: *Cognitive Science* 9.1 (1985), pp. 147–169.

[2] ActivMedia Robotics. *Pioneer 2/PeopleBot Operations Manual*. 2001.

[3] U. Aickelin and S. Cayzer. "The Danger Theory and its Application to Artificial Immune Systems". In: *Proceedings of the 1st International Conference on Artificial Immune Systems*. Canterbury, UK, 2002, pp. 141–148.

[4] U. Aickelin, D. Dasgupta and F. Gu. "Artificial immune systems". In: *Search Methodologies*. Springer, 2014, pp. 187–211.

[5] L. Altenberg. "The Schema Theorem and Price's Theorem". In: *9th International Workshop, Foundations of Genetic Algorithms 2007*. Mexico City: Springer, 2007, pp. 23–49.

[6] J. R. Anderson et al. "An integrated theory of the mind". In: *Psychological Review* 111.4 (2004), pp. 1036–1060.

[7] M. Anthony. *Boolean functions and artificial neural networks*. Tech. rep. LSE-CDAM-2003-1. London School of Economics and Political Science, 2003.

[8] M. A. Arbib and J.-M. Fellous. "Emotions: from brain to robot". In: *Trends in Cognitive Sciences* 8.12 (2004), pp. 554–561.

[9] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[10] R. C. Arkin. "Homeostatic Control for a Mobile Robot - Dynamic Replanning in Hazardous Environments". In: *Journal of Robotic Systems* 9.2 (1992), pp. 197–214.

[11] R. C. Arkin and T. Balch. "AuRA: Principles and practice in review". In: *Journal of Experimental and Theoretical Artificial Intelligence* 9.2-3 (1997), pp. 175–189.

[12] R. C. Arkin and D. C. Mackenzie. "Planning to Behave: A Hybrid Deliberative/Reactive Robot Control Architecture for Mobile Manipulation". In: *International Symposium on Robotics and Manufacturing*. Maui, HI, USA, 1994.

[13] W. R. Ashby. *An Introduction to Cybernetics*. London: Chapman & Hall, 1957.

[14] W. R. Ashby. *Design for a Brain*. New York: Wiley, 1952.

[15] R. D. Beer. "A dynamical systems perspective on agent-environment interaction". In: *Artificial Intelligence* 72.1 (1995), pp. 173–215.

[16] R. D. Beer. "Dynamical approaches to cognitive science". In: *Trends in Cognitive Sciences* 4.3 (2000), pp. 91–99.

[17] R. D. Beer. "On the dynamics of small continuous-time recurrent neural networks". In: *Adaptive Behavior* 3.4 (1995), pp. 469–509.

[18] R. D. Beer and J. C. Gallagher. "Evolving dynamical neural networks for adaptive behavior". In: *Adaptive Behavior* 1.1 (1992), pp. 91–122.

[19] U. Behn. "Idiotypic networks: toward a renaissance?" In: *Immunological Reviews* 216.1 (2007), pp. 142–152.

[20] M. Ben-Ari and F. Mondada. *Elements of Robotics*. Springer International Publishing, 2018.

[21] Y. Bengio, P. Frasconi and P. Simard. "Learning long term dependencies in recurrent networks". In: *IEEE International Conference on Neural Networks*. 1993, pp. 1183–1188.

[22] Y. Bengio, P. Simard and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166.

[23] Y. Bengio et al. "Greedy layer-wise training of deep networks". In: *Advances in Neural Information Processing Systems*. 2007, pp. 153–160.

[24] I. S. N. Berkeley. *A revisionist history of connectionism*. 1997. URL: `http://www.universelle-automation.de/1969_Boston.pdf` (visited on 16/08/2019).

[25] H.-G. Beyer and H.-P. Schwefel. "Evolution strategies – A comprehensive introduction". In: *Natural Computing* 1.1 (2002), pp. 3–52.

[26] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, 1995.

[27] C. M. Bishop. *Pattern Recognition and Machine Learning*. New York, NY, USA: Springer-Verlag, 2006.

[28] C. Blum and X. Li. "Swarm intelligence in optimization". In: *Swarm Intelligence*. Springer, 2008, pp. 43–85.

[29] E. K. Blum. "Approximation of Boolean Functions by Sigmoidal Networks: Part I: XOR and Other Two-Variable Functions". In: *Neural Computation* 1.4 (1989), pp. 532–540.

[30] V. Braitenberg. *Vehicles*. MIT Press, 1986.

[31] C. Brom and J. Bryson. *Action selection for intelligent systems*. Tech. rep. 044-1. European Network for the Advancement of Artificial Cognitive Systems, 2006.

[32] R. A. Brooks. "A robot that walks; emergent behaviors from a carefully evolved network". In: *Neural Computation* 1.2 (1989), pp. 253–262.

[33] R. A. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal of Robotics and Automation* 2.1 (1986), pp. 14–23.

[34] R. A. Brooks. "Artificial life and real robots". In: *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. Paris, France: MIT Press, 1992.

[35] R. A. Brooks. "Elephants don't play chess". In: *Robotics and Autonomous Systems* 6.1-2 (1990), pp. 3–15.

[36] R. A. Brooks and P. A. Viola. "Network based autonomous robot motor control". In: *Advanced Neural Computers*. Ed. by R. Eckmiller. Elsevier Science Publishers, 1990, pp. 341–348.

[37] R. A. Brooks. "Challenges for complete creature architectures". In: *First International Conference on Simulation of Adaptive Behavior*. 1991, pp. 434–443.

[38] R. A. Brooks. "Intelligence without representation". In: *Artificial Intelligence* 47.1 (1991), pp. 139–159.

[39] R. G. Brown. “Smoothing”. In: *Forecasting and Prediction of Discrete Time Series*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1963, p. 238.

[40] J. Bryson. “Cross-paradigm analysis of autonomous agent architecture”. In: *Journal of Experimental and Theoretical Artificial Intelligence* 12.2 (2000), pp. 165–189.

[41] J. J. Bryson. “Action selection and individuation in agent based modelling”. In: *Proceedings of Agent 2003*. 2003, pp. 317–330.

[42] D. Bucher and E. Marder. “SnapShot: neuromodulation”. In: *Cell* 155.2 (2013), p. 482.

[43] D. V. Buonomano and W. Maass. “State-dependent computations: spatiotemporal processing in cortical networks”. In: *Nature Reviews Neuroscience* 10.2 (2009), p. 113.

[44] F. M. Burnet. “A modification of Jerne’s theory of antibody production using the concept of clonal selection”. In: *CA: A Cancer Journal for Clinicians* 26.2 (1976), pp. 119–121.

[45] E. Cambria and B. White. “Jumping NLP curves: A review of natural language processing research”. In: *IEEE Computational Intelligence Magazine* 9.2 (2014), pp. 48–57.

[46] D. Cañamero. “A hormonal model of emotions for behavior control”. In: *Fourth European Conference on Artificial Life, ECAL ’97*. Brighton, UK, 1997.

[47] G. A. Carpenter and S. Grossberg. “ART 3: Hierarchical search using chemical transmitters in self-organizing pattern recognition architectures”. In: *Neural Networks* 3.2 (1990), pp. 129–152.

[48] G. A. Carpenter and S. Grossberg. “The ART of adaptive pattern recognition by a self-organizing neural network”. In: *Computer* 21.3 (1988), pp. 77–88.

[49] G. A. Carpenter, S. Grossberg and J. H. Reynolds. “ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network”. In: *Neural Networks* 4.5 (1991), pp. 565–588.

[50] G. Chevalier and J. M. Deniau. “Disinhibition as a basic process in the expression of striatal functions”. In: *Trends in Neurosciences* 13.7 (1990), pp. 277–280.

[51] K. Cho et al. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1724–1734.

[52] A. Clark and R. Grush. “Towards a cognitive robotics”. In: *Adaptive Behavior* 7.1 (1999), pp. 5–16.

[53] D. Cliff. “Computational Neuroethology: A Provisional Manifesto”. In: *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. Cambridge, MA, USA: MIT Press, 1990, pp. 29–39.

[54] D. Cliff. “Neuroethology, computational”. In: *The Handbook of Brain Theory and Neural Networks*. Ed. by M. Arbib. 2nd ed. MIT Press, 2003, pp. 737–741.

[55] M. Cohn. “The real ‘danger’ lies in the failure to confront fundamentals”. In: *Scandinavian Journal of Immunology* 88.6 (2018), e12726.

[56] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314.

[57] M. F. Dacrema, P. Cremonesi and D. Jannach. “Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches”. In: *Proceedings of the 13th ACM Conference on Recommender Systems (RecSys 2019)*. Copenhagen, 2019.

[58] Y. Dan and M. M. Poo. “Hebbian depression of isolated neuromuscular synapses in vitro”. In: *Science* 256.5063 (1992), pp. 1570–1573.

[59] L. Danziger and G. L. Elmergreen. “Mathematical models of endocrine systems”. In: *The Bulletin of Mathematical Biophysics* 19.1 (1957), pp. 9–18.

[60] F. Dario et al. “Evolution of spiking neural circuits in autonomous mobile robots”. In: *International Journal of Intelligent Systems* 21.9 (2006), pp. 1005–1024.

[61] L. N. De Castro and J. Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer Science & Business Media, 2002.

[62] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197.

[63] R. Dechter. “Learning while searching in constraint-satisfaction problems”. In: *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*. Philadelphia, PA, USA, 1986.

[64] M. P. Deisenroth. “A Survey on Policy Search for Robotics”. In: *Foundations and Trends in Robotics* 2.1-2 (2013), pp. 1–142.

[65] E. A. Di Paolo. “Organismically-inspired robotics: homeostatic adaptation and teleology beyond the closed sensorimotor loop”. In: *Dynamical Systems Approach to Embodiment and Sociality* (2003), pp. 19–42.

[66] S. Doncieux et al. “Evolutionary robotics: what, why, and where to”. In: *Frontiers in Robotics and AI* 2 (2015), p. 4.

[67] M. Dorigo and M. Birattari. “Ant colony optimization”. In: *Encyclopedia of Machine Learning*. Springer, 2011, pp. 36–39.

[68] M. Dorigo, E. Bonabeau and G. Theraulaz. “Ant algorithms and stigmergy”. In: *Future Generation Computer Systems* 16.8 (2000), pp. 851–871.

[69] K. Doya. “Metalearning and neuromodulation”. In: *Neural Networks* 15.4-6 (2002), pp. 495–506.

[70] H. L. Dreyfus. *Alchemy and artificial intelligence*. Tech. rep. P-3244. Santa Monica, CA, USA: Rand Corporation, 1965.

[71] H. L. Dreyfus. “Why computers must have bodies in order to be intelligent”. In: *The Review of Metaphysics* (1967), pp. 13–32.

[72] R. O. Duda, P. E. Hart and D. G. Stork. *Pattern Classification*. John Wiley & Sons, 2012.

[73] P. Dürr, C. Mattiussi and D. Floreano. “Neuroevolution with analog genetic encoding”. In: *Parallel Problem Solving From Nature - PPSN IX*. Reykjavik, Iceland, 2006, pp. 671–680.

[74] R. C. Eberhart and J. Kennedy. “A new optimizer using particle swarm theory”. In: *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*. Vol. 1. New York, NY, USA, 1995, pp. 39–43.

[75] J. L. Elman. "Finding structure in time". In: *Cognitive Science* 14.2 (1990), pp. 179–211.

[76] J. L. Elman. "Distributed representations, simple recurrent networks, and grammatical structure". In: *Machine Learning* 7.2-3 (1991), pp. 195–225.

[77] D. Erhan et al. "Why Does Unsupervised Pre-training Help Deep Learning?" In: *Journal of Machine Learning Research* 11.Feb (2010), pp. 625–660.

[78] B. S. Everitt, S. Landau and M. Leese. *Cluster Analysis*. John Wiley & Sons, Inc., 1993.

[79] S. E. Fahlman, G. E. Hinton and T. J. Sejnowski. "Massively parallel architectures for Al: NETL, Thistle, and Boltzmann machines". In: *Proceedings of AAAI-83*. Vol. 113. Washington, DC, USA, 1983.

[80] L. S. Farhy. "Modeling of Oscillations in Endocrine Networks with Feedback". In: *Numerical Computer Methods, Part E*. Vol. 384. Methods in Enzymology. Academic Press, 2004, pp. 54–81.

[81] L. S. Farhy et al. "A construct of interactive feedback control of the GH axis in the male". In: *American Journal of Physiology-Regulatory, Integrative and Comparative Physiology* 281.1 (2001), R38–R51.

[82] B. G. Farley and W. Clark. "Simulation of self-organizing systems by digital computer". In: *Information Theory, Transactions of the IRE Professional Group On* 4.4 (1954), pp. 76–84.

[83] J. D. Farmer, N. H. Packard and A. S. Perelson. "The immune system, adaptation, and machine learning". In: *Physica D: Nonlinear Phenomena* 22.1-3 (1986), pp. 187–204.

[84] J.-M. Fellous. "From human emotions to robot emotions". In: *Architectures for Modeling Emotion: Cross-Disciplinary Foundations, American Association for Artificial Intelligence* (2004), pp. 39–46.

[85] J. C. Finnis. "Homeostatic Robot Control Using Simple Neuromodulatory Techniques". In: *Proceedings of TAROS (Towards Autonomous Robotic Systems)*. Guildford, UK, 2017, pp. 325–339.

[86] J. C. Finnis. *Rover walking: a neuroendocrine controller for switching between rolling and walking locomotion*. Aberystwyth University. 2013. URL: `http://users.aber.ac.uk/jcf1/finalYearReport.pdf` (visited on 15/08/2019).

[87] J. C. Finnis and M. Neal. "A simple drive load-balancing technique for multi-wheeled planetary rovers". In: *Proceedings of TAROS (Towards Autonomous Robotic Systems)*. Oxford, UK, 2013.

[88] J. C. Finnis and M. Neal. "UESMANN: A feed-forward network capable of learning multiple functions". In: *International Conference on Simulation of Adaptive Behavior*. Aberystwyth, UK, 2016, pp. 101–112.

[89] D. Floreano and C. Mattiussi. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. MIT press, 2008.

[90] D. Floreano and C. Mattiussi. "Evolution of spiking neural controllers for autonomous vision-based robots". In: *International Symposium on Evolutionary Robotics*. 2001, pp. 38–61.

[91] D. Floreano and F. Mondada. "Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot". In: *From Animals to Animats 3: Proceedings of the Third*

International Conference on Simulation of Adaptive Behavior. 1994, pp. 421–430.

[92] K. Friston. "The free-energy principle: a unified brain theory?" In: *Nature Reviews Neuroscience* 11.2 (2010), pp. 127–138.

[93] K. Fukushima and S. Miyake. "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition". In: *Competition and Cooperation in Neural Nets*. Springer, 1982, pp. 267–285.

[94] K. Funahashi and Y. Nakamura. "Approximation of dynamical systems by continuous time recurrent neural networks". In: *Neural Networks* 6.6 (1993), pp. 801–806.

[95] J. Garson. "Connectionism". In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta. Winter 201. Metaphysics Research Lab, Stanford University, 2016.

[96] GCTronic. *Extensions - Omnidirectional Vision Turret*. URL: `http://www.e-puck.org/index.php?option=com_content&view=article&id=26&Itemid=21` (visited on 21/01/2020).

[97] R. Ge et al. "Escaping from saddle points—online stochastic gradient for tensor decomposition". In: *Proceedings of the 28th Conference on Learning Theory*. Paris, France, 2015, pp. 797–842.

[98] I. Gerostathopoulos et al. "Architectural homeostasis in self-adaptive software-intensive cyber-physical systems". In: *European Conference on Software Architecture*. 2016, pp. 113–128.

[99] R. Gesztelyi et al. "The Hill equation and the origin of quantitative pharmacology". In: *Archive for History of Exact Sciences* 66.4 (2012), pp. 427–438.

[100] B. Girard et al. "Where neuroscience and dynamic system theory meet autonomous robotics: A contracting basal ganglia model for action selection". In: *Neural Networks* 21.4 (2008), pp. 628–641.

[101] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 2010, pp. 249–256.

[102] I. Goodfellow et al. "Generative adversarial nets". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680.

[103] J. Greensmith, A. Whitbrook and U. Aickelin. "Artificial immune systems". In: *Handbook of Metaheuristics*. Springer, 2010, pp. 421–448.

[104] D. Greer, P. McKerrow and J. Abrantes. "Robots in urban search and rescue operations". In: *Australasian Conference on Robotics and Automation, Auckland*. 2002, pp. 27–29.

[105] W. Grey Walter. *The Living Brain*. Gerald Duckworth & Co., 1953.

[106] S. Grossberg. "Adaptive Resonance Theory: How a brain learns to consciously attend, learn, and recognize a changing world". In: *Neural Networks* 37 (2013), pp. 1–47.

[107] K. Gurney, T. J. Prescott and P. Redgrave. "A computational model of action selection in the basal ganglia. I. A new functional anatomy". In: *Biological Cybernetics* 84.6 (2001), pp. 401–410.

[108] M. T. Hagan and M. B. Menhaj. "Training feedforward networks with the Marquardt algorithm". In: *IEEE Transactions on Neural Networks* 5.6 (1994), pp. 989–993.

[109] H. Hamann et al. "A hormone-based controller for evolutionary multi-modular robotics: From single modules to gait learning". In: *IEEE Congress on Evolutionary Computation*. 2010, pp. 1–8.

[110] H. Hamann et al. "Artificial hormone reaction networks: Towards higher evolvability in evolutionary multi-modular robotics". In: *Artificial Life XII: Proceedings of the 12th International Conference on the Synthesis and Simulation of Living Systems, ALIFE 2010*. 2010, pp. 773–780.

[111] L. G. C. Hamey. "XOR has no local minima: A case study in neural network error surface analysis". In: *Neural Networks* 11.4 (1998), pp. 669–681.

[112] S. Harnad. "Minds , Machines and Searle". In: *Journal of Experimental and Theoretical Artificial Intelligence* 1.August (1989), pp. 5–25.

[113] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[114] I. Harvey et al. "Evolutionary robotics: A new scientific tool for studying cognition". In: *Artificial Life* 11.1-2 (2005), pp. 79–98.

[115] D. Hassabis. "Artificial Intelligence: Chess match of the century". In: *Nature* 544.7651 (2017), p. 413.

[116] K. He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[117] D. O. Hebb. *The Organization of Behavior; A Neuropsychological Theory*. Psychology Press, 1949.

[118] J. J. Henley and D. P. Barnes. "An artificial neuro-endocrine kinematics model for legged robot obstacle negotiation". In: *8th ESA Workshop on Advanced Space Technologies for Robotics and Automation*. 2004, p. 22.

[119] J. Hinson et al. *The Endocrine System : Basic Science and Clinical Conditions*. 2nd ed. Churchill Livingstone, 2010, p. 185.

[120] G. E. Hinton. "Learning distributed representations of concepts". In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Vol. 1. Amherst, MA, USA, 1986, p. 12.

[121] G. E. Hinton. "To recognize shapes, first learn to generate images". In: *Progress in Brain Research* 165 (2007), pp. 535–547.

[122] G. E. Hinton and T. Shallice. "Lesioning an attractor network: investigations of acquired dyslexia". In: *Psychological Review* 98.1 (1991), pp. 74–95.

[123] S. Hochreiter and J. Schmidhuber. "Long short-term memory." In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[124] A. L. Hodgkin and A. F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of Physiology* 117.4 (1952), pp. 500–544.

[125] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, 1975.

[126] J. J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the National Academy of Sciences of the United States of America* 79.8 (1982), pp. 2554–2558.

[127] D. H. Hubel and T. N. Wiesel. "Receptive fields of single neurones in the cat's striate cortex". In: *The Journal of Physiology* 148.3 (1959), pp. 574–591.

[128] R. Humza et al. "Towards energy homeostasis in an autonomous self-reconfigurable modular robotic organism". In: *Proceedings of Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. Athens, Greece, 2009, pp. 21–26.

[129] P. Husbands and I. Harvey. "Evolution Versus Design: Controlling Autonomous Robots". In: *Proceedings of the Third Annual Conference of AI, Simulation, and Planning in High Autonomy Systems 'Integrating Perception, Planning and Action'.* Los Alamitos, CA, USA: IEEE Computer Society, 1992, pp. 139–146.

[130] P. Husbands. "Evolving robot behaviours with diffusing gas networks". In: *Evolutionary Robotics: Proceedings of the European Workshop on Evolutionary Robotics*. Paris, France, 1998, pp. 71–86.

[131] P. Husbands et al. "Brains, Gases and Robots". In: *ICANN'98:Proceedings of the 8th International Conference on Artificial Neural Networks*. Skövde, Sweden, 1998, pp. 51–63.

[132] P. Husbands et al. "Volume signalling in real and robot nervous systems". In: *Theory in Biosciences* 120.3-4 (2001), pp. 253–269.

[133] M. Hutson. "Has artificial intelligence become alchemy?" In: *Science* 360.6388 (2018), p. 478.

[134] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning*. Lille, 2015, pp. 448–456.

[135] W. Isaacson. *The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution*. Simon & Schuster, Limited, 2015.

[136] H. Jaeger. *The "echo state" approach to analysing and training recurrent neural networks*. Tech. rep. 148. Bonn, Germany: GMD-Forschungszentrum Informationstechnik, 2001.

[137] H. Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach*. Tech. rep. 159. Bonn, Germany: GMD-Forschungszentrum Informationstechnik, 2002.

[138] N. Jakobi. "Evolutionary robotics and the radical envelope-of-noise hypothesis". In: *Adaptive Behavior* 6.2 (1997), pp. 325–368.

[139] N. Jakobi. "Half-baked, ad-hoc and noisy: Minimal simulations for evolutionary robotics". In: *Fourth European Conference on Artificial Life*. Vol. 4. Brighton, UK, 1997, p. 348.

[140] N. Jakobi, P. Husbands and I. Harvey. "Noise and the reality gap: The use of simulation in evolutionary robotics". In: *Advances in Artificial Life. ECAL 1995*. Granada, Spain, 1995, pp. 704–720.

[141] F. Jenkins. "Practical requirements for a domestic vacuum-cleaning robot". In: *Proceedings of AAAI 1993 Fall Symposium Series: Instantiating Real-World Agents*. 1993, pp. 85–90.

[142] N. K. Jerne. "Towards a network theory of the immune system." In: *Annales d'Immunologie* 125.1-2 (1974), p. 373.

[143] S. Józefowski. "The danger model: questioning an unconvincing theory". In: *Immunology and Cell Biology* 94.2 (2016), pp. 164–168.

[144] K. Kaneko. "Chaos as a source of complexity and diversity in evolution". In: *Artificial Life* 1.1_2 (1993), pp. 163–177.

[145] D. Katz, J. Kenney and O. Brock. "How can robots succeed in unstructured environments". In: *Workshop on Robot Manipulation: Intelligence in Human Environments at Robotics: Science and Systems (RSS 2008)*. Atlanta, GA, USA, 2008.

[146] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.

[147] S. Kernbach et al. "Symbiotic robot organisms: REPLICATOR and SYMBRION projects". In: *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*. 2008, pp. 62–69.

[148] K. Kirby. "Context dynamics in neural sequential learning". In: *Proc. Florida AI Research Symposium*. 1991, pp. 66–70.

[149] D. E. Knuth. "Two Notes on Notation". In: *The American Mathematical Monthly* 99.5 (1992), pp. 403–422.

[150] Y. Kochura et al. "Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes". In: *Conference on Computer Science and Information Technologies*. 2017, pp. 243–256.

[151] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vol. 3. Sendai, Japan, 2004, pp. 2149–2154.

[152] S. Kohlbrecher and J. Meyer. *hector_gazebo*. Technische Universität Darmstadt. URL: `https://github.com/tu-darmstadt-ros-pkg/hector_gazebo` (visited on 04/11/2019).

[153] T. Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[154] S. Koos, J.-B. Mouret and S. Doncieux. "The transferability approach: Crossing the reality gap in evolutionary robotics". In: *IEEE Transactions on Evolutionary Computation* 17.1 (2012), pp. 122–145.

[155] E. A. Kravitz. "Hormonal control of behavior: amines and the biasing of behavioral output in lobsters". In: *Science* 241.4874 (1988), p. 1775.

[156] K. Krippendorf. *A Dictionary of Cybernetics*. Tech. rep. 1986.

[157] A. Krizhevsky, I. Sutskever and G. E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems* 25 (2012), pp. 1097–1105.

[158] A. Krogh and J. A. Hertz. "A simple weight decay can improve generalization". In: *Advances in Neural Information Processing Systems*. 1992, pp. 950–957.

[159] J. D. Kropotov and S. C. Etlinger. "Selection of actions in the basal ganglia–thalamocortical circuits: Review and model". In: *International Journal of Psychophysiology* 31.3 (1999), pp. 197–217.

[160] B. J. A. Kröse and M. Eecen. "A self-organizing representation of sensor space for mobile robot navigation". In: *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent*

Robots and Systems '94. Vol. 1. 1994, 9–14 vol.1.

[161] V. Kyrylov, L. A. Severyanova and A. Vieira. "Modeling robust oscillatory behavior of the hypothalamic-pituitary-adrenal axis". In: *IEEE Transactions on Biomedical Engineering* 52.12 (2005), pp. 1977–1983.

[162] P. L'Ecuyer and R. Simard. "TestU01: A C library for empirical testing of random number generators". In: *ACM Transactions on Mathematical Software (TOMS)* 33.4 (2007), p. 22.

[163] J. E. Laird, A. Newell and P. S. Rosenbloom. "Soar: An architecture for general intelligence". In: *Artificial Intelligence* 33.1 (1987), pp. 1–64.

[164] C. G. Langton. "Computation at the edge of chaos: phase transitions and emergent computation". In: *Physica D: Nonlinear Phenomena* 42.1-3 (1990), pp. 12–37.

[165] Y. LeCun. *My take on Ali Rahimi's "Test of Time" award talk at NIPS.* 2017. URL: `https://www.facebook.com/yann.lecun/posts/10154938130592143` (visited on 17/08/2019).

[166] Y. A. LeCun et al. "Efficient backprop". In: *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 9–48.

[167] Y. LeCun, Y. Bengio and G. Hinton. "Deep learning". In: *Nature* 521.7553 (2015), p. 436.

[168] Y. Lecun and C. Cortes. *The MNIST database of handwritten digits*. URL: `http://yann.lecun.com/exdb/mnist/` (visited on 15/08/2019).

[169] Y. LeCun et al. "Backpropagation applied to handwritten zip code recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551.

[170] S. Lee et al. "Evolving Gaits for Physical Robots with the HyperNEAT Generative Encoding: The Benefits of Simulation". In: *European Conference on the Applications of Evolutionary Computation*. Vienna, 2013, pp. 540–549.

[171] W. B. Levy and O. Steward. "Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus". In: *Neuroscience* 8.4 (1983), pp. 791–797.

[172] G. Li, B. Liu and Y. Liu. "A dynamical model of the pulsatile secretion of the hypothalamo-pituitary-thyroid axis". In: *Biosystems* 35.1 (1995), pp. 83–92.

[173] P. J. G. Lisboa and S. J. Perantonis. "Complete solution of the local minima in the XOR problem". In: *Network: Computation in Neural Systems* 2.1 (1991), pp. 119–124.

[174] B. Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Springer Science & Business Media, 2007.

[175] K. Z. Lorenz. "The comparative method in studying innate behavior patterns". In: *Symposia of the Society for Experimental Biology* 4 (1950), pp. 221–268.

[176] K. Z. Lorenz. *The Foundations of Ethology*. Springer-Verlag, 1981.

[177] H. Lövheim. "A new three-dimensional model for emotions and monoamine neurotransmitters". In: *Medical Hypotheses* 78.2 (2012), pp. 341–348.

[178] G. W. Lucas. *A Tutorial and Elementary Trajectory Model for the Differential Steering System of Robot Wheel Actuators*. The Rossum Project. URL: `http://rossum.sourceforge.net/papers/DiffSteer/DiffSteer.html` (visited on 12/10/2017).

[179] A. L. Maas, A. Y. Hannun and A. Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL 2013)*. Atlanta, GA, USA, 2013.

[180] W. Maass, T. Natschläger and H. Markram. "Real-time computing without stable states: A new framework for neural computation based on perturbations". In: *Neural Computation* 14.11 (2002), pp. 2531–2560.

[181] S. Magg and A. Philippides. "GasNets and CTRNNs – a comparison in terms of evolvability". In: *From Animals to Animats 9 (Proceedings of the Ninth International Conference on Simulation of Adaptive Behavior)*. Rome, Italy, 2006, pp. 461–472.

[182] E. Marder and V. Thirumalai. "Cellular, synaptic and network effects of neuromodulation". In: *Neural Networks* 15.4-6 (2002), pp. 479–493.

[183] J. Martens. "Deep Learning via Hessian-free Optimization". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. ICML'10. Haifa, Israel, 2010, pp. 735–742.

[184] L. M. Martinez and J.-M. Alonso. "Complex receptive fields in primary visual cortex". In: *The Neuroscientist* 9.5 (2003), pp. 317–331.

[185] M. T. Mason. "Creation myths: The beginnings of robotics research". In: *IEEE Robotics & Automation Magazine* 19.2 (2012), pp. 72–77.

[186] T. d. J. Mateo Sanguino. "50 years of rovers for planetary exploration: A retrospective review for future directions". In: *Robotics and Autonomous Systems* 94 (2017), pp. 172–185.

[187] H. R. Maturana and F. G. Varela. *Autopoiesis and Cognition: The Realization of the Living*. Springer, 1980.

[188] P. Matzinger. "Tolerance, danger, and the extended family". In: *Annual Review of Immunology* 12.1 (1994), pp. 991–1045.

[189] J. McCarthy and P. J. Hayes. "Some philosophical problems from the standpoint of artificial intelligence". In: *Machine Intelligence* 4 (1969), pp. 463–502.

[190] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.

[191] W. McCune. "Solution of the Robbins problem". In: *Journal of Automated Reasoning* 19.3 (1997), pp. 263–276.

[192] D. McFarland and E. Spier. "Basic cycles, utility and opportunism in self-sufficient robots". In: *Robotics and Autonomous Systems* 20.2-4 (1997), pp. 179–190.

[193] G. McHale. "Adaptive Networks for Robotics and the Emergence of Reward Anticipatory Circuits". PhD thesis. University of Sussex, 2012.

[194] M. Mendao. "A neuro-endocrine control architecture applied to mobile robotics". PhD thesis. Kent University, 2008.

[195] J. S. Milton and J. C. Arnold. *Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences*. 4th. New York, NY, USA: McGraw-Hill, Inc., 2002.

[196] M. L. Minsky. *The Society of Mind*. Simon and Schuster, 1988.

[197] M. L. Minsky and S. A. Papert. *Perceptrons - Expanded Edition: An Introduction to Computational Geometry*. MIT press Boston, MA: 1987.

[198] V. Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), p. 529.

[199] R. C. Moioli et al. “Towards the evolution of an artificial homeostatic system”. In: *IEEE Congress on Evolutionary Computation*. 2008, pp. 4023–4030.

[200] D. J. Montana and L. Davis. “Training Feedforward Neural Networks Using Genetic Algorithms”. In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. 1989, pp. 762–767.

[201] D. E. Moriarty and R. Mikkulainen. “Efficient Reinforcement Learning through Symbiotic Evolution”. In: *Machine Learning* 22.1 (1996), pp. 11–32.

[202] G. J. Morton, T. H. Meek and M. W. Schwartz. “Neurobiology of food intake in health and disease”. In: *Nature Reviews Neuroscience* 15.6 (2014), p. 367.

[203] D. Moser, R. Thenius and T. Schmickl. “First Investigations into Artificial Emotions in Cognitive Robotics”. In: *International Workshop on Medical and Service Robots*. 2016, pp. 213–227.

[204] A. C. Müller, S. Guido et al. *Introduction to Machine Learning With Python: A Guide for Data Scientists*. O’Reilly Media, Inc., 2016.

[205] V. Nair and G. E. Hinton. “Rectified linear units improve restricted Boltzmann machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 807–814.

[206] A. Nanty and R. Gelin. “Fuzzy controlled PAD emotional state of a NAO robot”. In: *Conference on Technologies and Applications of Artificial Intelligence, TAAI 2013*. Taipei, Taiwan, 2013, pp. 90–96.

[207] S. Nason and J. E. Laird. “Soar-RL: Integrating reinforcement learning with Soar”. In: *Cognitive Systems Research* 6.1 (2005), pp. 51–59.

[208] M. Neal and J. Timmis. “Once More Unto the Breach: Towards Artificial Homeostasis?” In: *Recent Developments in Biologically Inspired Computing*. Ed. by L. N. De Castro and F. J. Von Zuben. Idea Group, 2005, pp. 340–365.

[209] M. Neal and J. Timmis. “Timidity: a useful emotional mechanism for robot control?” In: *Informatica (Slovenia)* 27.2 (2003), pp. 197–204.

[210] A. Newell and H. A. Simon. “Computer science as empirical inquiry: symbols and search”. In: *Communications of the ACM* 19.3 (1976), pp. 113–126.

[211] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[212] S. Nolfi and D. Floreano. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. MIT press, 2000.

[213] E. Oja. “Simplified neuron model as a principal component analyzer”. In: *Journal of Mathematical Biology* 15.3 (1982), pp. 267–273.

[214] M. Oku, T. Makino and K. Aihara. “Pseudo-orthogonalization of memory patterns for associative memory”. In: *IEEE Transactions on Neural Networks and Learning Systems* 24.11 (2013), pp. 1877–1887.

[215] N. H. Packard. “Adaptation toward the edge of chaos”. In: *Dynamic Patterns in Complex Systems* 212 (1988), pp. 293–301.

[216] R. Pascanu, T. Mikolov and Y. Bengio. “On the difficulty of training recurrent neural networks”. In: *International Conference on Machine Learning*. Atlanta, USA, 2013, pp. 1310–1318.

[217] Y. Pazos, F. F. Casanueva and J. P. Camiña. "Basic aspects of ghrelin action". In: *Vitamins & Hormones* 77 (2007), pp. 89–119.

[218] B. A. Pearlmutter. *Dynamic recurrent neural networks*. Tech. rep. CMU-CS-90-196. Pittsburgh, PA, USA: School of Computer Science, Carnegie Mellon University, 1990.

[219] H. Peng et al. "Hormone-Inspired Cooperative Control for Multiple UAVs Wide Area Search". In: *ICIC 2008: Advanced Intelligent Computing Theories and Applications: With Aspects of Theoretical and Methodological Issues*. Ed. by D.-S. Huang et al. Shanghai, China, 2008, pp. 808–816.

[220] T. Peng. *LeCun vs Rahimi : Has Machine Learning Become Alchemy ?* A Medium Corporation. 2017. URL: https://medium.com/@Synced/lecun-vs-rahimi-has-machine-learning-become-alchemy-21cb1557920d (visited on 10/08/2019).

[221] C. M. A. Pennartz. "Reinforcement learning by Hebbian synapses with adaptive thresholds". In: *Neuroscience* 81.2 (1997), pp. 303–319.

[222] A. Philippides et al. "Flexible couplings: Diffusing neuromodulators and adaptive robotics". In: *Artificial Life* 11.1-2 (2005), pp. 139–160.

[223] D. M. W. Powers. "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation". In: *Journal of Machine Learning Technologies* 2.1 (2011), pp. 37–63.

[224] W. H. Press et al. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge University Press, 2007.

[225] M. Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. Vol. 3. 3.2. Kobe, Japan, 2009, pp. 5–10.

[226] A. Rahimi. *Test-of-time Award Presentation, NIPS 2017*. 2017. URL: https://www.youtube.com/watch?v=ORHFOnaEzPc (visited on 10/08/2019).

[227] A. Rahimi and B. Recht. *An Addendum to Alchemy*. 2017. URL: http://www.argmin.net/2017/12/11/alchemy-addendum/ (visited on 10/08/2019).

[228] A. Raza and B. R. Fernandez. "Immuno-inspired robotic applications: a review". In: *Applied Soft Computing* 37 (2015), pp. 490–505.

[229] P. Redgrave, T. J. Prescott and K. Gurney. "The basal ganglia: a vertebrate solution to the selection problem?" In: *Neuroscience* 89.4 (1999), pp. 1009–1023.

[230] J. D. M. Rennie. *On L2-norm Regularization and the Gaussian Prior*. Massachusetts Institute of Technology. 2003. URL: http://qwone.com/~jason/writing/l2gaussian.pdf (visited on 10/08/2019).

[231] M. Riedmiller. "Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method". In: *European Conference on Machine Learning*. Vol. 3720 LNAI. Porto, Portugal, 2005, pp. 317–328.

[232] M. Röder. *Precision, Recall and the F1 measure*. DICE Group, Paderborn University. 2015. URL: https://github.com/dice-group/gerbil/wiki/Precision,-Recall-and-F1-measure (visited on 10/08/2019).

[233] G. Rodriguez and C. R. Weisbin. "A new method to evaluate human-robot system performance". In: *Autonomous Robots* 14.2-3 (2003), pp. 165–178.

[234] P. Rohlfshagen and J. J. Bryson. "Flexible latching: A biologically-inspired mechanism for improving the management of homeostatic goals". In: *Cognitive Computation* 2.3 (2010), pp. 230–241.

[235] A. Roli et al. "Dynamical criticality: overview and open questions". In: *Journal of Systems Science and Complexity* 31.3 (2018), pp. 647–663.

[236] F. Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), p. 386.

[237] D. E. Rumelhart, G. E. Hinton and R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536.

[238] D. E. Rumelhart, J. L. McClelland and R. J. Williams. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Ed. by D. E. Rumelhart and J. L. McClelland. Vol. 1. Computational Models of Cognition and Perception 1. MIT Press, 1986.

[239] I. Sakata and T. Sakai. "Ghrelin cells in the gastrointestinal tract". In: *International Journal of Peptides* 2010 (2010).

[240] T. D. Sanger. "Optimal unsupervised learning in a single-layer linear feedforward neural network". In: *Neural Networks* 2.6 (1989), pp. 459–473.

[241] W. S. Sarle. *Neural network FAQ (Periodic posting to the Usenet newsgroup comp.ai.neural-nets)*. SAS Institute. 1997. URL: `ftp://ftp.sas.com/pub/neural/FAQ.html` (visited on 10/08/2019).

[242] C. Sauzé and M. Neal. "A neuro-endocrine inspired approach to long term energy autonomy in sailing robots". In: *Proceedings of TAROS (Towards Autonomous Robotic Systems)*. Plymouth, UK, 2010, pp. 255–262.

[243] C. Sauzé and M. Neal. "Artificial endocrine controller for power management in robotic systems". In: *IEEE Transactions on Neural Networks and Learning Systems* 24.12 (2013), pp. 1973–1985.

[244] C. Sauzé and M. Neal. "Long term power management in sailing robots". In: *Oceans 2011*. Santander, Spain, 2011, pp. 1–8.

[245] J. B. Saxon and A. Mukerjee. "Learning the motion map of a robot arm with neural networks". In: *IJCNN International Joint Conference on Neural Networks*. 1990, 777–782 vol.2.

[246] J. D. Schaffer, D. D. Whitley and L. J. Eshelman. "Combinations of genetic algorithms and neural networks: A survey of the state of the art". In: *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*. Baltimore, MD, USA, 1992, pp. 1–37.

[247] T. Schmickl, H. Hamann and K. Crailsheim. "Modelling a hormone-inspired controller for individual-and multi-modular robotic systems". In: *Mathematical and Computer Modelling of Dynamical Systems* 17.3 (2011), pp. 221–242.

[248] J. Schmidhuber. “Deep Learning in Neural Networks: An Overview”. In: *Neural Networks* 61 (2015), pp. 85–117.

[249] W. Schultz. “Predictive reward signal of dopamine neurons”. In: *Journal of Neurophysiology* 80.1 (1998), pp. 1–27.

[250] M. J. Schuster et al. “Towards Autonomous Planetary Exploration”. In: *Journal of Intelligent & Robotic Systems* 93.3-4 (2019), pp. 461–494.

[251] H. Schütze, C. D. Manning and P. Raghavan. *Introduction to Information Retrieval*. Vol. 39. Cambridge University Press, 2008.

[252] J. R. Searle. “Minds, brains, and programs”. In: *Behavioral and Brain Sciences* 3.3 (1980), pp. 417–424.

[253] A. K. Seth. “The cybernetic Bayesian brain”. In: *Open MIND*. Ed. by T. K. Metzinger and J. M. Windt. MIND Group, 2014, pp. 1–24.

[254] A. K. Seth, J. J. Bryson and T. J. Prescott. “Introduction. Modelling natural action selection”. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 362.1485 (2007), pp. 1521–1529.

[255] W. M. Shen et al. “Hormone-inspired self-organization and distributed control of robotic swarms”. In: *Autonomous Robots* 17.1 (2004), pp. 93–105.

[256] W.-M. Shen, C.-M. Chuong and P. Will. “Digital Hormone Models for Self-Organization”. In: *Artificial Life VIII*. 2003, pp. 116–120.

[257] P. Smolensky. “On the proper treatment of connectionism”. In: *Philosophy, Mind, and Cognitive Inquiry*. Springer, 1990, pp. 145–206.

[258] A. Soltoggio, K. O. Stanley and S. Risi. “Born to Learn: the Inspiration, Progress, and Future of Evolved Plastic Artificial Neural Networks”. In: *Neural Networks* 108 (2018), pp. 48–67.

[259] A. Soltoggio et al. “Evolving neuromodulatory topologies for reinforcement learning-like problems”. In: *2007 IEEE Congress on Evolutionary Computation, CEC 2007*. 2007, pp. 2471–2478.

[260] S. Song, K. D. Miller and L. F. Abbott. “Competitive Hebbian learning through spike-timing-dependent synaptic plasticity”. In: *Nature Neuroscience* 3.9 (2000), p. 919.

[261] I. G. Sprinkhuizen-Kuyper and E. J. W. Boers. “The error surface of the simplest XOR network has only global minima”. In: *Neural Computation* 8.6 (1996), pp. 1301–1320.

[262] K. O. Stanley, B. D. Bryant and R. Miikkulainen. “Real-time neuroevolution in the NERO video game”. In: *IEEE Transactions on Evolutionary Computation* 9.6 (2005), pp. 653–668.

[263] K. O. Stanley, D. B. D’Ambrosio and J. Gauci. “A hypercube-based encoding for evolving large-scale neural networks.” In: *Artificial Life* 15.2 (2009), pp. 185–212.

[264] K. O. Stanley and R. Miikkulainen. “Evolving neural networks through augmenting topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127.

[265] A. Stocco. “A Biologically Plausible Action Selection System for Cognitive Architectures: Implications of Basal Ganglia Anatomy for Learning and Decision-Making Models”. In: *Cognitive Science* 42.2 (2018), pp. 457–490.

[266] A. Storkey. "Increasing the capacity of a Hopfield network without sacrificing functionality". In: *International Conference on Artificial Neural Networks*. 1997, pp. 451–456.

[267] J. Stradner et al. "Evolving a novel bio-inspired controller in reconfigurable robots". In: *Advances in Artificial Life (ECAL 2009)*. Budapest, Hungary: Springer, 2009, pp. 132–139.

[268] H. Strömfelt, Y. Zhang and B. W. Schuller. "Emotion-augmented machine learning: Overview of an emerging domain". In: *7th International Conference on Affective Computing and Intelligent Interaction, ACII 2017*. 2017, pp. 305–312.

[269] L. Suchman. *Plans and Situated Actions*. Cambridge University Press, 1987, p. 224.

[270] I. Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *International Conference on Machine Learning*. Atlanta, GA, USA, 2013, pp. 1139–1147.

[271] R. S. Sutton. "Learning to predict by the methods of temporal differences". In: *Machine Learning* 3.1 (1988), pp. 9–44.

[272] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Vol. 1. 1. Cambridge, MA, USA: MIT Press, 1998.

[273] M. Talanov and A. Toschev. "Computational emotional thinking and virtual neurotransmitters". In: *International Journal of Synthetic Emotions* 5.1 (2014), pp. 1–8.

[274] G. Tanaka et al. "Recent advances in physical reservoir computing: A review". In: *Neural Networks* 115 (2019), pp. 100–123.

[275] P. Teerakittikul. "Artificial Hormone Network for Adaptable Robots". PhD thesis. York, 2013.

[276] P. Teerakittikul, G. Tempesti and A. M. Tyrrell. "Artificial hormone network for adaptive robot in a dynamic environment". In: *Conference on Adaptive Hardware and Systems (AHS), 2012 NASA/ESA*. 2012, pp. 129–136.

[277] R. Thenius, P. Zahadat and T. Schmickl. "EMANN - a model of emotions in an artificial neural network". In: *Advances in Artificial Life (ECAL 2013)*. Vol. 12. 2013, pp. 830–837.

[278] J. Timmis, L. Murray and M. Neal. "A Neural-Endocrine Architecture for Foraging in Swarm Robotic Systems". In: *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*. Springer, 2010, pp. 319–330.

[279] J. Timmis, M. Neal and J. Thorniley. "An adaptive neuro-endocrine system for robotic systems". In: *IEEE Workshop on Robotic Intelligence in Informationally Structured Space, 2009 (RIISS'09)*. 2009, pp. 129–136.

[280] N. Tinbergen. *The Study of Instinct*. Oxford, UK: Clarendon Press, 1951.

[281] S. S. Tomkins. *Affect Imagery Consciousness: The Complete Edition: Two Volumes*. Springer Publishing Company, 2008.

[282] E. Tunstel. "Operational performance metrics for Mars exploration rovers". In: *Journal of Field Robotics* 24.8-9 (2007), pp. 651–670.

[283] A. M. Turing. "The chemical basis of morphogenesis". In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 237.641 (1952), pp. 37–72.

[284] T. Tyrrell. "Computational mechanisms for action selection". PhD thesis. University of Edinburgh, Scotland, 1993.

[285] F. J. Valverde-Albacete and C. Peláez-Moreno. "100% Classification Accuracy Considered Harmful: The Normalized Information Transfer Factor Explains the Accuracy Paradox". In: *PLoS ONE* 9.1 (2014). Ed. by M. G. A. Paris, e84217.

[286] P. A. Vargas, E. A. Di Paolo and P. Husbands. "A study of GasNet spatial embedding in a delayed-response task." In: *Artificial Life XI, Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*. 2008, pp. 640–647.

[287] P. A. Vargas, E. A. Di Paolo and P. Husbands. "Preliminary investigations on the evolvability of a non-spatial GasNet model". In: *Advances in Artificial Life (ECAL 2007)*. Springer, 2007, pp. 966–975.

[288] P. A. Vargas et al. *The Horizons of Evolutionary Robotics*. MIT press, 2014.

[289] P. A. Vargas et al. "Artificial homeostatic system: A novel approach". In: *Advances in Artificial Life (ECAL 2005)* (2005), pp. 754–764.

[290] P. A. Vargas et al. "Homeostasis and evolution together dealing with novelties and managing disruptions". In: *International Journal of Intelligent Computing and Cybernetics* 2.3 (2009), pp. 435–454.

[291] F. Vaussard et al. "Cutting down the energy consumed by domestic robots: Insights from robotic vacuum cleaners". In: *Towards Autonomous Robotic Systems*. Bristol, UK, 2012, pp. 128–139.

[292] P. F. M. J. Verschure and P. Althaus. "The study of learning and problem solving using artificial devices: Synthetic Epistemology". In: *Bildung Und Erziehung* 52.3 (1999), pp. 317–334.

[293] T. P. Vogl et al. "Accelerating the Convergence of the Back-Propagation Method". In: *Biological Cybernetics* 59.4-5 (1988), pp. 257–263.

[294] K. Wada et al. "Effects of robot-assisted activity for elderly people and nurses at a day service center". In: *Proceedings of the IEEE* 92.11 (2004), pp. 1780–1788.

[295] J. Walker and M. Wilson. "A performance sensitive hormone-inspired system for task distribution amongst evolving robots". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2008, pp. 1293–1298.

[296] J. Walker and M. Wilson. "Hormone-inspired control for group task-distribution". In: *Proceedings of TAROS (Towards Autonomous Robotic Systems)* (2007), pp. 1–8.

[297] C. J. C. H. Watkins and P. Dayan. "Q-learning". In: *Machine Learning* 8.3-4 (1992), pp. 279–292.

[298] P. Werbos. "Beyond regression: New tools for prediction and analysis in the behavioral sciences". PhD thesis. Harvard, 1974.

[299] P. J. Werbos. "Generalization of backpropagation with application to a recurrent gas market model". In: *Neural Networks* 1.4 (1988), pp. 339–356.

[300] B. Widrow and M. E. Hoff. *Adaptive switching circuits*. Tech. rep. 1553-1. Stanford Electronics Laboratories, Stanford University, 1960.

[301] B. Widrow and M. A. Lehr. "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1415–1442.

[302] R. J. Williams and D. Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks". In: *Neural Computation* 1.2 (1989), pp. 270–280.

[303] D. R. Wilson and T. R. Martinez. "The general inefficiency of batch training for gradient descent learning". In: *Neural Networks* 16.10 (2003), pp. 1429–1451.

[304] A. Wolf et al. "Determining Lyapunov exponents from a time series". In: *Physica D: Nonlinear Phenomena* 16.3 (1985), pp. 285–317.

[305] Q.-z. Xu and L. Wang. "Recent advances in the artificial endocrine system". In: *Journal of Zhejiang University Science C* 12.3 (2011), pp. 171–183.

[306] M. Yamamoto. "Sozzy: A hormone-driven autonomous vacuum cleaner". In: *Proceedings of the SPIE Mobile Robots VIII Conference*. 1993, pp. 292–305.

[307] J. Yoder. "Evolving neuromodulator architectures on non-associative learning tasks". In: *IEEE Symposium Series on Computational Intelligence (SSCI)*. 2017, pp. 1–9.

Colophon

This thesis was typeset in pdfLaTeX3.14159265-2.6-1.40.18, using the Palatino-like fonts supplied by the `newpx` package. The code herein was largely written in C++, taking the form of a set of shared libraries acting as plugins for the Angort concatenative language. Plots were generated largely with R and Angort, and explanatory figures were drawn in Inkscape by the author. Photographs were provided by the author.

The git commit ID for this version is `d9a433f` .