Connectionist Adaptive Control

Timothy Tristram Jervis

Trinity Hall
Cambridge
England

December 1993

This dissertation is submitted for consideration for the degree of Doctor of Philosophy at the University of Cambridge

Summary

Machines that perform difficult, mundane or dangerous tasks for us add to our quality of life. Our lives might be further improved by making machines, and controllers for them, that learn to be more capable.

Learning controllers aim to avoid the need for complex design techniques by embodying the exploration strategy of the control engineer. They should perform better than non-adaptive controllers by finding better control policies. Learning controllers might also offer solutions to problems that have so far resisted conventional approaches.

This work considers a general framework for learning control, known as reinforcement learning. It documents the first application of a reinforcement learning controller to the task of regulating an inverted pendulum in hardware. It explores the application of non-linear parametric models known as connectionist models, or neural networks, to learning control. It approaches learning control as an optimization problem, and proposes a promising new learning control algorithm that uses neural networks.

Acknowledgements

There is a long list of people I would like to thank. At the top of the list is the late Prof. Frank Fallside, who created the friendly and open environment at Cambridge where this work took place. He was my supervisor until his untimely death earlier this year.

I am delighted to acknowledge Dr. Bill Fitzgerald, who has taken over the rôle of my supervisor. I am grateful for Bill's advice and encouragement.

I owe a debt to every member of the Speech, Vision and Robotics group at Cambridge University Engineering Department. I cannot imagine a more desirable place in which to work. Some members deserve a special mention: Patrick Gosling, Tony Robinson, Richard Prager, Andrew Piper, Andrew Gee and Carl Seymour for maintaining the excellent computing resources at the lab; and Dave Gautrey and Alan Thorne for their help with the special hardware I used.

I am grateful to the Free Software Foundation and others for providing emacs, LaTeX, gnuplot and bash.

Tempting though it is to mention all my friends who make my life worth living, in the context of this thesis I must limit myself to mentioning those that have had a direct impact on my intellectual development, such as it is. In particular, I would like to mention my friends (in the order of longest acquaintance) David MacKay, Andrew Senior and Ian Braithwaite.

I thank the Science and Engineering Research Council, and especially my parents, for their substantial financial support.

My final words are to my parents. They always put myself and my brother first. They have given me the freedom and support to find my happiness. It is to my parents that I dedicate this work.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration with any other party.

Length

This dissertation is a by-product of cycling 15,000 miles in and out of Cambridge. It was written at a rate of two words per mile.

Contents

Chapter 1

Introduction

The task of control is to apply inputs to a plant so that the plant performs in a desirable way. There are many control problems, of varying difficulty, from turning off a kettle once the water has boiled to staying alive and happy in the modern world. A control engineer normally designs a controller by tailoring a control paradigm to a particular problem, for example by setting the parameters of a PI or PID controller[1]. This approach is tried and tested in industry. However, there are more challenging control tasks that require a more ambitious approach: tasks such as making a robot that can iron a shirt, or do the gardening. Learning control aims eventually to solve these more challenging tasks.

Connectionist models are non-linear models parametrized by a set of weights. They were born out of work concerned with the structure of the brain, and are for this reason called neural networks. They have since found a place in the toolbox of the engineer alongside other non-linear models. They are interesting in the context of this work because they are capable of forming multiple-input, multiple-output non-linear mappings parametrized by a finite set of weights. Control laws formed by neural networks are therefore potentially widely applicable.

This chapter introduces learning control, and describes some kinds of learning, and what is meant by reinforcement learning. It then covers some problems that have been approached by machine learning, and finally outlines the remainder of this work.

1.1 Learning is an optimization problem

A controller makes an observation of the environment, which conditions the action the controller will take. The plant and controller combination become a new autonomous system. The performance of this combination is valued by an objective function. Figure 1.1 depicts how a controller interacts with its environment.

Learning control is an optimization problem. Optimization algorithms find points in a space where an objective function has an extremum. In learning control, the space to search in is a space of control policies. The objective function is some measure of the controller's performance.

The problem of learning control is to find an efficient optimization algorithm for the kinds of objective function of interest.

1.1.1 Supervised, unsupervised and reinforcement learning

Learning is traditionally divided into two types: supervised and unsupervised. Supervised learning makes use of a teacher that supplies input/target pairs. The learner's ability to map the inputs to the targets is measured by an objective function. The parameters of the learner are adjusted to optimize this performance measure. Unfortunately, in learning control there are no explicit training targets.

[1]Propotional Integral (PI) or Proportional Integral Derivative (PID) (Åström and Wittenmark 1989).

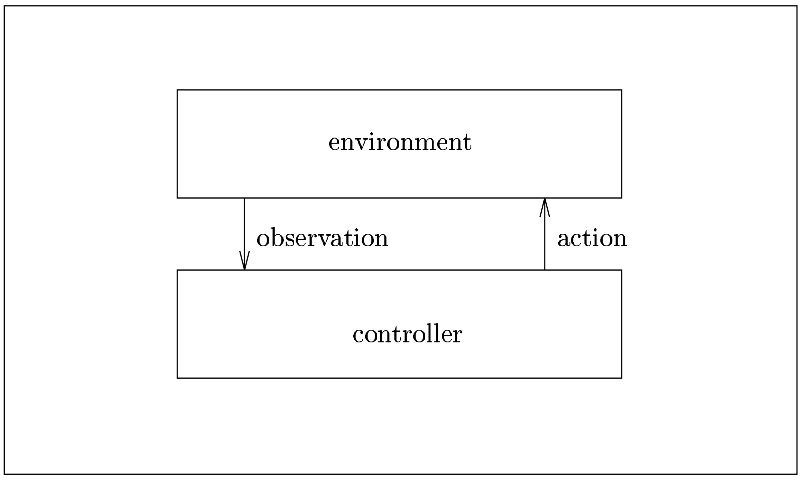


Figure 1.1: An autonomous system formed by the application of a controller to its environment.

Unsupervised learning uses an objective function built-in to the learning agent to group together exemplars of various kinds. For example, this kind of learning could be used to cluster together sensory examples from the same object, and differentiate them from examples from other objects.

Both supervised and unsupervised learning are optimization problems. Both have an objective function to be optimized.

The term 'reinforcement learning' comes from the field of psychology, in the context of understanding animal behaviour. It is neither in the class of supervised learning nor unsupervised learning. In reinforcement learning, one specifies positive reinforcement for the kind of behaviour one would like to see, and negative reinforcement, or penalties, for undesirable behaviour. Ideally, this is all the work required to implement a reinforcement learning controller. The attractiveness of reinforcement learning can now be seen: it is potentially easy to apply to new problems, and is furthermore widely applicable.

Unfortunately maximizing positive reinforcement as discussed above is an ill-posed problem, because the reinforcement from a control policy is a time series. Some function of the time series of reinforcement is needed to reduce the series to a scalar. Only then can alternative policies be ranked in preference, so an optimum may be selected. A function that collapses the time series of reinforcement into a scalar can serve as the objective function of the reinforcement learning optimization problem. However, to make one evaluation of the objective function takes an infinite amount of time (the time needed to generate a complete time series of reinforcement).

Some problems that are formulated as reinforcement learning problems can be approached as more straightforward optimization problems. When the problem breaks down into a series of trials, a useful objective function is frequently the duration of each trial. In this case, an evaluation of the objective function takes the length of a trial. Game playing, pole balancing, lorry reversing and peg-in-hole tasks may be considered as a series of trials. Life, on the other hand, is more like a reinforcement learning problem in that only one time series of reinforcement will ever be available, forcing the expedient optimization algorithm to adjust the policy before the first and only evaluation of the objective function is available.

Reinforcement learning problems are approached by making inferences about the value of the objective function at a point in policy space, based on a segment of the reinforcement time series. This assessment is then used to modify the control policy towards one that is more likely to be near an optimum.

1.2 Reinforcement and machine learning problems

Several problems have been tackled by reinforcement learning and machine learning. A popular one is regulating an inverted pendulum (also known as pole balancing). Michie and Chambers attempted the problem using their 'boxes' paradigm (Michie and Chambers 1968), later improved upon by Sutton, Barto and Anderson's ASE/ACE controller (Barto, Sutton and Anderson 1983). Anderson has applied neural networks to the task (Anderson 1989, Anderson 1993). Jordan and Jacobs used two neural networks in their forward-modelling solution (Jordan and Jacobs 1990). Jang also uses a restricted form of the problem as a platform for an adaptive fuzzy-logic controller (Jang 1992).

Moore has approached a number of different machine learning problems, including manipulating a two-jointed dynamic robot, juggling balls (Moore 1991), and playing billiards (Moore 1992). Moore has used kD-Trees (Bentley 1975) as function approximators. A kD-Tree is a form of look-up table. All the data are stored explicitly. The kD-Tree allows for fast nearest-neighbour searches in the data, but at a cost of balancing the tree occasionally as more data arrive. A disadvantage of look-up table methods is the problem of coping with noisy data. Moore has also investigated the use of these memory-based function approximators using cross-validation methods to avoid the problems of noise (Moore and Atkeson 1992).

Barto et al. have worked on the race-track problem (Barto, Bradtke and Singh 1993). A car moves from state to state around a track towards a goal. The car is penalized for hitting the edge of the track and failing to reach the goal. A feature of the problem is that it can have a large number of states.

Nguyen and Widrow used neural networks to learn to reverse a simulated articulated lorry into a loading bay (Nguyen and Widrow 1989, Nguyen and Widrow 1990). Thrun uses a robot navigation task to illustrate the trade-off between exploration and exploitation in machine learning (Thrun 1992). Tham and Prager have approached controlling a simulated robot arm as a reinforcement learning task (Tham and Prager 1992). The arm is penalized for collisions by an amount proportional to their velocity. Tham and Prager use a neural network known as a Cerebellar Model Articulation Controller, or CMAC (Albus 1975), a popular function approximator for reinforcement learning. The function approximator uses a fixed mapping from the sensory input to a set of overlapping cells. The output of the function is a weighted sum of the activities of the cells. The weights of the weighted sum are the adaptive element of the CMAC.

Neural networks have been applied to real apparatus, commonly to robots. Van der Smagt has used feed-forward neural networks to control a robot arm to position itself over an object using a camera as the sensory input (van der Smagt 1991). Miller has used two CMAC networks to control the orientation of the image of an object in the field of view of a camera mounted on a robot arm (Miller III 1989). Mel has used a neural network approach to control the trajectory of a robot arm through obstacles using video feedback (Mel 1989).

Another application of neural networks is in the field of process control. One small-scale example of this is Khalid and Omatu's use of a neural controller for regulating the temperature of a water bath (Khalid and Omatu 1992).

Machine learning is conventionally applied to games. An early example is Samuel's work on the game of draughts (Samuel 1963). The most impressive application of reinforcement learning and neural network learning is Tesauro's work on the game of backgammon (Tesauro 1991). Tesauro's backgammon program is ranked as one of the strongest players in the world.

1.3 An outline of this work

This work casts machine learning as an optimization problem. Chapter 2 looks at two successful reinforcement learning strategies, from Barto, Sutton and Anderson (Barto et al. 1983), and from Watkins (Watkins 1989), and verifies their results. Chapter 3 is a review of neural networks, introducing some of the concepts needed for chapter 4, which moves out of the finite state and action spaces of chapter 2 and describes some of the techniques that have been developed by others for the continuous domain, based on neural networks.

Chapter 5 documents the first successful application of reinforcement learning to the inverted pendulum problem in hardware. The application used the finite techniques of chapter 2 because the reported results of the continuous techniques indicated that the schemes would not have converged within the constraints of the real apparatus.

The experience of chapter 5 motivated a return to the continuous domain, in a search for a more adaptive learning paradigm. Thus chapter 6 describes the use of some optimization techniques to the problem of finding a continuous controller for the inverted pendulum task. The chapter contains a description and some preliminary results for a new learning control scheme, based on neural networks.

A summary and the conclusions are given in chapter 7. Some suggestions for future work in this area may be found in chapter 8. The appendices give additional information on some points covered in the main text.

Chapter 2

Finite space controllers

A significant result in reinforcement learning was the inverted pendulum controller of Barto et al. (Barto et al. 1983). The next significant step forward, Q-learning, was introduced by Watkins (Watkins 1989). This chapter introduces these two devices, after a description of the finite Markov control problem and of dynamic programming.

2.1 The Markov control problem

An agent operates in an environment over discrete time. At any time t, the agent is in a state $s(t) \in S$, where S is a finite set of states. The agent must choose an action $u(t) \in U(s(t))$ at each time step, where $U(s)$ is the set of admissible control actions when the plant is in state s. Taking an action u in state s incurs a cost $c(u, s, w)$, where w is a random disturbance. The Markov property of the problem is that the cost depends on the current state $s(t)$ and not any previous states.

A policy π is a sequence of functions $\{\mu(0), \mu(1), ...\}$ that map the state space S into action space for each time step. A stationary policy is thus a policy for which $\mu(0) = \mu(1) = ...$ Following a policy π over time incurs a series of costs $c(t)$. In order to assess the quality of different policies, a policy's cost sequence must be collapsed into a scalar. Unfortunately the expected value of the sum

$$c(0) + c(1) + \ldots = \sum_{t=0}^{\infty} c(t) \tag{2.1}$$

is usually unbounded. A common way of bounding the sum is through discounting the costs, as follows:

$$J_\pi(s(0)) = \lim_{N \to \infty} E\left(\sum_{t=0}^{N} \lambda^t c(t)\right) \tag{2.2}$$

where $s(0) \in S$ is the state at time $t = 0$, $0 < \lambda < 1$ is a discount factor and the expectation is taken over the disturbance w in the cost expression.

Discounting is usually justified on the grounds of mathematical convenience. It is useful in the field of economics because it models the growth of capital with interest. It also models a constant probability of terminating a control period at each discrete time interval.

The use of discounting, unfortunately, alters the view of the reinforcement problem seen by the controller. It discounts long-term reward in favour of short-term gain. But in general, the controller should forsake short term gain for the benefit of improved performance in the long term. Furthermore, the reinforcement schedule is a form of specification of the problem for the controller. To have a discounted cost imposed on the reinforcement sequence is to alter the specification of the problem.

An alternative method for rendering a reinforcement sequence into a scalar is to use the long-term average cost:

$$\kappa = \lim_{N \to \infty} \frac{1}{N} \sum_{t=0}^{N} c(t) \tag{2.3}$$

Although attractive as a cost criterion, the long-term average cost is more difficult to manipulate. It is discussed by Howard (Howard 1960) and Bertsekas (Bertsekas 1987).

2.1.1 The temporal credit assignment problem

Costs may be distant in space and time, presenting two kinds of credit assignment problems. The first is how to answer the question "Of the actions I took then, which ones were responsible for the cost I received?". The second is the question "Of the actions I have already taken over time, which ones were responsible for the cost I received?". Both questions must be answered before the agent can efficiently change its actions for the better, in the context of Markov decision tasks.

The Markov decision problem outlined above is difficult because actions have to be taken with regard to their long-term effect. The existence of state in the problem imposes constraints on the availability of low costs. This shows itself in reinforcement learning contexts as the problem of temporal credit assignment.

In a reinforcement learning context, an agent takes a series of control actions, and at the same time receives a series of costs. In order to generate a desirable series of costs (minimized in some way), the agent must assess the cause and effect relationship between the actions it took and the costs it experienced. When actions and their effects are separated in time, this is known as the temporal credit assignment problem. A similar problem exists when several actions are taken at once (a vector of actions) and one has to discover which actions were responsible for the outcome. This is a spatial credit assignment problem.

Dynamic programming can be used in reinforcement learning contexts to solve the temporal credit assignment problem. Sutton has used the term temporal-difference, or TD, in association with this problem and its solutions (Barto, Sutton and Watkins 1990).

2.2 Dynamic programming

Dynamic programming, introduced by Bellman, is an optimization method (Bellman 1957). It translates the problem of getting performance in the long-term into two smaller tasks. The first task is to produce a function, denoted by J, that represents the long-term effect of taking choices. The second task is to make choices that optimize the J function in the short term (choices that are 'greedy' with respect to J).

The problem of learning control is to optimize the policy, π, to minimize the discounted cost. That is, find:

$$\pi^* = \arg\min_{\pi} J_\pi. \tag{2.4}$$

Dynamic programming offers a solution to this problem through an intermediate function J^*. J^* is a solution to the optimality equation:

$$J^*(i) = \min_{u \in U(i)} \left\{ c(i,u) + \lambda \sum_{j \in S} p(i,j,u) J^*(j) \right\} \tag{2.5}$$

where $p(i,j,u)$ is the probability of a state transition from state i to state j under action u, and $c(i,u)$ is the cost of action u in state i.

The actions that produce the required minimum in equation 2.5 define an optimal policy for minimizing the infinite discounted sum of costs for each state. For the infinite horizon problem, the optimal policy is stationary, and is thus a fixed mapping from the state space to the action space. For a finite horizon, the optimal policy is generally not stationary, as can be seen from the intuitive idea that as the end of the control period approaches, different actions may become desirable even though the state is the same.

2.2.1 The Bertsekas formalism

Bertsekas introduces a convenient notation and formalism for discussing dynamic programming (Bertsekas 1987). For the system

$$x(t+1) = f(x(t), u(t), v(t)) \tag{2.6}$$

where $x(t)$ is the state at time t, $u(t)$ is the control action and $v(t)$ is a random disturbance parameter, he introduces the following operator T:

$$T(J)(x) = \min_{u \in U(x)} E_w \left\{ g(x,u,w) + \lambda J(f(x,u,v)) \right\} \tag{2.7}$$

where $J(x)$ is a function mapping states $x \in S$ into costs; $g(x,u,w)$ is a transition cost, a function of the state x, control action u and a disturbance w; and λ is a discount factor. The expectation is taken over the disturbances w and v.

The operator T returns a function over the states S. $T(J)(x)$ is the optimal cost for the one-step Markov decision problem starting in state x, with transition cost g and terminal cost function λJ.

Let the composition of the operator with itself k times be denoted by T^k, such that

$$T^k(J)(x) = T(T^{k-1}(J))(x) \tag{2.8}$$

and

$$T^0(J)(x) = J(x). \tag{2.9}$$

The optimal long-term costs are then given by

$$J^*(x) = \lim_{k \to \infty} T^k(J)(x). \tag{2.10}$$

Equation 2.5 can be solved for the optimal policy, i.e. the optimal action u for every state i, using dynamic programming (Narendra and Thathachar 1989).

2.3 The ASE/ACE controller

Michie and Chambers made an early attempt to get a machine to learn. The task they set themselves was to balance an inverted pendulum (Michie and Chambers 1968). Barto et al. attempted to improve on this (Barto et al. 1983). They described their controller as 'neuron-like', although the weights of the controller were only used linearly. Their system converged in fewer trials than that of Michie and Chambers.

Figure 2.1 shows the general structure of the ASE/ACE controller and its relationship with the plant, the inverted pendulum.

2.3.1 The simulated inverted pendulum

The simulated pendulum is shown in figure 2.2. It comprised a straight, horizontal track, like a railway track, with a carriage free to move along it. On the carriage was an axis, perpendicular to the track and pointing out to the side, about which a pendulum was free to turn. The controller's task was to keep the pendulum upright, by alternately pushing and pulling the carriage along the track.

The carriage's position, its velocity, the angle of the pendulum from the vertical and the pendulum's angular velocity made up the state of the system. The simulation equations mapped the

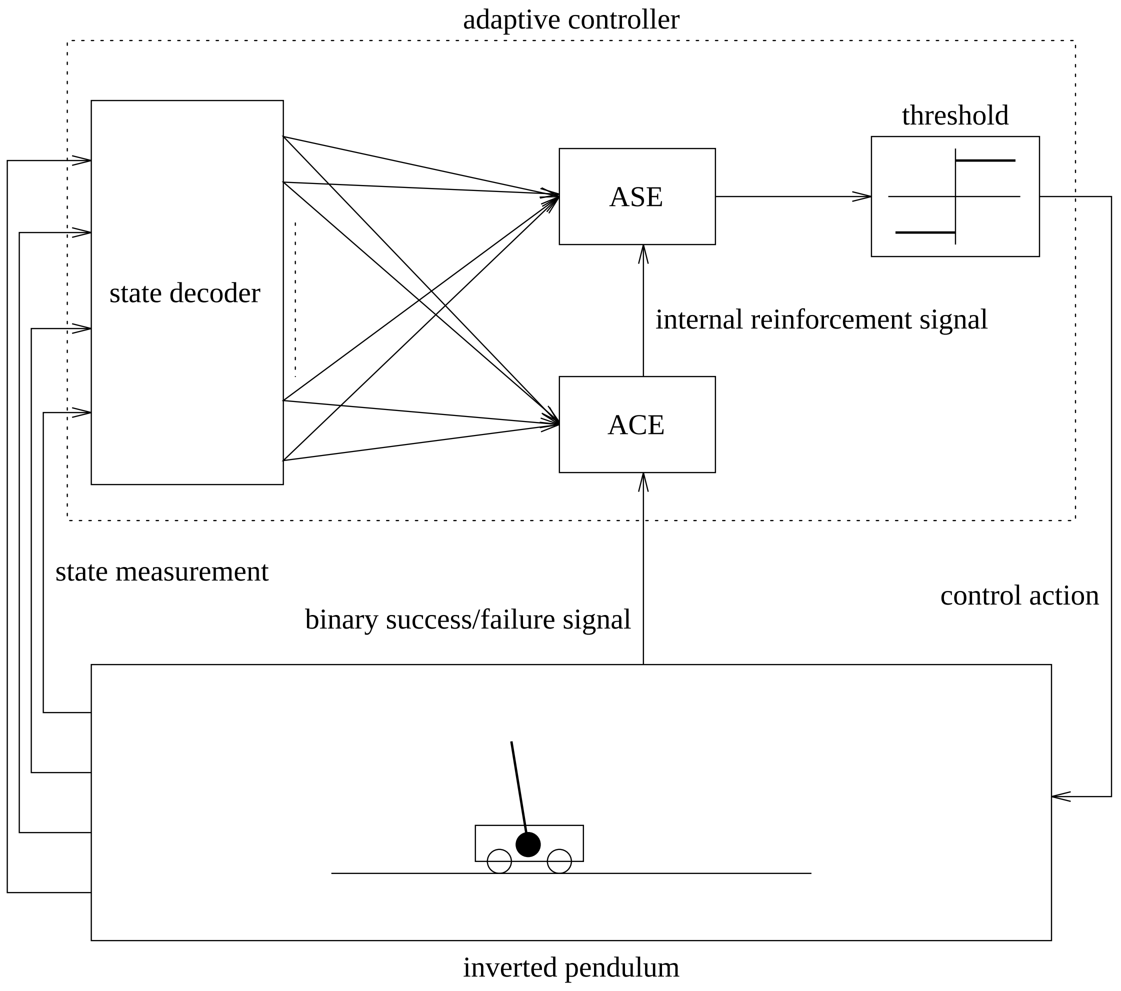


Figure 2.1: The ASE/ACE controller in place.

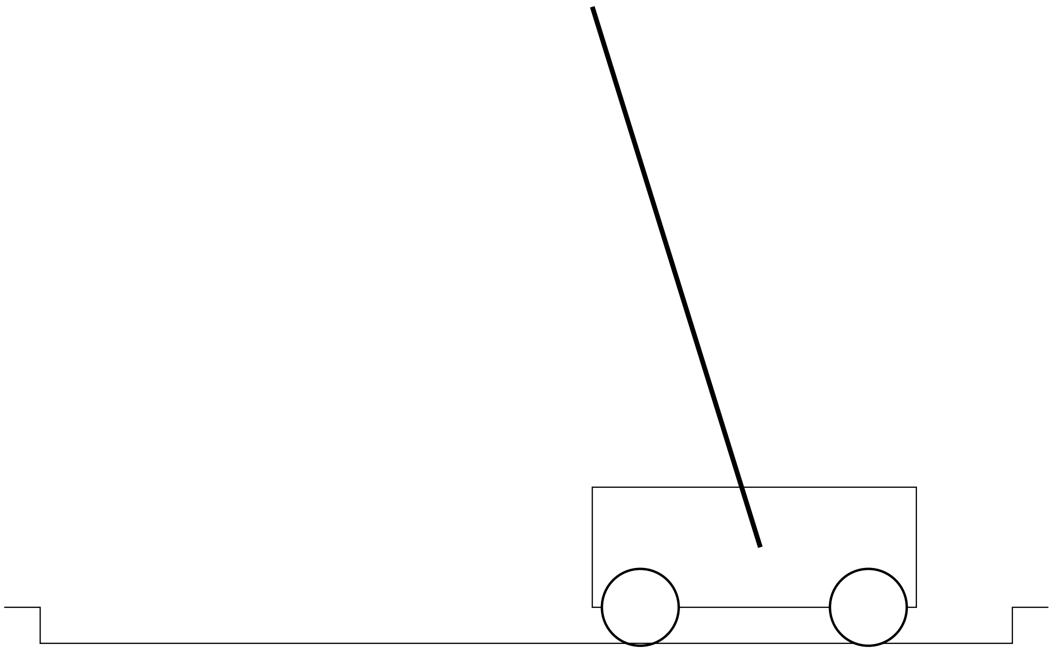


Figure 2.2: The inverted pendulum.

system's state and the control action into a linear acceleration of the carriage and an angular acceleration of the pendulum, which were then integrated using Euler's method to generate the next state vector.

The equations governing the simulation are given in appendix B. The parameters for the simulation are the Barto parameters also detailed in that appendix.

The difficulty of pendulum balancing

Regulating an inverted pendulum can be solved by a linear state feedback controller. This suggests that balancing the pendulum is not difficult. A more difficult problem, with a correspondingly more interesting solution, arises when incomplete state information is used. Hecht-Nielsen cites Tolat and Widrow attempting this more difficult problem (Hecht-Nielsen 1990). They used a video image of the inverted pendulum system as the input to their inverted pendulum controller, trained off-line using a simulation. Even in its more straightforward form, however, the inverted pendulum problem still has a use in assessing learning controllers, since a learning algorithm of satisfactory flexibility and speed has yet to be found for the problem.

2.3.2 The reinforcement signal

The simplicity of the external reinforcement signal demonstrated the attractive goal-directed nature of the scheme. The signal was always zero, except if the pendulum was out of bounds (more than $12°$ from the upward vertical or 2.4 m from the centre of the track), when the reinforcement was minus one.

Reinforcement can either be crude or exacting. A crude reinforcement measure, often used in, for example, the inverted pendulum task, distinguishes good from bad performance and little else. A reinforcement controller using this performance measure can give a number of good solutions. However, the solution may be criticized as giving a performance with no fixed point (van Luenen, de Jager, van Amerongen and Franken 1993). For example, an inverted pendulum regulator may avoid the negative reinforcement of a collapse by placing the pendulum at an arbitrary position along the track, so long as this is within bounds. This may be unsatisfactory.

More exacting control may come from a more specific reinforcement signal. This signal, although more difficult to satisfy, may make the task easier by giving a more detailed measure of the error in the controller's output.

2.3.3 The controller

The controller was made up from three elements: a decoder, an Associative Search Element (ASE), and an Adaptive Critic Element (ACE). The decoder passed the information in the state vector of the inverted pendulum to the ASE, which held the control policy and generated the control action. The ACE judged the performance of the policy, allowing it to be improved. Each of these devices will be considered in turn.

The decoder

The decoder mapped the 4-dimensional state vector of the pendulum system into a 162 element vector, through a partition of the state space into 162 fixed regions, the so called "boxes" of Michie and Chambers (Michie and Chambers 1968). Each element of the decoder's output vector corresponded to a box within the state space. If a box was occupied by the current state of the system, its corresponding element in the output vector would be set to 1, otherwise it would be set to zero.

The disadvantage of this kind of method originates with the decoder. The decoder is required to map a continuous state, or sensor, space into a finite discrete space. One could ease the problem by employing the finest grain decoder that the available computing power can offer, but unfortunately this then affects the controller's ability to generalize to novel situations.

The ASE

Each element of the decoder's output vector was linked to the ASE. Each link had a weight. The output $y(t)$ of the ASE was defined as follows:

$$y(t) = f\left[\sum_{i=1}^{n} w_i(t)x_i(t) + \text{noise}(t)\right] \tag{2.11}$$

where t is a time index, n is the number of weights w_i equal to the number of boxes in the decoder, x_i is the i'th element of the decoded state vector, noise(t) is a Gaussian noise signal of small-variance and zero mean, used to explore the environment by randomizing the policy, and f is a threshold function like an ideal relay. The threshold function is given by:

$$f(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \tag{2.12}$$

The output of the ASE was scaled by a fixed magnitude and fed to the pendulum system as a control action. The ASE/ACE controller had only two actions to offer, which made it easier to learn to balance the inverted pendulum. When one action was seen to be poor, the sole alternative was to select the other. However, the restricted action set also made the controller's task harder because no null action was available; taking an action also disturbed the pendulum from an upright position.

The ACE

The Adaptive Critic Element (ACE) solved the temporal credit assignment problem in the inverted pendulum task. The problem shows itself in this context as follows. An obvious approach to reinforcement learning is to associate the most recent reinforcement with the most recent action. This means that the negative reinforcement from a collapse of the pendulum penalizes the last action. However, more often than not a poor action taken earlier is to blame for the pendulum's collapse — recent actions could even have been optimal for the circumstances. Without a solution to this problem, a controller will fail to learn a successful policy.

The ACE solved the temporal credit assignment problem by maintaining what can be thought of as a 'danger level' for each box in the state space. Whenever the system moved towards a less dangerous box, the most recent action would be rewarded; whenever it moved to a more dangerous box, the most recent action would be penalized (thus making an alternative action more likely next time). The 'danger level' map was an approximation to the J function in the dynamic programming equation. Actions were rewarded or penalized by respectively increasing or decreasing the relevant weights w_i in the ASE.

The danger map was created using the external (primary) reinforcement signal. Each entry in the map represented a prediction of the external reinforcement to be expected in the future. The control policy was not changed so long as the performance was expected, even if the performance was poor. Only unexpected performance would lead to a change in the ASE's policy.

The output of the ACE, the predicted performance $p(t)$, was given by:

$$p(t) = \sum_{i=1}^{n} v_i(t)x_i(t) \tag{2.13}$$

where x_i represents the same decoder outputs used in equation (2.11), and where v_i represents a set of link weights for the ACE, similar to those of the ASE.

2.3.4 Adapting the ASE and ACE

Each weight v_i of the ACE was updated as follows:

$$v_i(t+1) = v_i(t) + \beta\left[r(t) + \gamma p(t) - p(t-1)\right]\overline{x}_i(t) \tag{2.14}$$

where $r(t)$ is the primary reinforcement signal, γ and β are positive constants, and $\overline{x}_i(t)$ is a trace of the decoder's output. The trace offered a way of generalizing across boxes, and is defined as follows:

$$\overline{x}_i(t+1) = \lambda\overline{x}_i(t) + (1-\lambda)x_i(t) \tag{2.15}$$

where $0 \leq \lambda < 1$ determines the rate of trace decay. The trace worked by weighting boxes more heavily if they were occupied more recently.

The internal reinforcement signal used to update the ASE was defined by:

$$\hat{r}(t) = r(t) + \gamma p(t) - p(t-1). \tag{2.16}$$

The same term appears in brackets in the weight update equation for the ACE (equation (2.14)).

Normally, the external reinforcement $r(t)$ was zero, and the internal reinforcement $\hat{r}(t)$ was equal to $\gamma p(t) - p(t-1)$. When a collapse occurred, however, the external reinforcement was -1 and, because there was no box in the decoder that corresponded to a collapsed state, the prediction $p(t)$ was zero. Thus at collapse the internal reinforcement would be $-1 - p(t-1)$. This amounted to substituting the prediction $p(t)$ with the real reinforcement $r(t)$ at a collapse.

The internal reinforcement signal updated the ASE weights as follows:

$$w_i(t+1) = w_i(t) + \alpha\hat{r}(t)e_i(t) \tag{2.17}$$

where α is a constant and e_i is a trace, similar to $\overline{x}_i(t)$ above and defined as follows:

$$e_i(t+1) = \delta e_i(t) + (1-\delta)y(t)x_i(t) \tag{2.18}$$

where $0 \leq \delta < 1$. The ASE traces were reset to zero after a collapse.

The values for the constants used in the ASE/ACE equations are given in table 2.1.

ASE	
δ	0.9
α	1000.0
ACE	
λ	0.8
γ	0.95
β	0.5

Table 2.1: ASE/ACE constants

2.3.5 Results

The results of using the ASE/ACE controller with the simulated inverted pendulum problem are given in figure 2.3. The graph plots a trace of the duration of each trial against the number of trials. The trace is used to smooth the graphs. It is described fully in section 5.4.3.

Six out of the ten runs plotted converged to a satisfactory policy. The runs differed in the seed of the random number generator.

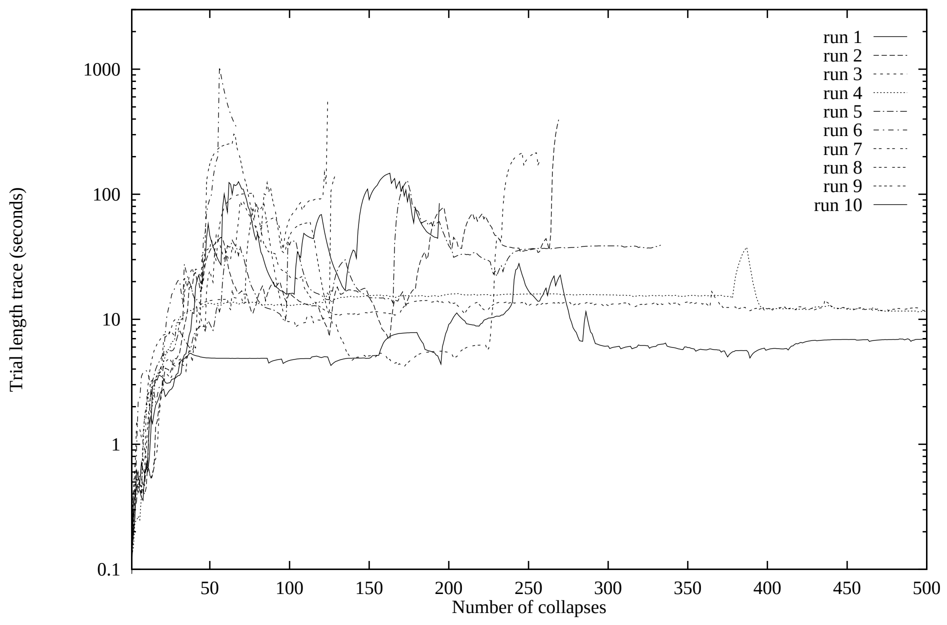


Figure 2.3: Results of using the ASE/ACE controller

2.4 Q learning

Q-learning, introduced by Watkins (Watkins 1989), is another implementation of dynamic programming. This time, rather than storing a value of J for each state, a value $Q(i,u)$ for each state i and action u is held. Control actions could be selected based on maximizing their Q value, but this is not necessary, allowing exploration in state and action space to update Q.

The Q values are updated as follows:

$$Q(i,u) = (1-\alpha)Q(i,u) + \alpha(c(i,u) + \lambda \min_{u \in U(j)} Q(j,u)) \tag{2.19}$$

δ	0.9
λ	0.8
γ	0.95
β	0.5

Table 2.2: Q-learning parameters

where α is a learning rate term and λ is the cost discounting factor.

2.4.1 Q learning applied to the inverted pendulum

The ASE/ACE controller has two complementary structures, the ASE to hold the control policy, and the ACE to judge that policy. A similar kind of controller can be made by holding the control policy implicitly. By storing a figure of merit for the available control actions at each time step, and choosing the action of greatest merit, one can simplify the controller and reduce the number of free parameters. The essence of the scheme is to have two structures like the ACE (for a controller with two actions). The number of parameters is reduced through a symmetry argument: the parameters for learning the merit of one action should be the same as any other action. This development is equivalent to Q-learning.

The Q-learning method may be applied to the inverted pendulum problem. As before, one makes use of a decoder to split the state space into partitions. Consider a controller with two actions of equal magnitude but opposite sign. The figures of merit for the actions are given by:

$$p_{left}(t) = \sum_{i=1}^{n} left_i(t) x_i(t) \tag{2.20}$$

$$p_{right}(t) = \sum_{i=1}^{n} right_i(t) x_i(t) \tag{2.21}$$

The controller's action may be chosen to be the action with the greatest merit.

The action values may be updated in a way similar to the ACE in the ASE/ACE scheme:

$$left_i(t+1) = left_i(t) + \beta\left[r(t) + \gamma p(t) - p(t-1)\right] \bar{l}_i(t) \tag{2.22}$$

$$right_i(t+1) = right_i(t) + \beta\left[r(t) + \gamma p(t) - p(t-1)\right] \bar{r}_i(t) \tag{2.23}$$

where $\bar{l}_i(t)$ and $\bar{r}_i(t)$ are traces for box occupancy when taking the left and right actions respectively.

2.4.2 Results

The results of using Q-learning on the inverted pendulum problem are given in figure 2.4. The Barto parameters given in appendix B were used for the simulation. Six out of the ten runs converged to a satisfactory controller. The only difference between the runs was the seed of the random number generator.

The parameters of the Q-learning controller for the results in figure 2.4 are given in table 2.2. Note that there is no need for the decay parameter α.

2.5 How good are the reinforcement learning systems?

Both the ASE/ACE actor/critic of Barto et al., and Q-learning, are optimizers searching for a mapping from state to action space that regulates the pendulum. To measure the worth of the

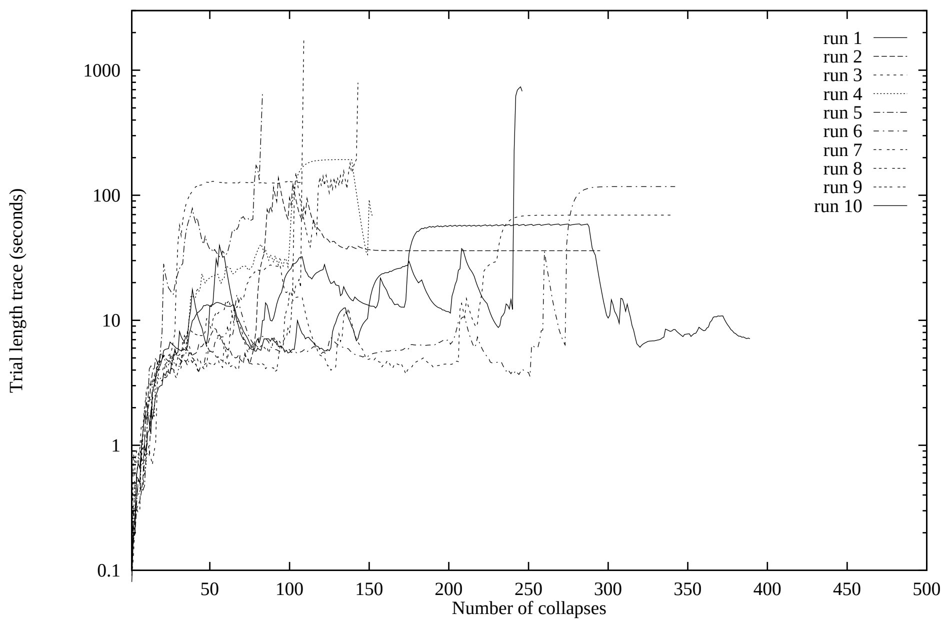


Figure 2.4: Q-learning results.

optimization schemes, the space in which they are searching may be sampled to estimate the density of suitable solutions.

Figure 2.5 shows a histogram of the performance of 5000 policies chosen uniformly at random in the discrete policy space. None of the samples gave satisfactory control (balancing longer than 100 seconds). The controllers are better than a random search, and the sampling suggests good control policies are more scarce than 1 in 5000.

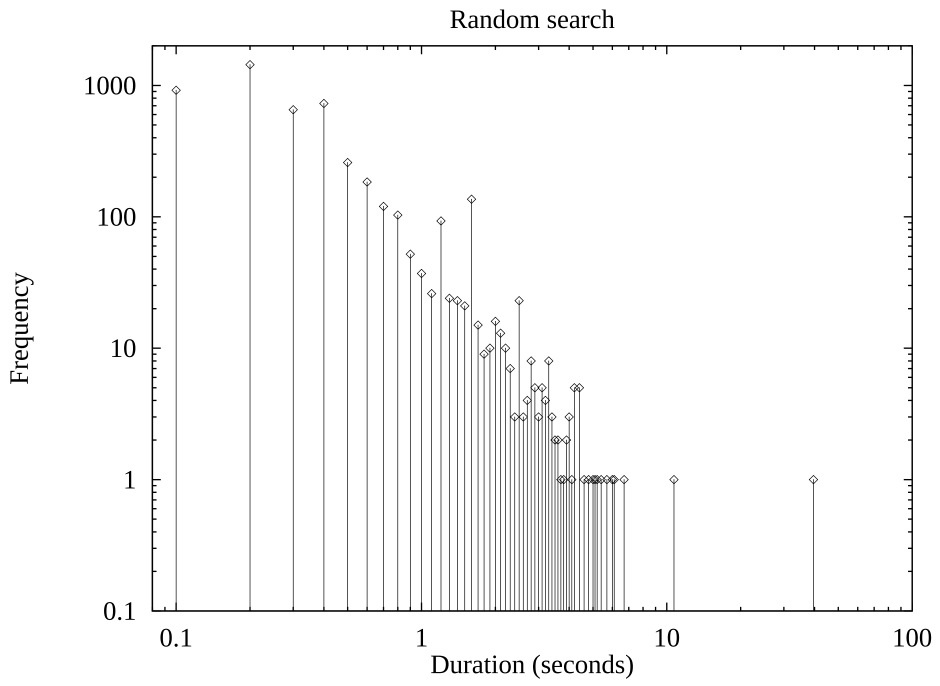


Figure 2.5: Performance of policies sampled at random from policy space.

Chapter 3

Neural networks

Before the treatment of adaptive controllers is extended into the continuous domain, a review of neural networks is given. This chapter introduces neural networks, with a view to their application for learning control discussed in the next chapter.

3.1 Introduction

A neural network maps input vectors to output vectors. They are so called because of similarities between their structure and that of the human brain. They are characterized by a set of simple processing elements (the 'neurons') connected by weighted links. Among the applications for neural networks are classification, prediction of the next state of a discretized system based on the current state and the control action, and production of control actions as a function of the current state of a system.

Neural networks are parametrized by a set of weights[1]. The architecture of the network is defined by the number of nodes and the way they are linked. The input/output behaviour of a network is defined by its architecture, the functions at its nodes and the weights on the links.

A neural network comprises a set of weighted links and a set of nodes (figure 3.1). The nodes are grouped into layers. Layers that are not the input layer or the output layer are known as hidden layers. The maximum number of hidden layers that will be considered here is one. A feed-forward network is a network in which all the connections are in the direction of input to output.

A network with a layered connectivity contains links from the input layer to the hidden layer, and from the hidden layer to the output layer. A network with full connectivity has a layered connectivity with extra links directly from the input layer to the output layer.

A network node forms a weighted sum x of its inputs. The activation y of the node is a function of x. The function may be linear or non-linear. For nodes in the hidden layer, two candidate functions are the tanh function and the log-sigmoid function:

$$y = \frac{1}{1 + \exp(-x)}. \tag{3.1}$$

Nodes in the output layer often have a linear function ($y = x$) so that the outputs are not limited in size.

Radial basis function (RBF) networks, Hopfield networks and Boltzmann machines are not considered here. Rumelhart et al. give a fuller tutorial on neural networks (Rumelhart and McClelland 1986). A generalization of feed-forward neural networks is the recurrent network (Robinson and Fallside 1987, Williams and Zipser 1989). Such networks have extra connections from their outputs back to their inputs. These networks are also beyond the scope of this chapter.

[1]Network nodes often have a bias to give non-zero values when there is zero input. A bias term for each node in the network is just like a weighted link from a hypothetical node with unit activation, and can be adapted like any other weight. Using this interpretation means we need only talk of the weights of a network, rather than having to say 'the weights and biases'.

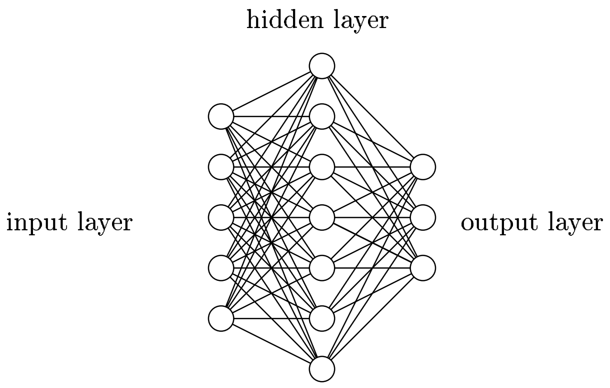


Figure 3.1: A feed-forward neural network with a layered connectivity for a task mapping a 5-dimensional input space to a 3-dimensional output space.

Neural networks have been used for learning control by a number of authors. They act as function approximators. Because they are continuous and not discrete functions, they can be used to avoid quantizing continuous state and action spaces. They offer a promising set of qualities to the field of control:

- They perform non-linear mappings
- Input and output can be vector-valued
- The mappings are differentiable
- They can be trained to reproduce examples
- They can generalize (interpolate and extrapolate) from examples
- They operate on real-valued data
- Their structure is amenable to fast parallel processing

Because they can form non-linear mappings they can be used to model non-linear plant. They can fuse sensory information, allowing multiple channels of different sensory information to be used collaboratively in determining an appropriate control action. Because the models they form are differentiable, gradient information can be used for efficient optimization of their parameters. Gradient information can also be used to measure the dependence between input and output. Learning control requires a controller with the potential to emulate a wide range of control laws, and neural networks are thus suitable candidates for this rôle.

Neural networks have stimulated (popular) philosophical discussions of minds and brains. These discussions can detract from their useful analytical rôle as parametrized non-linear functions and models. They are used in this work in no other rôle.

In cases where some knowledge about the structure of the underlying process exists, it is sensible to use a model that embodies that knowledge. If, however, little is known about the problem, a neural network is a useful free-form model.

3.2 Non-linear probabilistic models

Neural networks have successfully been cast as general non-linear modelling tools for statistical analysis. The Bayesian probabilistic framework, introduced to neural networks by MacKay (MacKay 1991), allows a number of additional features to be teased out of a neural network model. These include error bars on the weights of the network, error bars on the network outputs, and the ability to do model comparison between models of different orders. This last feature can be used to choose between neural networks with different numbers of hidden units, or between neural networks and other kinds of model. Finally, the traditional 'weight decay' terms used to train neural networks have been cast as regularization parameters, representing a prior distribution over the parameters of the network.

The probabilistic framework allows neural networks to be seen in the context of statistical inference. One may describe a neural network with a set of weights as a non-linear model with a set of parameters. Error bars may be found for the model.

Figure 3.2 shows the error bars of a neural network interpolating between the points in a data set. The data set is a sampled sinusoid, with part of the sequence missing. The graph shows that the error bars widen in the region of the missing data, and diverge where the network is extrapolating beyond the data. The error bars are plus and minus the roots of the variances $v(x)$ given below.

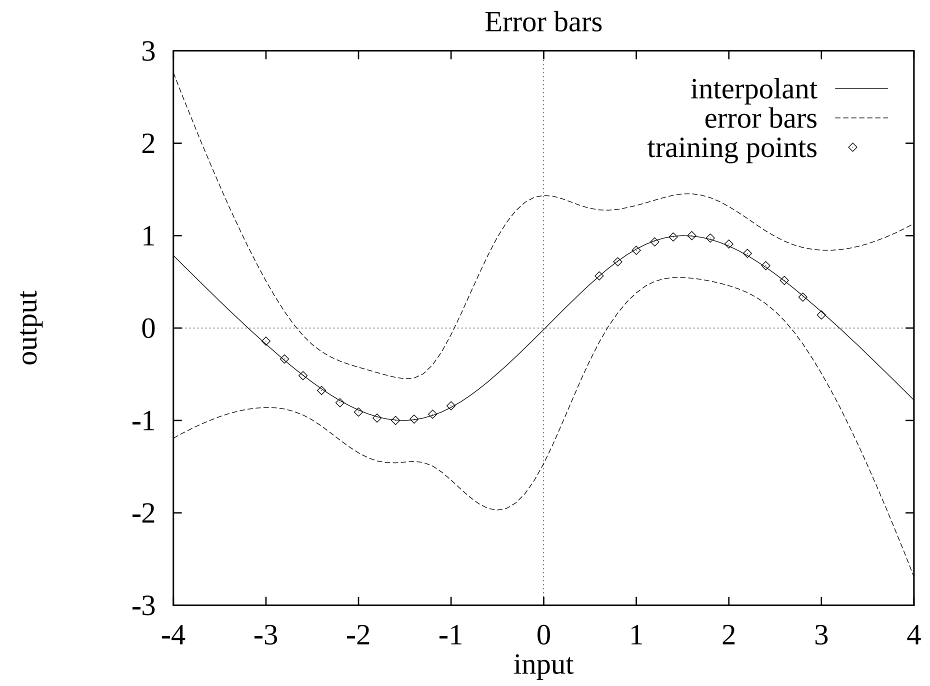


Figure 3.2: Error bars on a neural network interpolation. The error bars shown are +/- one standard deviation from the interpolant. All the regularization parameters (decay terms) α were set to 0.01 for this example.

The graph was produced by applying some of the techniques prescribed by MacKay (MacKay 1991). The network was trained to find a minimum of the misfit objective function (described in section 3.3.2). The Hessian $\mathbf{H}$ of the objective function with respect to the weights was calculated using the method of finite differences. The error bars were then calculated at sample points uniformly spaced along the abscissa from:

$$v(x) = \frac{\partial o}{\partial \mathbf{w}}\Big|_x \mathbf{H}^{-1} \frac{\partial o}{\partial \mathbf{w}}\Big|_x^T \tag{3.2}$$

where $v(x)$ is the variance of the output at input x, $\frac{\partial o}{\partial \mathbf{w}}\big|_x$ is a row vector of derivatives of the output with respect to the parameters $\mathbf{w}$, and $\mathbf{H}^{-1}$ is the inverse Hessian of the misfit objective function with respect to the parameters $\mathbf{w}$ at the misfit minimum. For multiple outputs, the term $\frac{\partial o}{\partial \mathbf{w}}\big|_x$ is a matrix, the Jacobian of the outputs with respect to the parameters.

MacKay has successfully applied his Bayesian framework to real-world data (MacKay 1994), making use of a technique he terms 'Automatic Relevance Determination', which allows his networks to select relevant inputs and reject irrelevant ones. The idea is based on an extension of the mechanism to set the weight decay parameters, grouping parameters on weights from common inputs instead of on weights from common layers.

3.2.1 Generalization

Part of the utility of neural networks comes from their ability to extrapolate from and interpolate between the training data used to set the weights. Setting the weights of a network so that this generalization is sensible may be done in two ways.

One way is to use cross-validation. These methods use part of the data as a training set, and part as a validation set. One cross-validation method is called 'early stopping'. The network is trained for a while on the training set, and then tested on the validation set. As training progresses, the error on the validation set initially falls. However, the validation error will eventually start to rise, and the network is being over-trained. Training is stopped when a minimum of the validation error is reached, even though proceeding might give lower training errors. It is not desirable for a network to be trained to the point where it can faithfully reproduce the training set, because it would generalize poorly.

Another way to ensure good generalization is to take a Bayesian procedure when training the weights (MacKay 1991, Thodberg 1993). A prior probability distribution for the weights and a noise model for the network mapping are used to find a posterior distribution for the weights in the light of the data. MacKay uses the misfit function, described in section 3.3.2, in connection with this framework.

Generalization ability is not a feature of an optimization strategy. Both these methods of generalization can benefit from fast optimization strategies, but gradient descent, for example, does not by itself necessarily lead to good generalization.

3.3 Objective functions

Various optimization techniques, generally based on gradient descent methods, can be used to find weights that give a network a suitable input/output mapping, for either classification or regression. This means finding weights that optimize an objective function.

The concept of the objective function is central to optimization. To aid visualization a simple sum-squared error objective function is given in figure 3.3.

Figure 3.3 can help to visualize the motivation behind optimization schemes, but should not be taken as being representative of all objective functions in weight space. It is easy to make very different surfaces for even this simple system.

The task for a neural network determines the objective function for a neural network optimization problem. Some objective functions are given below.

3.3.1 Sum-squared error

For regression problems (also known as interpolation problems), a model is required that fits some data in a reasonable way. The way this is done is to define a measure of error between the model and the data. This error becomes the objective function for an optimization method to train regression networks.

A training set comprises a set of patterns p, each pattern being a pair of input and target vectors. For each pattern p, the difference between the output produced by the network $\mathbf{o}_p$ and the target $\mathbf{t}_p$ is an error vector $\mathbf{e}_p$:

$$\mathbf{e}_p = \mathbf{o}_p - \mathbf{t}_p. \tag{3.3}$$

A scalar measure of the difference between the output and the target is the sum-squared error:

$$e_p = \frac{1}{2}\sum_i (\mathbf{o}_p(i) - \mathbf{t}_p(i))^2 = \frac{1}{2}\mathbf{e}_p^T\mathbf{e}_p \tag{3.4}$$

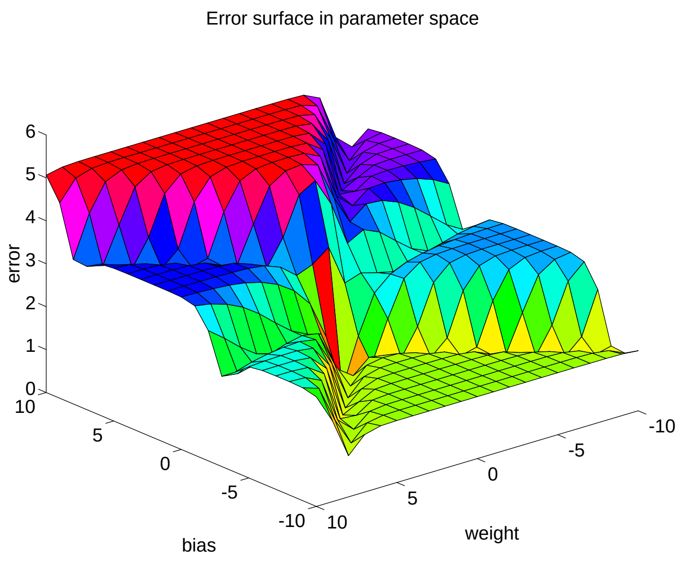


Figure 3.3: An error surface for the simplest network possible. The network has one node, with one weight and a bias. This is the only system for which an error surface can be plotted. The node has a hyperbolic tangent non-linearity. Three training inputs were used: 0.3, 0.8 and 1.2. The three targets were −0.23, −0.4 and −0.3 respectively.

where $\mathbf{o}_p(i)$ is element i of the output vector for pattern p and $\mathbf{t}_p(i)$ is the target for that value, the sum being taken over all the elements in the output vector. The error over the whole training set is the sum of the errors e_p over the patterns p:

$$e = \sum_p e_p \tag{3.5}$$

The error over the training set is a function of the weights of the network. The error surface in weight space is a surface whose height at a point is the error e over the training set when the network weights are set to the coordinates of the point in weight space.

3.3.2 Misfit

The Bayesian framework for neural networks developed by MacKay makes use of a modified objective function, termed the 'misfit' m, made up from the data error (equation 3.5) and a 'weight error', related to the prior distribution of the weights:

$$m = \sum_i \alpha_i w_i^2 + \sum_j \beta_j (o_j - t_j)^2 \tag{3.6}$$

The factors α and β above are regularization parameters. Traditionally, use of such parameters has been termed 'weight decay'. Members of the statistics community use the term 'ridge regression'. Both MacKay and Thodberg give prescriptions for fitting these hyper-parameters to the data (MacKay 1991, Thodberg 1993). Regularization reduces over-fitting of the data. It also reduces sensitivity to over-parametrization of the model, which can happen when there are too many hidden nodes in the network.

3.3.3 Log probability with softmax

Apart from the sum-squared error objective function used for regression, another popular objective function is a probability function based on the softmax function introduced by Bridle (Bridle 1990).

The softmax function is popular because it allows a neural network to act as a probabilistic classifier. Let the i'th output of a neural network with a linear output layer be denoted by x_i. The softmax function produces an output o_i, given by:

$$o_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{3.7}$$

Thus the softmax function has outputs that sum to one and are in the range $[0,1]$.

The output of the function is interpreted as a probability for input data being in one of a number of classes. A data set can be given a probability by taking the product of probabilities for the class of each input datum. One reason for adopting the softmax function as the output of a classifier is the simplicity it gives in the log domain, and so the log probability of the data set is normally considered. Differentiating the log probability with respect to the outputs yields a simple result. The back-propagation algorithm (section 3.5) can then be applied to this derivative to find the derivative of the log probability with respect to the parameters of the neural network model.

The probability $p(t(\mathbf{d}) = c)$ of an input $\mathbf{d}$ being in class c is given by:

$$\log p(t(\mathbf{d}) = c) = x_c - \log \sum_j \exp(x_j) \tag{3.8}$$

where $t(\mathbf{d})$ is the class of the datum $\mathbf{d}$ and x_c is the c'th element of the linear output of the net $\mathbf{x}$ input with $\mathbf{d}$. x_c is the output corresponding to class c. The right hand term of the equation is the normalizing factor. For a series of inputs $\mathbf{d}_i$, the log probability is the sum $\sum_i \log p(t(\mathbf{d}_i))$.

The derivative of the log probability of a datum $\mathbf{d}$ with respect to the linear output $\mathbf{x}$ is given by

$$\frac{\partial \log p}{\partial x_i} = \delta_{ic} - o_i \tag{3.9}$$

where x_i is the i'th element of the linear output $\mathbf{x}$; δ_{ic} is the target for the i'th output of the classifier, one if the i'th output is for the correct class c and zero otherwise; and o_i is the i'th output of the softmax function over the linear outputs.

The purpose of optimizing the parameters of such a classification network is to maximize the probability of the model and to find the most probable parameters.

3.4 On-line and off-line optimization

Off-line optimization methods update the weights after a complete pass of the training data. An epoch refers to a complete pass through the training data. Thus, off-line methods update the weights after each epoch. This is also known as batch learning.

On-line optimization updates the weights after a single training pattern (a pair of input and target vectors). The regression error e_p for the single training pattern p can be used. On-line methods therefore update the weights several times during an epoch.

An off-line optimization procedure can often be transformed into an on-line procedure by using an element of the objective function for the current pattern instead of a full evaluation over the whole training set. Weights may also be updated after any number of input/target presentations, so on-line and off-line learning are extremes. Introducing these changes, however, means that the optimization procedure is using several different objective functions and thus the learning dynamics is more complicated.

Schiffmann et al. report that on-line gradient descent is superior to more advanced second-order techniques running in batch mode when training with large data sets (Schiffmann, Joost and Werner 1992). The problem of training networks on large, redundant data sets (compared with the size of the network) is an issue not covered here, and is linked with the problem of generalization. The results to follow are valid for training sets of similar size to the number of parameters in the network. For large data sets with redundancy, one could prune the data set or consider an on-line learning scheme.

On-line methods are crucial for control problems where there can be a large amount of data over the duration of the control period. On-line methods should approximate to training with all the data. One approach to on-line, or incremental, learning proposed by Kadirkamanathan involves a trade-off between fitting the most recent data point, and minimizing the distance in function space moved between the previous regressor and the newest (Kadirkamanathan 1991). On-line learning is a problem for further research.

3.5 The back-propagation algorithm

A resurgence of interest in neural networks occurred when the back-propagation algorithm (commonly referred to as backprop) was introduced (Rumelhart, Hinton and Williams 1986). This algorithm is an application of the chain rule to neural networks. The algorithm is able to compute the partial derivative, $\frac{\partial f}{\partial w}$, of a suitable function f with respect to each weight w in the network. The log probability of the data in a classification task, using the softmax function (Bridle 1990), is a suitable function, as is the sum-squared error and misfit (section 3.3.2) function.

Define $\frac{\partial f}{\partial \mathbf{w}}$ to be a column vector of n elements, where n is the number of weights in the network:

$$\frac{\partial f}{\partial \mathbf{w}} = \nabla f = \left[\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \cdots, \frac{\partial f}{\partial w_n}\right]^T. \tag{3.10}$$

The back-propagation algorithm is derived as follows. Let node i in the network have an input from the other nodes given by x_i and an input from the external world given by I_i. The output, or activation, of the node, y_i, is denoted by:

$$y_i = \sigma_i(x_i) + I_i \tag{3.11}$$

where σ_i is the node function (e.g. linear, log-sigmoid or hyperbolic tangent). Thus, for nodes in the input layer of the network, $x_i = 0$, and I_i is an element of the input pattern to the network. For nodes not in the input layer, $I_i = 0$ and x_i is given by:

$$x_i = \sum_j w_{ji} y_j \tag{3.12}$$

where w_{ji} is the weight[2] of the link between node j and node i.

Equations 3.11 and 3.12 define the forward pass of the back-propagation algorithm for a feed-forward network (where the connections are such that the values of x_i and y_i may be computed in an order that does not lead to recursion.)

We are looking for an expression for $\frac{\partial f}{\partial w}$ for each weight w. We proceed by tracing the effect that changing each weight has on the following chain of dependence: the effect of the weights on the inputs to the nodes; the effect of these inputs on the output of the nodes; the effect of these outputs on the function f. Firstly, the effect that changing the weight w_{ij} has on the input to node j gives, using equation 3.12:

$$\frac{\partial f}{\partial w_{ij}} = \frac{\partial f}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = \frac{\partial f}{\partial x_j} y_i. \tag{3.13}$$

An expression for $\frac{\partial f}{\partial x_j}$ in terms of the node activations y_i comes from equation 3.11:

$$\frac{\partial f}{\partial x_j} = \frac{\partial f}{\partial y_j}\frac{\partial y_j}{\partial x_j} = \frac{\partial f}{\partial y_j}\sigma_j'(x_j). \tag{3.14}$$

Finally $\frac{\partial f}{\partial y_i}$ is given by:

[2]Sometimes it will be convenient to index the network weights twice, to indicate the source and destination nodes of the weighted link. At other times it will be convenient to number the weights with a single integer, treating the weights as a vector. The convention will be clear from the context.

$$\frac{\partial f}{\partial y_i} = \frac{df}{dy_i} + \sum_j w_{ij} \frac{\partial f}{\partial x_j} \tag{3.15}$$

where $\frac{df}{dy_i}$ is the direct derivative of the function with respect to the activation y_i. $\frac{df}{dy_i}$ is zero for any non-output node. For an output node and the sum-squared error criterion, the right-hand term is zero, and $\frac{df}{dy_i}$ is $y_i - t_i$, where t_i is the target value for the output node i.

Equations 3.13, 3.14 and 3.15 define the backward pass of the backprop algorithm. The algorithm is so called because the direct derivatives of the function f at the output nodes (equation 3.15) are evaluated first, and the results propagated backwards through the network towards the inputs.

3.6 First order optimization methods

The simplest approach to finding suitable weights is to follow the gradient of the objective function in weight space. Let the set of weights at iteration i be denoted by $\mathbf{w}(i)$. Then the update rule for gradient descent is given by:

$$\mathbf{w}(i+1) = \mathbf{w}(i) - \eta \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i \tag{3.16}$$

where $\left.\frac{\partial f}{\partial \mathbf{w}}\right|_i$ is the vector of gradients of the function f with respect to each weight w evaluated at iteration i, and η is a learning rate parameter. The value of the learning rate parameter, or step size, is crucial for the success of the algorithm.

3.6.1 Setting the step size

A small step size leads to slow learning and the possibility of getting trapped in local minima of the objective function. A large step size might overshoot the minimum. The weights may then be set to a point in weight space on a high plateau of the objective function. A plateau in the objective function will exist where the nodes have saturated activations. Here, the gradient is small. The small gradient then means the large step size is ineffective and the network learns slowly again, but this time from a high value of the objective function. Thus both small and large step sizes are undesirable.

The best step size depends on the objective function, itself a function of the architecture of the network (the number of nodes and the connections between them) and the training data. The following heuristic to set η includes a dependence on the network architecture and the training data:

$$\eta = \frac{0.01}{\left| \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i=1} \right|} \tag{3.17}$$

where $\left| \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i=1} \right|$ is the Euclidean norm of the gradient at the first iteration, so

$$\left|\frac{\partial f}{\partial \mathbf{w}}\right|^2 = \sum_w \left(\frac{\partial f}{\partial w}\right)^2 . \tag{3.18}$$

Equation 3.17 has been used in the optimization schemes below to set a step size when one is not specified by the optimization strategy. In practice it appears useful for a variety of different architectures and training tasks.

3.6.2 Line searches

A line search is a search for an optimum of the objective function along a line. It avoids the problem of setting a fixed step size. The search may be exact, or inexact, depending on the method and the stopping criterion.

An inexact line minimization samples the objective function at a number of points along the search direction. The search returns the point with the lowest value from among the samples. Samples at points nearer and further than the length of the default step size given in equation 3.17 might be chosen. The dynamic range of the steps can be modified after each iteration, based on the position of the minimum sample. Such a procedure avoids sticking to a fixed step size, but is inefficient in its sampling strategy. The same number of evaluations might be done at more appropriate places, based on the results of previous evaluations.

A more efficient line search first finds an interval, along the search direction, in which a minimum exists, and then finds a minimum within the interval. The interval is characterized by three points. The objective function at the central point is lower than at the outer two. The interval therefore brackets a minimum. The interval comprises two sections. A point within the larger section is then sampled, giving four samples. From the four samples, three can be found which form another, narrower, interval that brackets a minimum. The procedure may be continued until the interval is small. The linmin procedure implements this method (Press, Flannery, Teukolsky and Vetterling 1992).

Sometimes there is scope for a less accurate line search for reduced computational effort. Polynomial approximations can be used to estimate the position of an optimum along a line, initially quadratic and then cubic after a further data point has been acquired, until a point of sufficient quality has been reached. This is a search for a lower point rather than a minimization. The lnsrch procedure implements this kind of search (Press et al. 1992).

A line search inevitably requires several evaluations of the objective function to find a minimum. Gradient information can also be used to improve the search, at further computational cost. The value of a good line search has to be judged against its computational cost. There is a playoff between the effort that should be made to minimize during the line search, and the effort that should be made searching in other directions.

3.6.3 Adding momentum

Adding a momentum term to gradient descent alters the search direction by adding some of the previous search directions to the current gradient. In practice, adding momentum speeds convergence.

Firstly, define the weight change $\boldsymbol{\delta}\mathbf{w}(i)$ for iteration i:

$$\mathbf{w}(i+1) = \mathbf{w}(i) + \boldsymbol{\delta}\mathbf{w}(i). \tag{3.19}$$

Momentum is used as follows:

$$\boldsymbol{\delta}\mathbf{w}(i) = -\eta \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i + \alpha \boldsymbol{\delta}\mathbf{w}(i-1) \tag{3.20}$$

where $\alpha = 0.9$ is the momentum term and $\boldsymbol{\delta}\mathbf{w}(i-1)$ is the previous weight change. The value of α is usually set to 0.9 independent of the problem and the architecture of the network.

3.7 Second order optimization methods

Improved training methods for neural networks have been created since the introduction of the back-propagation algorithm. These faster methods might lead to neural network learning controllers being implemented in real time. A review of such optimization methods follows.

Optimization procedures are commonly based on making an assumption that the local error surface has a simple form. First-order gradient descent makes a linear approximation, whereas second-order methods use extra information to make a local quadratic model. The success of second order methods will be determined firstly by the validity of the quadratic model, and secondly, when the model breaks down, what strategies are used to resurrect the procedure.

Second order optimization methods make use of second derivatives of the objective function in weight space. The key to second order methods is the Hessian matrix $\mathbf{H}$.

The Hessian matrix is a matrix of second derivatives of the objective function f with respect to the parameters $\mathbf{w}$:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial w_1^2} & \frac{\partial^2 f}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 f}{\partial w_1 \partial w_n} \\ \frac{\partial^2 f}{\partial w_2 \partial w_1} & \frac{\partial^2 f}{\partial w_2^2} & \cdots & \frac{\partial^2 f}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial w_n \partial w_1} & \frac{\partial^2 f}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 f}{\partial w_n^2} \end{bmatrix}. \tag{3.21}$$

The Hessian describes the curvature of the objective function in weight space. It contains information about how the gradient changes in different directions in weight space, and answers the question "How does the gradient change if I move off from this point in that direction?". Let the Hessian $\mathbf{H}$ be evaluated at a point $\mathbf{w}$ in weight space. Let us then consider a direction $\mathbf{v}$ from this point in weight space. The product $\mathbf{Hv}$ is the rate of change of the gradient along the direction $\mathbf{v}$ from the point $\mathbf{w}$.

The Hessian is useful for calculating both search directions and step sizes for optimization schemes. It is also large and difficult to compute. Fortunately, there are several techniques that can implicitly calculate the Hessian, avoiding the need to calculate it or store it explicitly.

3.7.1 Conjugate gradient descent

The idea behind conjugate[3] gradients is to choose search directions that complement each other. This is meant to avoid the possibility of 'undoing' the minimization of previous iterations, by choosing appropriate search directions.

Assume the objective function is a quadratic in weight space:

$$f(\mathbf{w}) = \mathbf{c}^T\mathbf{w} + \frac{1}{2}\mathbf{w}^T\mathbf{H}\mathbf{w} \tag{3.22}$$

where $\mathbf{c}$ is a constant and $\mathbf{H}$ is a Hessian matrix, constant throughout the weight space. The search direction at iteration i is denoted by $\mathbf{v}(i)$. The search directions are constructed to be conjugate in the following sense: for all $i \neq j$,

$$\mathbf{v}(i)^T\mathbf{H}\mathbf{v}(j) = 0. \tag{3.23}$$

It is not necessary to compute the Hessian to find a set of conjugate directions. Conjugate directions may be generated as follows. Set the first direction, $\mathbf{v}(1)$, to minus the gradient $-\left.\frac{\partial f}{\partial \mathbf{w}}\right|_1$. Set subsequent directions using:

$$\mathbf{v}(i) = -\left.\frac{\partial f}{\partial \mathbf{w}}\right|_i + \beta(i)\mathbf{v}(i-1) \tag{3.24}$$

where $\beta(i)$ is a scalar value acting like an adaptive momentum term. β may be defined in a number of ways, each of which produces conjugate directions. The differences between the definitions show themselves when the objective function is not quadratic. Two definitions for β are given below. The Polak-Ribiere rule (Hertz, Palmer and Krogh 1991) for β is given by:

$$\beta(i) = \frac{\left(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_i - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i-1}\right)^T \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i}{\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i-1}^T \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i-1}}. \tag{3.25}$$

The Hestenes-Stiefel rule (Møller 1993) for β is given by:

[3] One definition for 'conjugate' from Webster: 'having features in common but opposite or inverse in some particular'

$$\beta(i) = \frac{\left(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i-1} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i}\right)^T \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i}}{\mathbf{v}(i)^T \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i-1}} \tag{3.26}$$

and is reported to be more robust to non-quadratic surfaces (Møller 1993). In both equations, $\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i}$ is the column vector of partial derivatives of the objective function f by each weight w in the network evaluated at iteration i.

Conjugate gradient descent will reach the minimum of a quadratic surface in, at most, as many steps as there are dimensions of the weight space. For non-quadratic objective functions, the minimum may not have been reached after this number of steps. Instead, the algorithm is restarted by setting β to zero for a step, and the procedure continued as before.

Conjugate gradients give only a search direction, not a step size. Furthermore, the power of conjugate gradients is only apparent if the objective function is minimized along the current search direction. This is usually done with a line search, the difficulties of which have been outlined above.

Beale has given an accessible introduction to conjugate gradients (Beale 1971).

Avoiding the line search

If the objective function is quadratic, or has a slowly varying Hessian, a good approximation to the optimal step size for a search direction can be found.

If the objective function is minimized along a search direction, the gradient of the function at the minimum is perpendicular to the search direction. Hence:

$$\mathbf{v}^T \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+r\mathbf{v}} = 0. \tag{3.27}$$

where $\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+r\mathbf{v}}$ is a column vector of partial derivatives of f with respect to each weight evaluated at $\mathbf{w}+r\mathbf{v}$, $\mathbf{v}$ is the search direction and r is the step size along the search direction to the minimum.

Assuming a quadratic surface, equation 3.22 can be used to find an expression for the gradient:

$$\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}} = \mathbf{c} + \mathbf{H}\mathbf{w} \tag{3.28}$$

so the gradient at $\mathbf{w} + r\mathbf{v}$ is given by:

$$\begin{aligned}
\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+r\mathbf{v}} &= \mathbf{c} + \mathbf{H}(\mathbf{w} + r\mathbf{v}) && (3.29)\\
&= \mathbf{c} + \mathbf{H}\mathbf{w} + r\mathbf{H}\mathbf{v} && (3.30)\\
&= \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}} + r\mathbf{H}\mathbf{v}. && (3.31)
\end{aligned}$$

Substituting equation 3.31 into equation 3.27:

$$\begin{aligned}
\mathbf{v}^T \left(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}} + r\mathbf{H}\mathbf{v} \right) &= 0 && (3.32)\\
\mathbf{v}^T \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}} + r\mathbf{v}^T\mathbf{H}\mathbf{v} &= 0. && (3.33)
\end{aligned}$$

Rearranging equation 3.33 gives an expression for r, the distance to the minimum along the current search direction $\mathbf{v}$:

$$r = -\frac{\mathbf{v}^T \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}}{\mathbf{v}^T\mathbf{H}\mathbf{v}}. \tag{3.34}$$

Equation 3.34 has been avoided in the past due to the effort of finding the Hessian $\mathbf{H}$. The Hessian $\mathbf{H}$ may be approximated by finite differences, or the product $\mathbf{H}\mathbf{v}$ may be approximated by taking a difference of gradients. However, the equation becomes useful again in the light of the RBackprop algorithm, given in appendix A. RBackprop gives the product $\mathbf{Hv}$ computationally cheaply, and exactly, although this is not necessarily worthwhile for non-quadratic objective functions.

3.7.2 Scaled conjugate gradient

Møller has introduced a variation on conjugate gradient descent that takes some account of the non-quadratic nature of the objective function in weight space (Møller 1993).

The quadratic optimal step size given in equation 3.34 gives the distance to the turning point of the objective function along the search direction under the quadratic assumption. This turning point may be a maximum or a minimum. The Hessian can be used to find whether the turning point is a maximum or a minimum.

The product $\mathbf{Hv}$ of the Hessian $\mathbf{H}$ and a direction $\mathbf{v}$ is the rate of change of gradient in the direction $\mathbf{v}$. The expression $\mathbf{v}^T\mathbf{Hv}$ is negative if the gradient is increasing along the direction $\mathbf{v}$, and positive if the gradient is decreasing along the direction $\mathbf{v}$. Within the quadratic assumption, the Hessian is constant, and the sign of the change of gradient along the direction is constant. If the gradient is increasing, the graph of the objective function along $\mathbf{v}$ is a cup, and there is a minimum, otherwise there is a maximum.

To monitor the sign of the product $\mathbf{v}^T\mathbf{Hv}$, and therefore the type of the turning point, define δ by

$$\delta = \mathbf{v}^T\mathbf{Hv}. \tag{3.35}$$

If $\delta \leq 0$ for non-zero $\mathbf{v}$, there is a minimum along the direction $\mathbf{v}$. But for non-quadratic objective functions, it may be that $\delta \leq 0$ and yet there is still a minimum to find, since $\mathbf{H}$ changes along $\mathbf{v}$. Møller's scaled conjugate gradient method takes account of this.

Møller introduces two new variables, λ and $\bar{\lambda}$, to define an altered value of δ, $\bar{\delta}$. These variables are charged with ensuring that $\bar{\delta} > 0$. This does not affect the objective function, and the Hessian with the quadratic approximation will still suggest there is a maximum along the search direction. However, his method produces a step size that shows good results in practice.

$\bar{\delta}$ is defined as follows:

$$\bar{\delta} = \delta + (\bar{\lambda} - \lambda)\mathbf{v}^T\mathbf{v}. \tag{3.36}$$

The requirement for $\bar{\delta} > 0$ gives a condition for $\bar{\lambda}$:

$$\bar{\lambda} > \lambda - \frac{\delta}{\mathbf{v}^T\mathbf{v}}. \tag{3.37}$$

Møller then sets $\bar{\lambda} = 2\left(\lambda - \frac{\delta}{\mathbf{v}^T\mathbf{v}}\right)$ to satisfy equation 3.37 and so ensures $\bar{\delta} > 0$. This allows for a substitution in equation 3.36 to give:

$$\bar{\delta} = -\delta + \lambda\mathbf{v}^T\mathbf{v}. \tag{3.38}$$

Subsequently, $\bar{\delta}$ is substituted for expressions that would otherwise involve δ. Thus the step size r is found by substituting $\bar{\delta}$ for δ in equation 3.34:

$$r = -\frac{\mathbf{v}^T\frac{\partial f}{\partial \mathbf{w}}}{\bar{\delta}} = \frac{\mathbf{v}^T\frac{\partial f}{\partial \mathbf{w}}}{\delta - \lambda\mathbf{v}^T\mathbf{v}}. \tag{3.39}$$

It is now necessary to specify how to update the scale λ. This is based on a measure of fit between the quadratic approximation and the real surface. The fit is measured by the variable Δ, and is a ratio between the actual change in the objective function f produced by stepping along the search direction $\mathbf{v}$ by an amount r, and the predicted change in the function based on the quadratic approximation:

$$\Delta = \frac{f|_{\mathbf{w}+r\mathbf{v}} - f|_{\mathbf{w}}}{e_q|_{\mathbf{w}+r\mathbf{v}} - f|_{\mathbf{w}}} \tag{3.40}$$

where $f_q|_{\mathbf{w}+r\mathbf{v}}$ is the quadratic approximation of the function f at $\mathbf{w}+r\mathbf{v}$, given by the first three terms of the Taylor expansion:

$$f_q|_{\mathbf{w}+r\mathbf{v}} = f|_{\mathbf{w}} + r \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}^{T} \mathbf{v} + \frac{1}{2} r^2 \bar{\delta} \tag{3.41}$$

where a term that would naturally be $\mathbf{v}^T \mathbf{H}|_{\mathbf{w}} \mathbf{v}$ has been replaced by the 'new' value for δ, $\bar{\delta}$.

Substituting equation 3.41 into equation 3.40 gives:

$$\Delta = \frac{f|_{\mathbf{w}+r\mathbf{v}} - f|_{\mathbf{w}}}{\left(f|_{\mathbf{w}} + r \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}^{T} \mathbf{v} + \frac{1}{2} r^2 \bar{\delta}\right) - f|_{\mathbf{w}}} \tag{3.42}$$

$$\Delta = \frac{f|_{\mathbf{w}+r\mathbf{v}} - f|_{\mathbf{w}}}{r \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}^{T} \mathbf{v} + \frac{1}{2} r^2 \bar{\delta}} \tag{3.43}$$

and substituting for r (equation 3.39) gives:

$$\Delta = \frac{f|_{\mathbf{w}+r\mathbf{v}} - f|_{\mathbf{w}}}{\left(-\frac{\mathbf{v}^T \frac{\partial f}{\partial \mathbf{w}}}{\bar{\delta}}\right) \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}^{T} \mathbf{v} + \frac{1}{2}\left(-\frac{\mathbf{v}^T \frac{\partial f}{\partial \mathbf{w}}}{\bar{\delta}}\right)^2 \bar{\delta}} \tag{3.44}$$

$$\Delta = \frac{f|_{\mathbf{w}+r\mathbf{v}} - f|_{\mathbf{w}}}{-\frac{\left(\mathbf{v}^T \frac{\partial f}{\partial \mathbf{w}}\right)^2}{\bar{\delta}} + \frac{1}{2}\frac{\left(\mathbf{v}^T \frac{\partial f}{\partial \mathbf{w}}\right)^2}{\bar{\delta}}} \tag{3.45}$$

$$\Delta = \frac{2\bar{\delta}\left(f|_{\mathbf{w}} - f|_{\mathbf{w}+r\mathbf{v}}\right)}{\left(\mathbf{v}^T \frac{\partial f}{\partial \mathbf{w}}\right)^2}. \tag{3.46}$$

λ is then updated as follows:

$$\lambda(i+1) = \begin{cases} \frac{1}{4}\lambda(i) & \text{if } \Delta(i) > 0.75 \\ \lambda(i) + \frac{\mathbf{v}(i)^T \mathbf{H}(i) \mathbf{v}(i)(1-\Delta(i))}{\mathbf{v}(i)^T \mathbf{v}(i)} & \text{if } 0 < \Delta(i) < 0.25 \\ \lambda(i) & \text{otherwise} \end{cases} \tag{3.47}$$

where $\lambda(i)$ is the value of λ at iteration i.

Finally, a step is only taken if $\Delta > 0$. If $\Delta \leq 0$, the next iteration is started at the current point in weight space, in a new conjugate direction. The algorithm is started by setting the first value of λ, $\lambda(1)$, to be small. For the comparisons below, $\lambda(1) = 10^{-6}$.

Scaled conjugate gradient is a form of Levenberg-Marquardt optimization. The principle of this form of optimization is to trade-off steps based on a quadratic approximation, and gradient descent steps for when the quadratic approximation is poor. Another algorithm, based on this principle and applied to neural networks, was presented by Williams (Williams 1991). This algorithm is comparable in performance to scaled conjugate gradient.

A place for RBackprop

Møller uses a difference of gradients to approximate δ above. His approximation for δ, $\tilde{\delta}$, is given by:

$$\tilde{\delta} = \frac{\mathbf{v}^T \left(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+\eta\mathbf{v}} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}\right)}{\eta} \tag{3.48}$$

where η is a scaled step along the direction $\mathbf{v}$, given by:

$$\eta = \frac{\sigma}{\mathbf{v}^T\mathbf{v}} \tag{3.49}$$

where $0 < \sigma \leq 10^{-4}$.

RBackprop (given in appendix A) calculates $\mathbf{H}|_w\mathbf{v}$ exactly. This avoids the need to substitute $\tilde{\delta}$ for δ above, and eliminates the variables σ and η.

3.7.3 Variable metric methods

Optimization of non-linear functions is an active area of research within the field of applied mathematics. Since finding weights for a neural network is a non-linear optimization problem, also known as a non-linear numerical programming problem, it is sensible to apply some of the results of this research to the problem of finding optimal weights for a neural network.

Variable metric optimization schemes are an outcome from this work. They attempt to estimate the Hessian of a quadratic approximation to the objective function, thus building up second-order information about the surface. With an estimate of the Hessian, each step can attempt to jump straight to the minimum.

The Davidon-Fletcher-Powell (DFP) technique centres around an iterative update rule for an estimate of the inverse Hessian $\mathbf{A} = \mathbf{H}^{-1}$:

$$\mathbf{A}_{i+1} = \mathbf{A}_i + \frac{(\mathbf{w}_{i+1} - \mathbf{w}_i)(\mathbf{w}_{i+1} - \mathbf{w}_i)^T}{(\mathbf{w}_{i+1} - \mathbf{w}_i)^T\mathbf{d}_{i+1}} - \frac{(\mathbf{A}_i\mathbf{d}_{i+1})(\mathbf{A}_i\mathbf{d}_{i+1})^T}{\mathbf{d}_{i+1}^T\mathbf{A}\mathbf{d}_{i+1}} \tag{3.50}$$

where

$$\mathbf{d}_{i+1} = \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i+1} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i \tag{3.51}$$

is a column vector.

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) scheme differs only in an additional extra term in the update rule. Let the BFGS estimate of the inverse Hessian be denoted by $\mathbf{B}$. Then:

$$\mathbf{B}_{i+1} = \mathbf{A}_{i+1} + \mathbf{d}_{i+1}^T\mathbf{A}\mathbf{d}_{i+1}\mathbf{u}_{i+1}\mathbf{u}_{i+1}^T \tag{3.52}$$

where

$$\mathbf{u}_{i+1} = \frac{\mathbf{w}_{i+1} - \mathbf{w}_i}{(\mathbf{w}_{i+1} - \mathbf{w}_i)^T\mathbf{d}_{i+1}} - \frac{\mathbf{B}_i(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i+1} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i)}{(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i+1} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i)^T\mathbf{B}(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i+1} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i)}. \tag{3.53}$$

With both the schemes above, the search step $\boldsymbol{\rho}_{i+1}$ is given by:

$$\boldsymbol{\rho}_{i+1} = \mathbf{H}^{-1}(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i+1} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i) \tag{3.54}$$

where $\mathbf{H}^{-1}$ is approximated by $\mathbf{A}$ in DFP and $\mathbf{B}$ in BFGS.

The prescriptions above only give an accurate search step if the quadratic approximation is valid, and the inverse Hessian approximation is accurate. This will in general not be the case. The solution is to make a line search along the direction of the search step. As will be seen in section 3.9.4 below, the choice of line search affects the performance of the schemes. It is often better to use a quicker line minimization routine for these schemes than a slower, but more accurate, method.

3.8 Local optimization methods

The methods covered so far have grouped the weights together to find a search direction and step size. The weights were updated in proportion to their contribution to the global search direction and step size.

Local optimization methods consider local changes for each weight. The methods are not gradient descent methods. The step in weight space is not necessarily along a gradient of the objective function. Each weight is treated as though the others did not exist. Because the methods below make use of the previous step's local gradients, they are still second order methods.

3.8.1 Delta-bar-delta

This technique uses gradient descent for the search direction, and then applies individual step sizes for each weight, which means the actual direction taken in weight space is not necessarily along the line of the gradient. It was introduced by Jacobs (Jacobs 1988).

The basic idea is as follows: if the weight updates between consecutive iterations are in opposite directions, the step size is decreased, otherwise it is increased. This is prompted by the idea that if the weight changes are oscillating, the minimum is between the oscillations, and a smaller step size might find that minimum. The step size may be increased again once the oscillations have stopped.

Let $\boldsymbol{\eta}(i)$ be a vector of step sizes, one for each weight w_k, at iteration i. Then $\boldsymbol{\eta}$ is updated as:

$$\boldsymbol{\eta}(i+1) = \boldsymbol{\eta}(i) + \boldsymbol{\delta\eta}(i) \tag{3.55}$$

where $\boldsymbol{\delta\eta}(i)$ is a vector of changes for each learning rate, given by:

$$\delta\eta_k(i) = \begin{cases} \kappa & \text{if } \bar{\delta}_k(i-1) \left.\frac{\partial f}{\partial w_k}\right|_i > 0 \\ -\phi\eta_k(i) & \text{if } \bar{\delta}_k(i-1) \left.\frac{\partial f}{\partial w_k}\right|_i < 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.56}$$

where $\delta\eta_k(i)$ is the learning rate for weight w_k at iteration i, and

$$\bar{\delta}_k(i) = (1-\theta) \left.\frac{\partial f}{\partial w_k}\right|_i + \theta\bar{\delta}_k(i-1). \tag{3.57}$$

This procedure introduces three extra parameters that need to be set: κ, ϕ and θ. Jacobs finds values of the parameters that work well for particular tasks, but for the simulations below $\phi = 0.1$, $\theta = 0.7$, and κ was set to the same value as the initial learning rate for each weight, a vector of multiple copies of the default step size given in equation 3.17.

3.8.2 RProp

Schiffmann et al. found RProp to be the fastest training algorithm they tested in their review of training methods (Schiffmann et al. 1992).

RProp uses different step sizes for each weight, like delta-bar-delta. However, it only uses the sign of the local gradient $\frac{\partial f}{\partial w_k}$ when updating the weight w_k, not its magnitude.

The vector of step sizes $\boldsymbol{\eta}$ is defined as follows:

$$\eta_k(i) = \begin{cases} \min(1.2\eta_k(i-1), \eta_{\max}) & \text{if } \left.\frac{\partial f}{\partial w_k}\right|_i \left.\frac{\partial f}{\partial w_k}\right|_{i-1} > 0 \\ \max(0.5\eta_k(i-1), \eta_{\min}) & \text{if } \left.\frac{\partial f}{\partial w_k}\right|_i \left.\frac{\partial f}{\partial w_k}\right|_{i-1} < 0 \\ \eta_k(i-1) & \text{otherwise} \end{cases} \tag{3.58}$$

where $\eta_{\max}$ and $\eta_{\min}$ limit the size of the step above and below. The values used in the tests below were $\eta_{\max} = 1$ and $\eta_{\min} = 10^{-7}$. The maximum value was chosen to be about a tenth of the expected range of the weights. The minimum value was chosen to be about the expected necessary resolution of the weights. Reducing $\eta_{\min}$ to 10^{-40} produced a lower final value of the objective function for some tasks, but did not lead to faster or slower training otherwise. The step sizes were initialised using the default step size given in equation 3.17.

The weights were updated as follows:

$$\delta w_k(i) = \begin{cases} -\text{sign}\left(\left.\frac{\partial f}{\partial w_k}\right|_i\right)\eta_k(i) & \text{if } \left.\frac{\partial f}{\partial w_k}\right|_i \left.\frac{\partial f}{\partial w_k}\right|_{i-1} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.59}$$

such that $\mathbf{w}(i+1) = \mathbf{w}(i) + \boldsymbol{\delta}\mathbf{w}(i)$. A further detail concerns the stored value of the previous gradient $\left.\frac{\partial f}{\partial w_k}\right|_{i-1}$. If, for a particular weight, $\left.\frac{\partial f}{\partial w_k}\right|_i \left.\frac{\partial f}{\partial w_k}\right|_{i-1} < 0$, the value of the stored gradient for that weight would be set to zero for the next time step.

Riedmiller (Riedmiller and Braun 1993) and Robinson (Robinson 1994) have also independently invented RProp. Robinson applies his version of RProp to training with large data sets, where large implies that taking gradients of the objective function over the whole data set at once is infeasible.

3.8.3 Quickprop

Quickprop (Fahlman 1988) assumes a quadratic objective function for each weight. It implicitly assumes the Hessian is diagonal. The technique works well in practice, but requires adjustments to avoid instability.

Consider any single weight in the network. This weight is assumed to have a quadratic relationship with the objective function independent of the other weights. A quickprop step sets the weight to the minimum of the quadratic approximation.

This step may not be possible. Firstly, the quadratic may have a maximum and not a minimum. Secondly, there may not be enough information to approximate the quadratic. This can happen when the procedure is first started, but also if there has been no weight change between consecutive epochs. This is because the quadratic is approximated using a difference of gradients between the current and the previous epoch.

When the quickprop step breaks down, a gradient descent step is used to keep the procedure moving.

The details of the algorithm are as follows. A quickprop step for iteration i, $\boldsymbol{\delta}\mathbf{w}_{qp}(i)$, is defined by:

$$\boldsymbol{\delta}\mathbf{w}_{qp}(i) = -\frac{\left.\frac{\partial f}{\partial \mathbf{w}}\right|_i \boldsymbol{\delta}\mathbf{w}(i-1)}{\left.\frac{\partial f}{\partial \mathbf{w}}\right|_i - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{i-1}} \tag{3.60}$$

where $\boldsymbol{\delta}\mathbf{w}(i-1)$ is the weight update at the previous epoch, not necessarily the same as $\boldsymbol{\delta}\mathbf{w}_{qp}(i-1)$, since adjustments are necessary to improve convergence. When the denominator of equation 3.60 is zero for a weight, the quickprop step for that weight is set to zero.

The next steps mend the quickprop approximation. Firstly, the weight changes are limited by setting them to be the minimum of the quickprop weight change and 1.75 times the previous weight change. Secondly, steps are not taken that find a maximum of the quadratic approximation. These steps are set to 1.75 time the previous weight change. Finally, a gradient descent step is found to add to the weight changes above. This starts the algorithm when all the quickprop steps are zero, and helps to change weights whose previous changes were zero.

The adjustments amount to a hybrid weight change $\boldsymbol{\delta}\mathbf{w}(i)$, given by:

$$\delta w_k(i) = \begin{cases} \delta w_{qp\,k}(i) - \eta\frac{\partial f}{\partial w_k}(i) & \text{if } \frac{\partial f}{\partial w_k}(i)\frac{\partial f}{\partial w_k}(i-1) < 0 \\ \delta w_{qp\,k}(i) & \text{otherwise} \end{cases} \tag{3.61}$$

where η is a fixed step size, that may be set using the default size of equation 3.17.

A number of alternatives around the basic idea of the quickprop step can be used. For example, the gradient may be adjusted by adding 0.1 to the values of $\sigma'_k(x)$ used in the back-propagation algorithm used to calculate $\frac{\partial f}{\partial \mathbf{w}}$ (equation 3.14). The motivation for this step is to avoid the flat areas of the derivative of the node activation functions slowing convergence. Fahlman gives details of both this gradient adjustment, and modifications to the sum-squared error objective functions (Fahlman 1988).

3.9 Training scheme comparisons

The power of an optimization scheme rests on the appropriateness of the approximations made about the objective function. For example, if the objective function is well approximated by a quadratic, conjugate gradient methods may be expected to perform well. If the objective function is not quadratic, conjugate gradient methods might perform poorly. This is not to say that conjugate gradient descent is a poor optimization scheme. It says that the performance of an optimization scheme will depend on the match between the assumptions inherent in the scheme and the objective function of the task.

In practice it is desirable to know how robust an optimization scheme is, that is, how well an optimization scheme works on different training tasks. An indication of the robustness of the various schemes discussed above is given in a comparison of their performance for different tasks.

Among the factors that will affect the convergence of a neural network optimization scheme are:

1. The function to be mapped.
2. The network's node functions.
3. The architecture of the network.
4. The initial weight settings and the starting point in weight space.

The comparisons below vary the first three factors. The fourth was not studied in detail. Five training sets are used to show variation over the function to be approximated. Networks with tanh hidden units and log-sigmoid hidden units are compared. Finally, networks with and without direct connections from input layer to output layer are compared.

3.9.1 The tasks

To measure the performance of the training schemes described here, six different training tasks were set. They are:

sin Sine function

abs Absolute value function

xor Exclusive-or

10encoder 10-5-10 encoder

simpole 5 input, 4 output real-valued inverted pendulum data

classify 2 input, 3 output classification problem

The sine and absolute value tasks have been included because good performance on the sine task was not necessarily followed by good performance on the absolute value task.

The sine task used input values $\mathbf{x} = [-3, -2.8, -2.6, \cdots, 2.8, 3]$. The targets were $\sin(\mathbf{x})$. The absolute value task used the same input values. The targets for this task were the absolute values of the inputs, $|x|$.

Exclusive-or is a standard task for neural networks, of little practical importance but often quoted in benchmark tests. The four input patterns were $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$. The targets were respectively 0.1, 0.9, 0.9 and 0.1.

The 10-5-10 encoder task required the network to produce a description of the 10 input nodes in the hidden layer, so that the input could be reproduced at the output. Ten input patterns were used, the ten rows of a 10 by 10 identity matrix, so the inputs alternately turned on a single node of the input. The targets were identical to the inputs for this task. This task is trivial for a fully connected network, but it is still a valid training task to compare different optimization schemes.

The inverted pendulum data was generated from a simulation of an inverted pendulum system using the Barto parameters given in appendix B. The state was set to random positions in the four-dimensional state space. A force was applied, and the state 0.02 seconds after the force was applied was found by Euler integration of the accelerations. The initial state and control action are the inputs to the network, the change in the state is the target. 270 input/target pairs were used.

The classification problem was to separate three Gaussian random variables in two dimensions, each with the same covariance matrix but with different means. To separate classes with different covariance matrices, additional cross-term inputs to the network would be needed. This was not attempted.

The data for the classification problem is shown diagrammatically in figure 3.4. It was generated using a covariance matrix $\mathbf{C}$ given by:

$$\mathbf{C} = \begin{bmatrix} 0.95 & 0 \\ 0 & 1.05 \end{bmatrix}. \tag{3.62}$$

rotated through 60°. The means for the three classes were set to $\begin{bmatrix} 0 & 0 \end{bmatrix}$, $\begin{bmatrix} 2 & 0 \end{bmatrix}$ and $\begin{bmatrix} 0 & 2 \end{bmatrix}$.

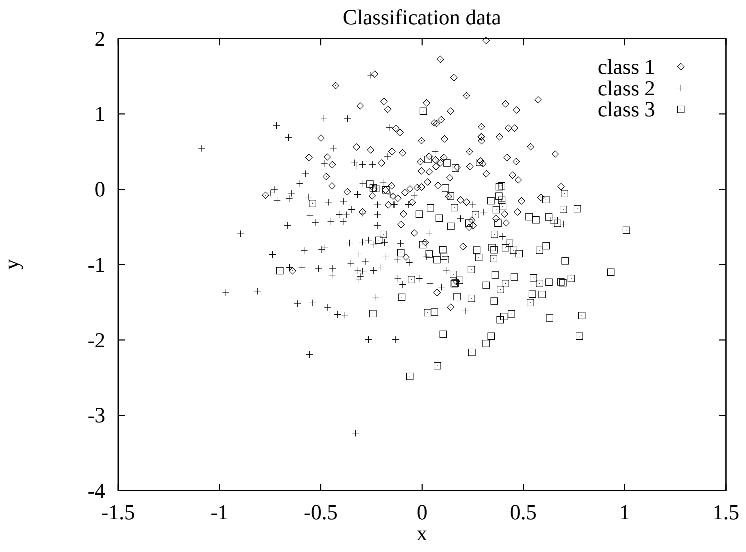


Figure 3.4: The data for the classification task.

3.9.2 The networks

Different networks were applied to the tasks. Some networks were not naturally suited to the tasks, for example the encoder problem is trivial with a fully-connected feed-forward network. However, each task forms an objective function for a network, and so the optimization schemes may still be assessed on their relative performance.

The networks were varied around a default network. The default network had a single hidden layer of five tanh nodes, and an output layer of linear nodes. The number of input and output nodes was defined by each task. The connections were layered.

Comparisons were then made by modifying the default network by using 2 and then 8 hidden nodes. Then the performance was compared by modifying the default network to use log-sigmoid instead of tanh hidden nodes. Finally, results for a default network modified for full connectivity were obtained. Full connectivity is sensible for a network designed to predict the next state of continuous control plant given the current state and control action, since in this case the target is only a perturbed version of the state component of the input vector. Full connectivity allows a network to include a linear component in its interpolation. It is useful when a linear regressor accounts for much of the trend in the data.

The initial weights for all the networks were sampled from a Gaussian distribution with zero mean and variance of 0.04.

3.9.3 The training schemes

The training schemes compared in the results are:

gdmom Gradient descent with momentum

hscgrbp Hestenes-Stiefel conjugate gradient descent with RBackprop

scg Scaled conjugate gradient

scgrbp Scaled conjugate gradient descent with RBackprop

rprop RProp

deltabardelta Delta-bar-delta

qp Quickprop

bfgslnsrch BFGS with lnsrch (Press et al. 1992)

bfgslinmin BFGS with linmin (Press et al. 1992)

dfplnsrch DFP with lnsrch

dfplinmin DFP with linmin

The momentum parameter α for gdmom was set to 0.9. Section 3.9.4 discusses tuning the parameters of this scheme.

The Hestenes-Stiefel conjugate gradient scheme used the Hestenes-Stiefel rule (equation 3.26 to find conjugate search directions. The step size was set by the quadratic-optimal step size given in equation 3.34. RBackprop was used to calculate the product of the Hessian and the search direction.

The scaled conjugate gradient method also used RBackprop to calculate the product of the Hessian and the search direction. The details are given by Møller (Møller 1993), save for the substitution of RBackprop in place of Møller's approximation to compute the product of the Hessian and the current search direction. Although the accuracy of RBackprop is not required (since the product is used in an approximation), it eliminates an extra parameter from the algorithm.

3.9.4 Results

The results plot evaluations of the objective function for each optimization scheme over a series of iterations for the different tasks and networks. The training error was generated as follows: predict each target in the data set; find the difference for each element of each prediction; square it; add all the squares up and divide by the number of patterns in the data set.

Most optimizers were run for 10^6 floating-point operations (flops). This is why plots of the objective function against the number of epochs have different lengths.

The majority of the results are given as plots of the objective function (usually training error) against the number of flops used by each optimization algorithm. Using flop-counts rather than execution time allows the data to be independent of variations in the computing load not directly related to the optimization, and gives an indication of the performance that might be expected using careful programming.

Figure 3.5 shows the difference between plotting against computational effort, and plotting against the number of epochs. Gradient descent is computationally cheap per epoch. For the same number of floating-point operations, more epochs of gradient descent may be completed. The graphs show, however, that the extra epochs do not make up the difference. Quickprop is also able to do many iterations for the same computational effort. However, quickprop's extra epochs allow it to compete with the conjugate gradient techniques, often winning, given the same computational constraints. The graphs are not identical in length, because the granularity of the floating-point requirements of the various procedures in the training loop interfere with the stopping criterion of 10^6 flops.

The link between floating-point operations and processing (CPU) time is shown in figure 3.6. Plotting the training error against flops rather than CPU time allows the results to be independent of the available computational power, and to a certain extent the efficiency of the schemes' implementations.

Figure 3.9 gives the relative performances of the optimization schemes for the single-input, single-output (SISO) sine and absolute value tasks on the default network of five tanh nodes in the hidden layer, and a layered connectivity. These tasks were chosen for comparison because schemes that worked well on the sine task were found to work less well on the absolute value task.

The SISO tasks also allow the interpolation that the network performs to be plotted. These interpolations are shown in figures 3.7 and 3.8. They show the relationship between the size of the training error and the resulting interpolation.

Figure 3.10 gives the relative performances of the optimization schemes for the xor, encoder and inverted pendulum data tasks.

Figures 3.11 and 3.12 plot the results of the training schemes on the training tasks for a default network modified by using only 2 nodes in the hidden layer. Figures 3.13 and 3.14 plot the results using 8 hidden layer nodes.

The effect of changing the non-linearity in the hidden layer is shown in Figures 3.15 and 3.16. These figures plot the training error for the tasks and the optimization schemes using the default network with five log-sigmoid hidden layer nodes.

Finally, the effect of changing the connectivity is shown in figures 3.17 and 3.18. These figures plot the results when a set of weighted links is added between the input and the output nodes.

Gradient descent with momentum

Gradient descent with momentum is a popular training algorithm, and so a small section is devoted to it here. Figure 3.19 shows the results of training the default network on the inverted pendulum data. The runs ran for 10^7 flops. The following schemes were used:

gd Steepest descent;

gdmom Gradient descent with momentum, $\alpha = 0.9$;

gdmom0.1 Gradient descent with momentum, $\alpha = 0.1$;

gdmomB Gradient descent with momentum, equation 3.63, $\alpha = 0.9$;

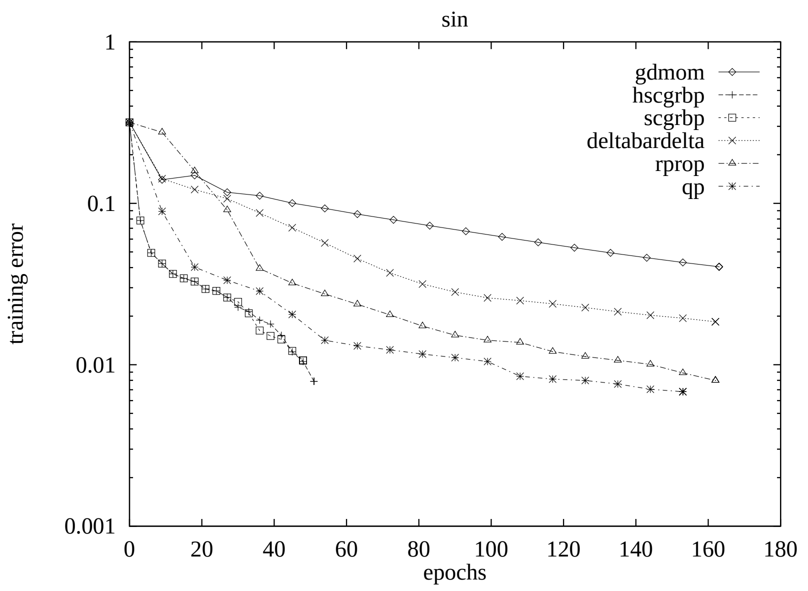
gdmomB0.1 Gradient descent with momentum, equation 3.63, $\alpha = 0.1$;

qp Quickprop.

All the gradient descent schemes used the default step size given by equation 3.17. Updating with equation 3.63 sets the weight change $\boldsymbol{\delta}\mathbf{w}$ by:

$$\boldsymbol{\delta}\mathbf{w}(i) = -(1-\alpha)\eta \left.\frac{\partial f}{\partial \mathbf{w}}\right|_i + \alpha\boldsymbol{\delta}\mathbf{w}(i-1), \qquad (3.63)$$

differing from the conventional update given in equation 3.20 by an added $(1-\alpha)$ term.



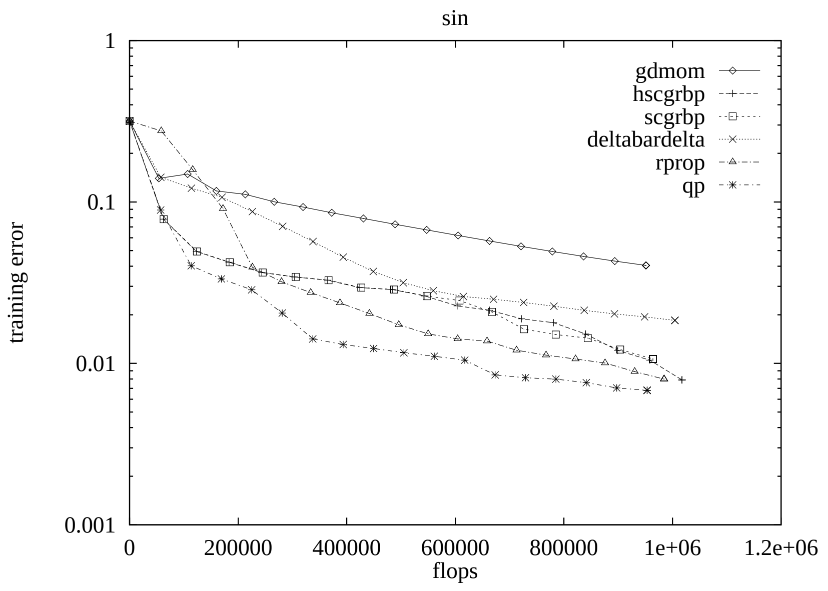


Figure 3.5: The upper graph plots the training error against epoch for the sine task on the default network. The lower graph plots the training error against the number of floating-point operations. The advantage of the advanced schemes over gradient descent appears diminished in the lower graph, but it is still clear that they are more efficient.

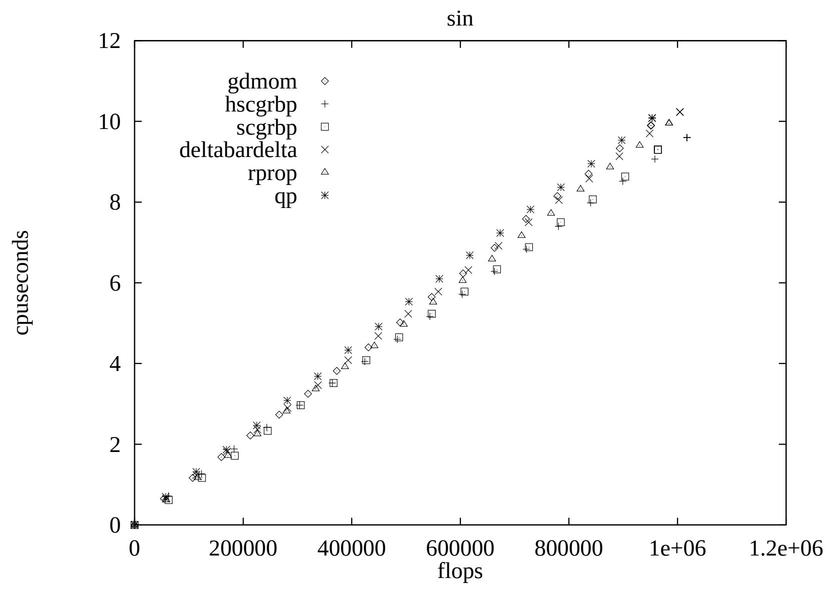


Figure 3.6: The linear relationship between flops and cpu seconds verifies that the floating-point count is linearly related to the computation time.

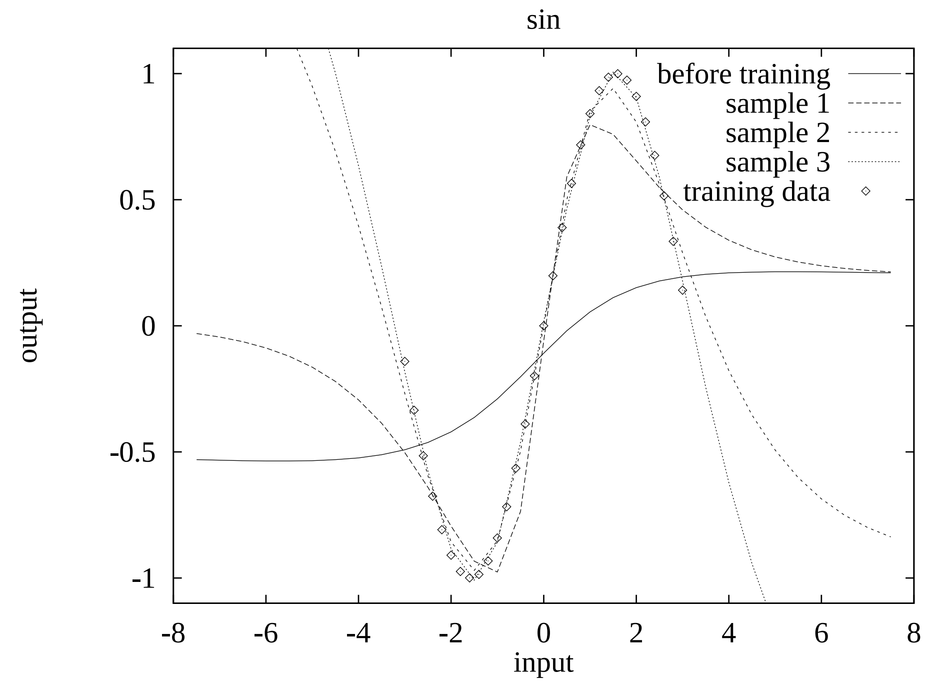


Figure 3.7: Interpolation for the sine data set. The objective function is the sum-squared error over the data. The values of the objective function are: before training, 8.78; sample 1, 0.928; sample 2, 0.0961; sample 3, 0.00975. The graph shows the connection between the sum-squared error values, and the quality of the interpolation that results.

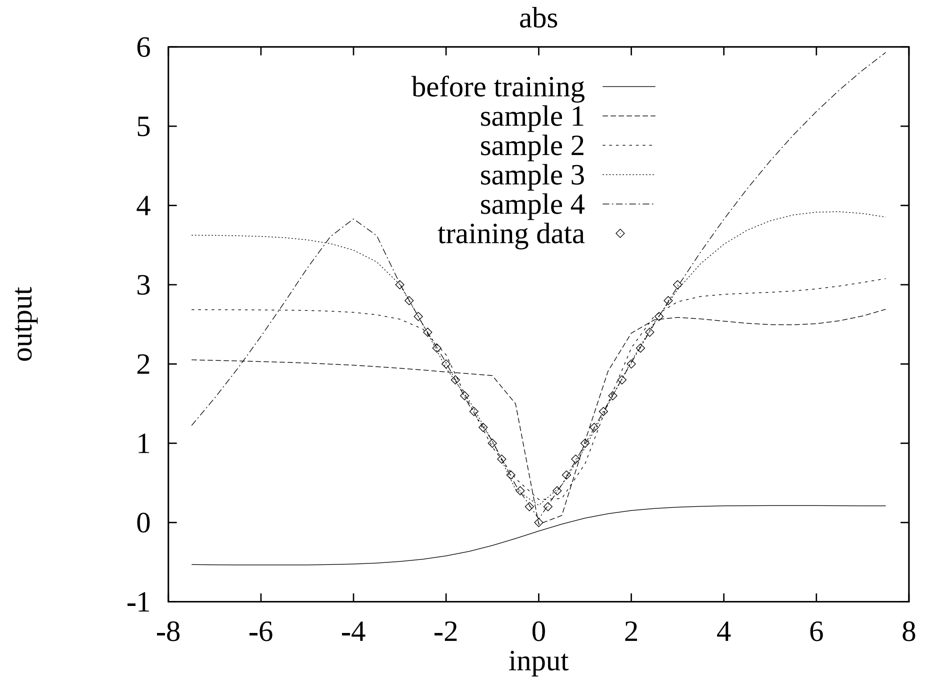


Figure 3.8: Interpolation for the abs data set. The objective function is the sum-squared error over the data. The values of the objective function are: before training, 114.475; sample 1, 8.63; sample 2, 0.886; sample 3, 0.0996; sample 4, 0.00998. Moving from sample 3 to sample 4 took 85% of the 82 seconds of training time using bfgslnsrch.

Quickprop is included in figure 3.19 to emphasize its improvement over standard gradient descent.

The graph shows that gradient descent with momentum, setting $\alpha = 0.9$ and using equation 3.20, was the best first-order gradient descent method tried. It was also the best first-order method when tested with the other tasks.

The results show that the second-order methods do better. It is possible that a first-order method could be found with more appropriate step size and momentum parameters, which might approach the performance of the second-order methods. However, the effort required to tune the parameters seems pointless, since the second-order methods already work.

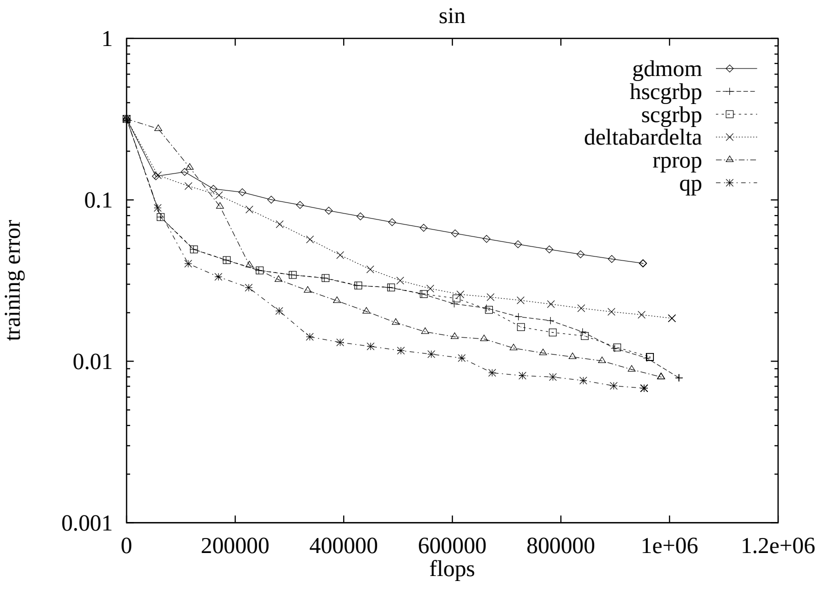
The effect of RBackprop on scaled conjugate gradient

Figure 3.20 plots the training error for the same tasks comparing scaled conjugate gradient (scg) using Møller's Hessian approximation, and scaled conjugate gradient using RBackprop (scgrbp). The results show similar performance for all the problems.

Variable metric methods

Figure 3.21 plots the training error for the same tasks comparing the two variable metric methods DFP and BFGS, with two line minimization schemes each (linmin and lnsrch (Press et al. 1992)), and what might be considered to be among the best of the alternative schemes, scaled conjugate gradient and quickprop. The results show BFGS with lnsrch to be an efficient optimization method. The results also show the dependence the variable metric methods have on the line search method. In this context, lnsrch is more efficient than, although not as accurate as, linmin. Line minimization is discussed in section 3.6.2.

The variable-metric results in figure 3.21 come from 10^6 flops for the sin, abs, 10encoder and xor tasks and 10^7 flops for the simpole task. The extra number of flops were needed for the latter



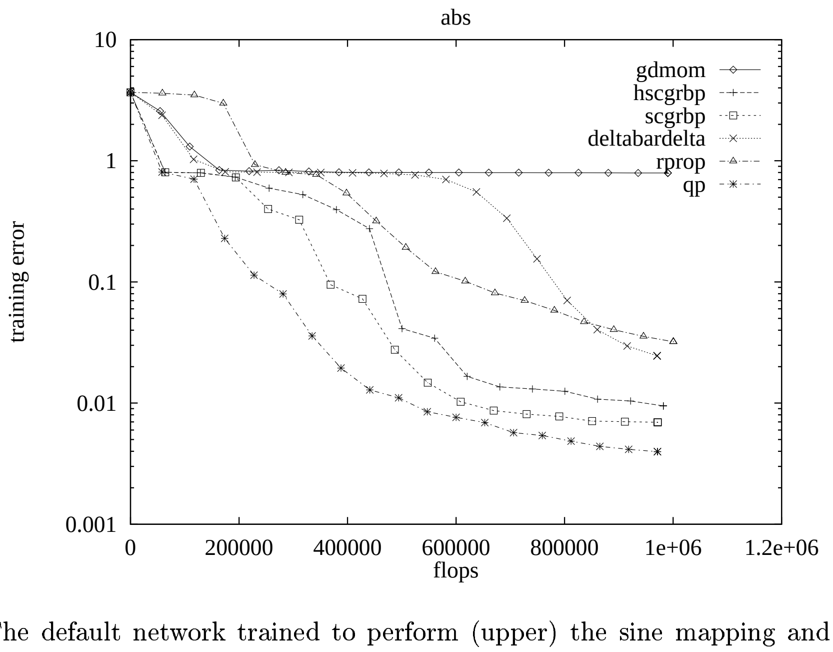
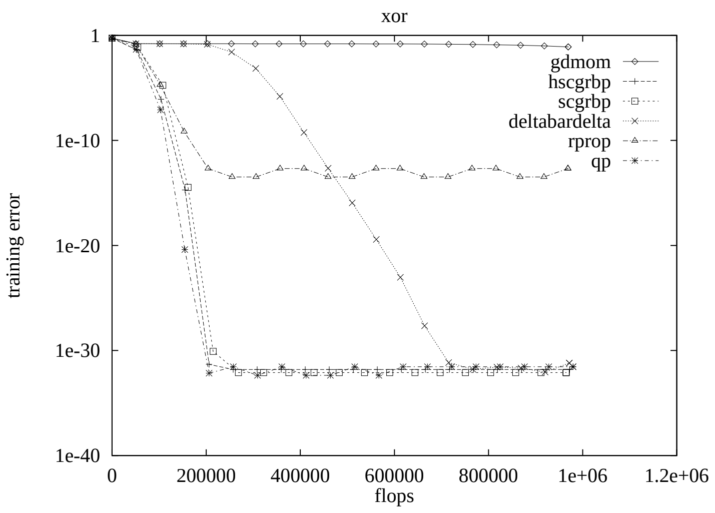
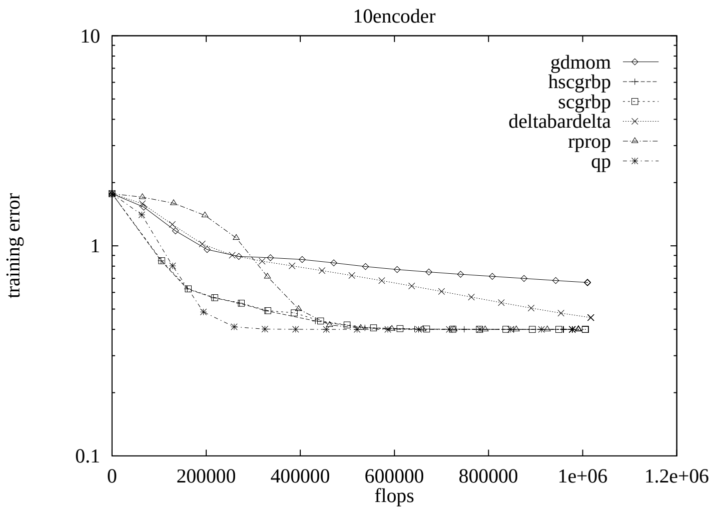


Figure 3.9: The default network trained to perform (upper) the sine mapping and (lower) the absolute value mapping.





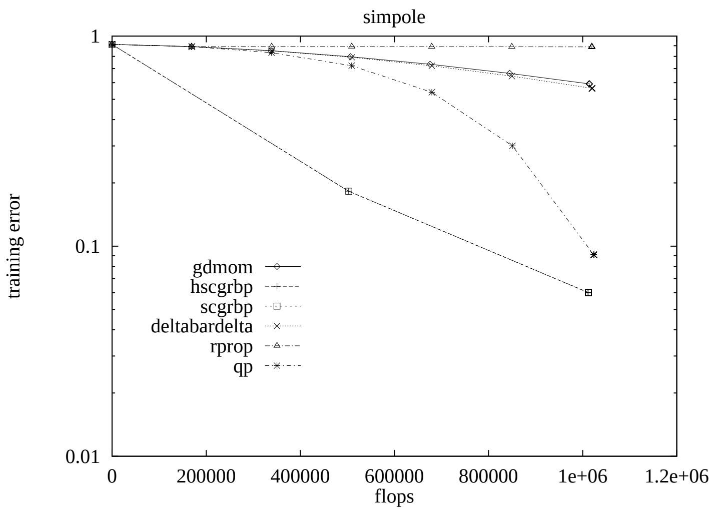
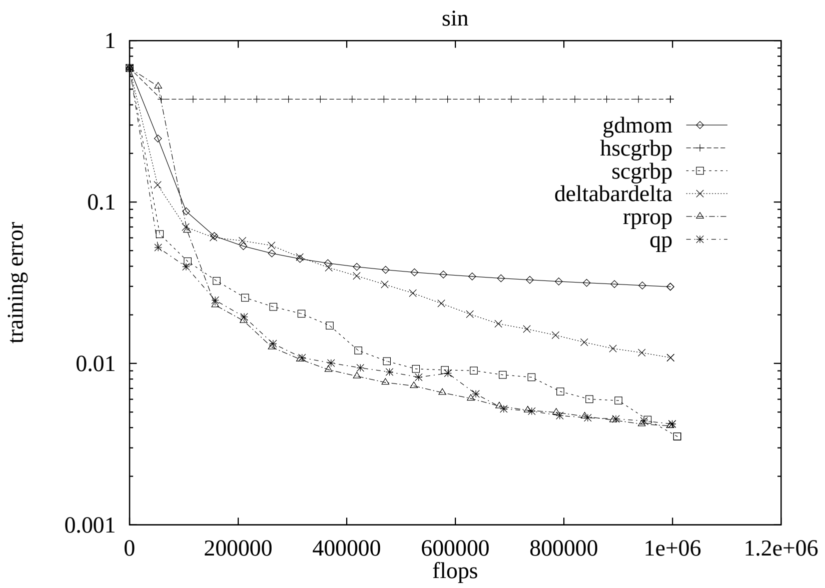


Figure 3.10: The default network trained on xor, the encoder problem and the inverted pendulum data.



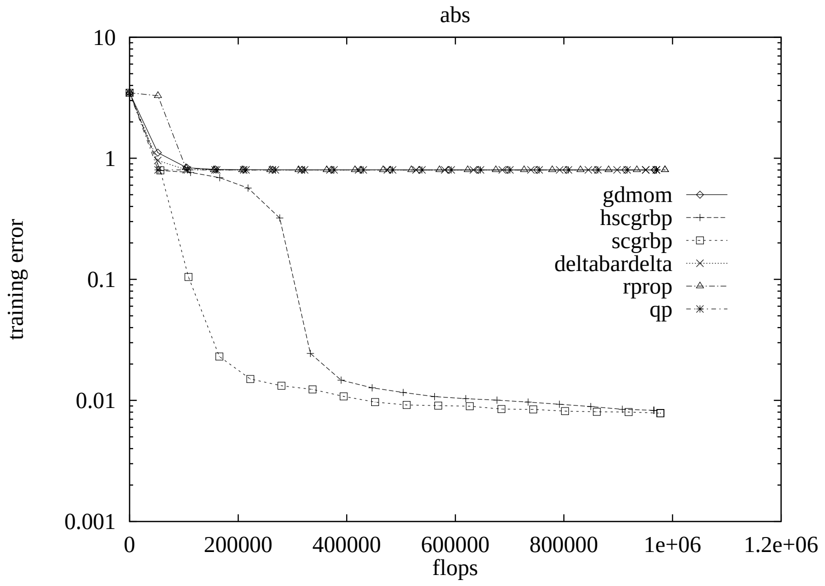


Figure 3.11: The default network with 2 hidden nodes trained on sine and absolute value functions.

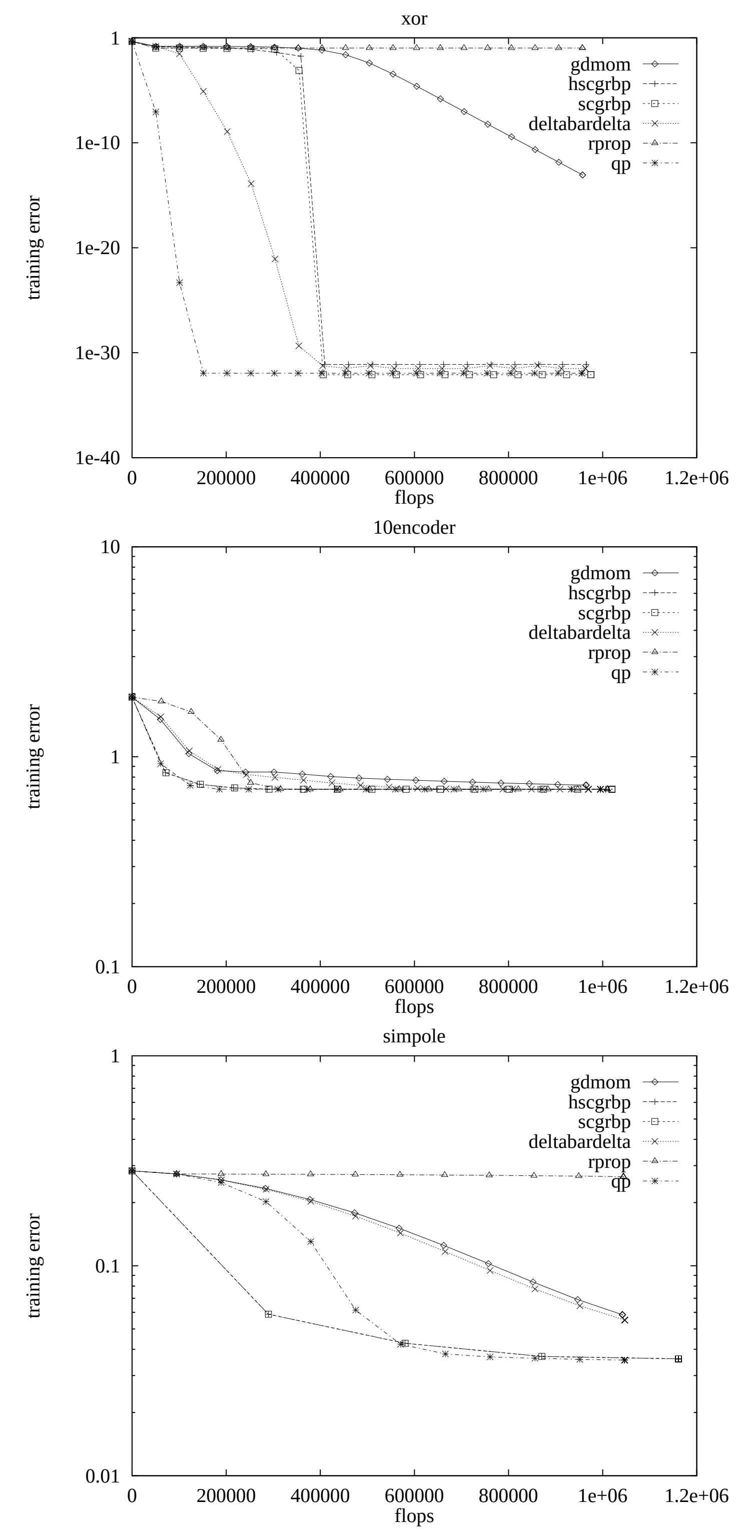
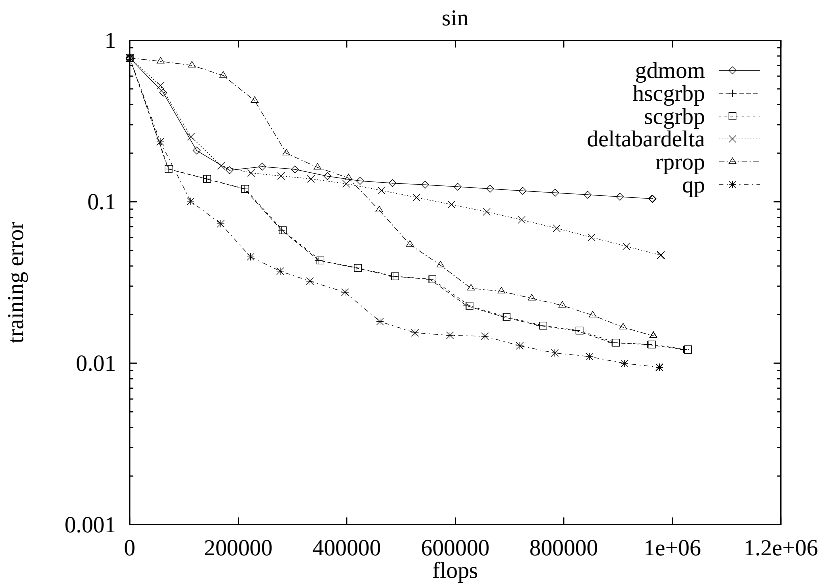


Figure 3.12: The default network with 2 hidden nodes trained on xor, the encoder problem and the inverted pendulum data.



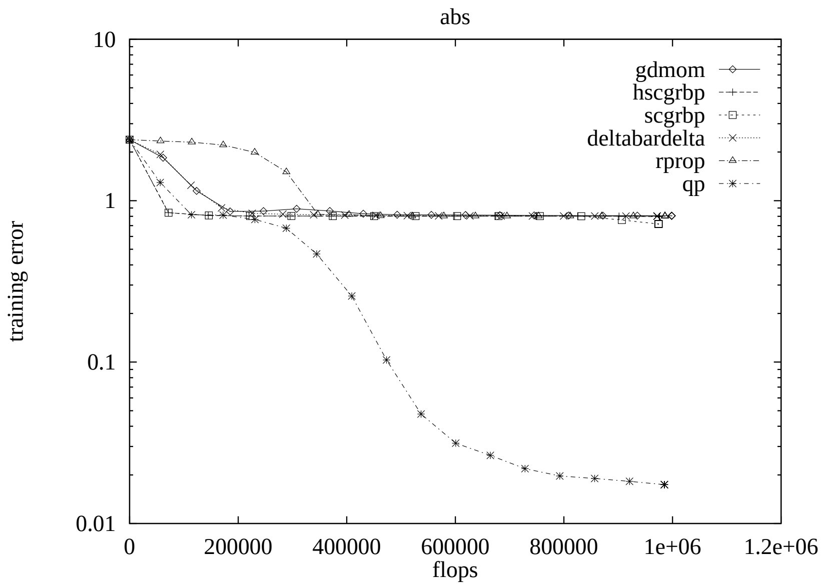


Figure 3.13: The default network with 8 hidden nodes trained on sine and absolute value functions.

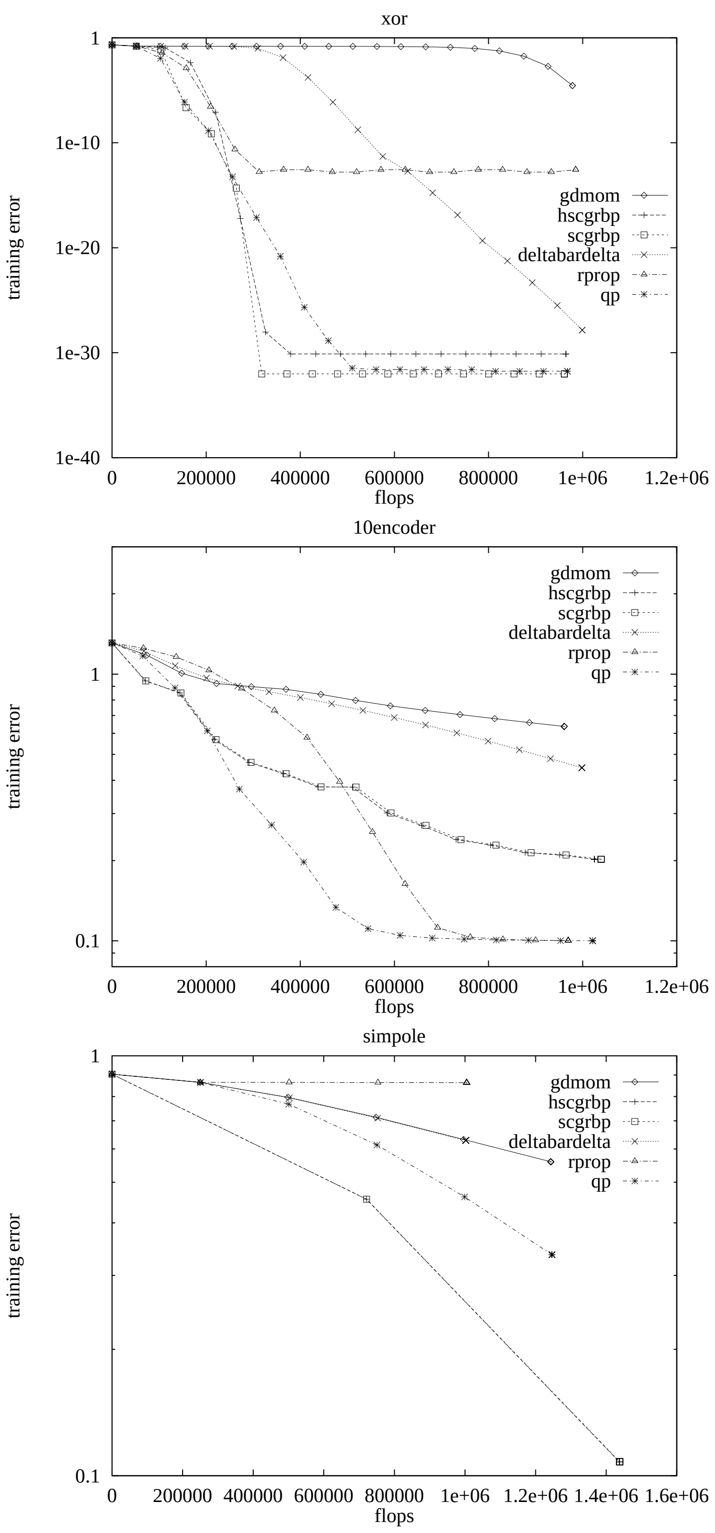
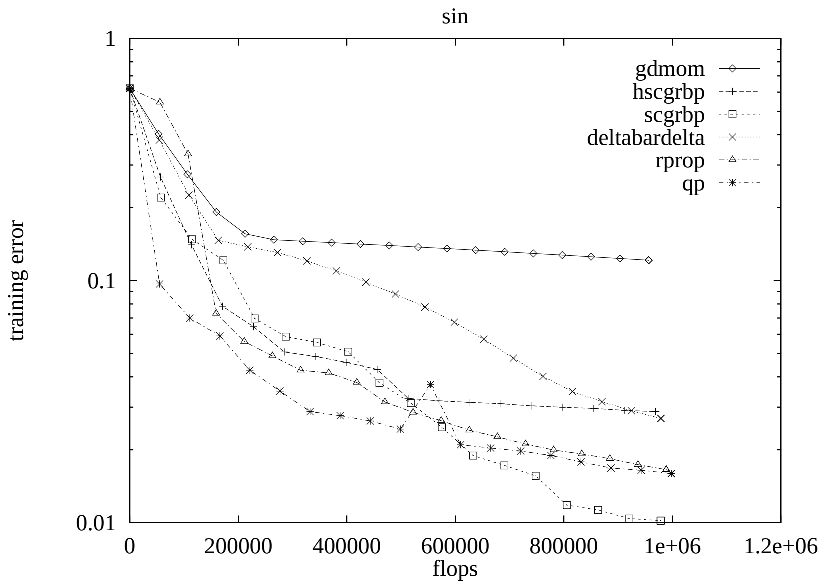


Figure 3.14: The default network with 8 hidden nodes trained on xor, the encoder problem and the inverted pendulum data.



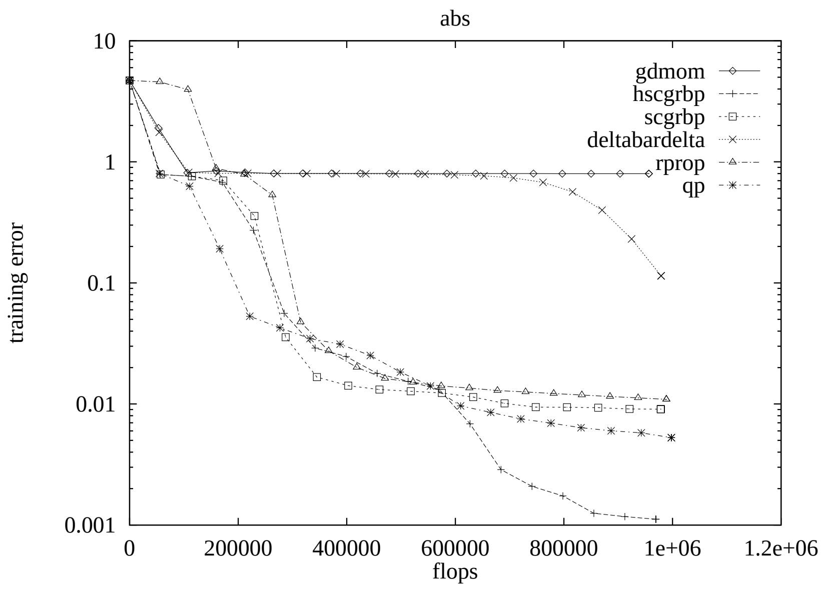


Figure 3.15: The default network with log-sigmoid non-linearities trained on sine and absolute value functions.

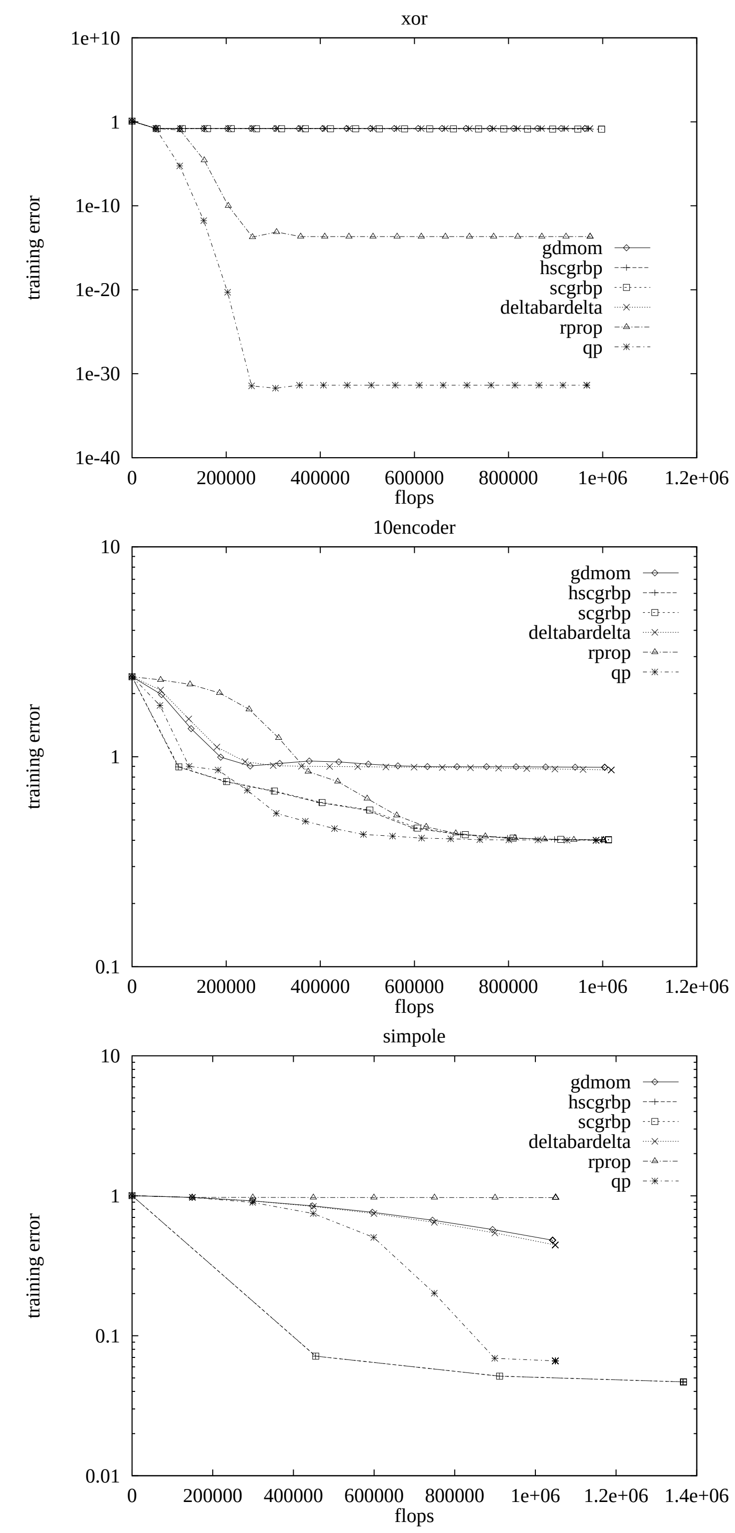
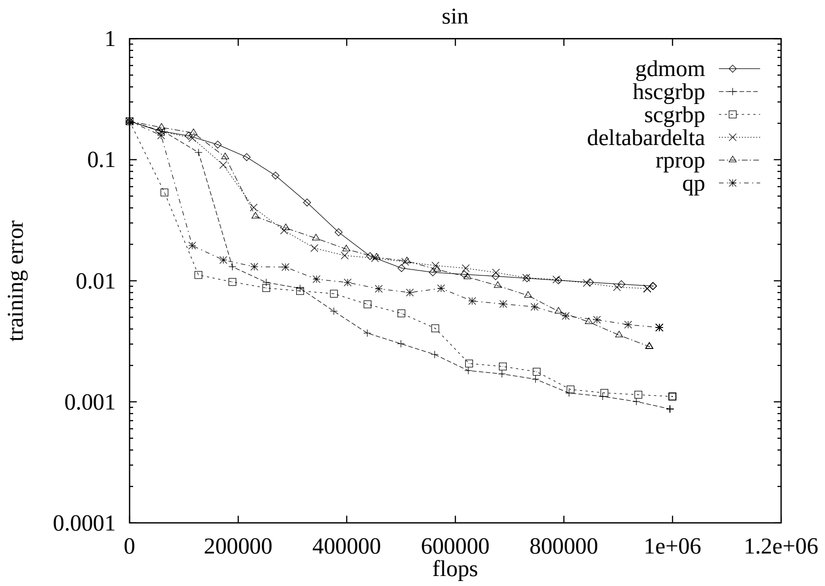


Figure 3.16: The default network with log-sigmoid non-linearities trained on xor, the encoder problem and the inverted pendulum data.



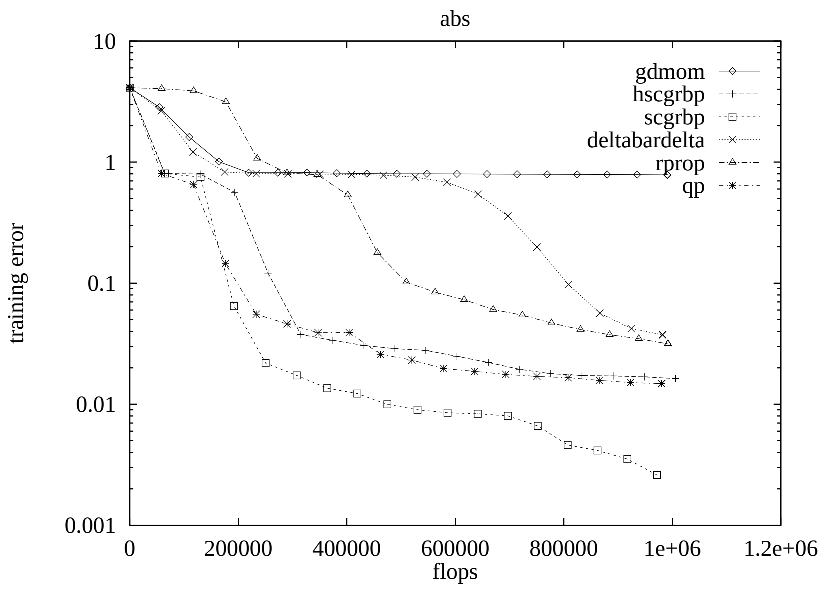


Figure 3.17: The default network with full connectivity trained on sine and an absolute value functions.

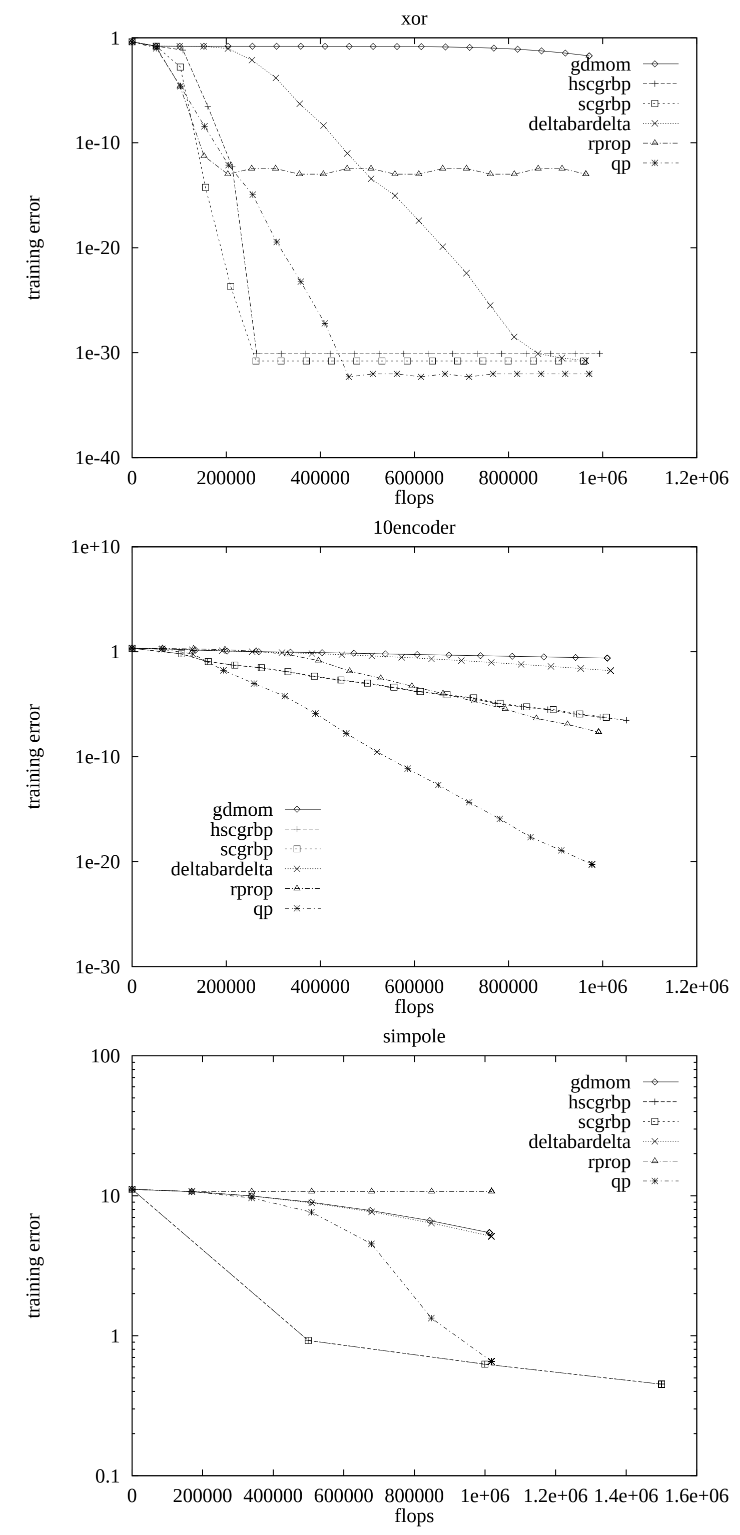


Figure 3.18: The default network with full connectivity trained on xor, the encoder problem and the inverted pendulum data.

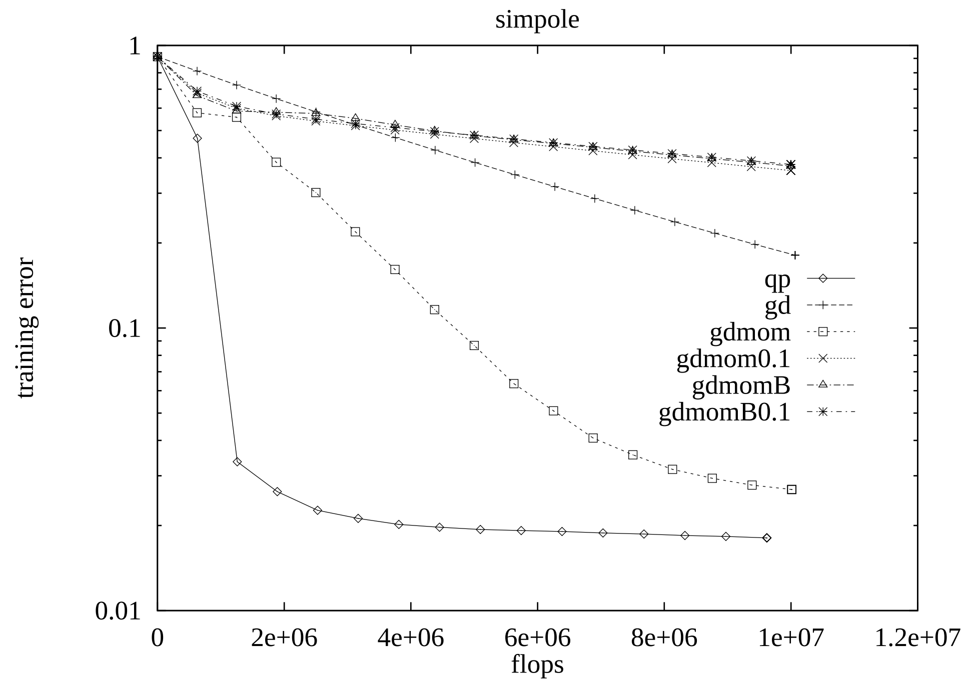


Figure 3.19: Gradient descent schemes training the default network on the inverted pendulum data.

task so a more representative number of iterations could be displayed.

The probabilistic objective function

Figure 3.22 shows the results of training the default network with full connectivity on the classification task using the softmax function. The algorithms were each given 10^8 flops to run. The objective function is the negative of the log probability of the data set, so that minimizing the objective function maximizes the log probability.

3.9.5 Conclusions

A neural network optimization scheme makes assumptions about the objective function's surface in weight space. The success of a scheme will depend on the validity of the assumptions. For different problems, the assumptions will be appropriate to varying degrees. This means there is no optimization scheme that will perform better than all other schemes on all possible tasks.

There exist neural network training algorithms that perform better than gradient descent with momentum, the traditional optimization method for neural networks. These algorithms find the minima of objective functions with less computational effort and therefore less time than the standard methods.

There is no optimization scheme that may be recommended without reservation. For a particular kind of task, a number of optimization schemes should be compared to find which ones are efficient. Candidate schemes might include scaled conjugate gradient and quickprop.

The benefits of using an appropriate optimization scheme are substantial. For some tasks, the best schemes produce objective function values that are orders of magnitude lower than gradient descent for the same computational effort. This kind of improvement is the kind necessary to make neural networks useful in the field of on-line adaptive control where data and time are precious.

Once the algorithms are fast enough, the most important factor in real time control is the need to be efficient with experience, learning as much as possible in as few trials as possible. With the

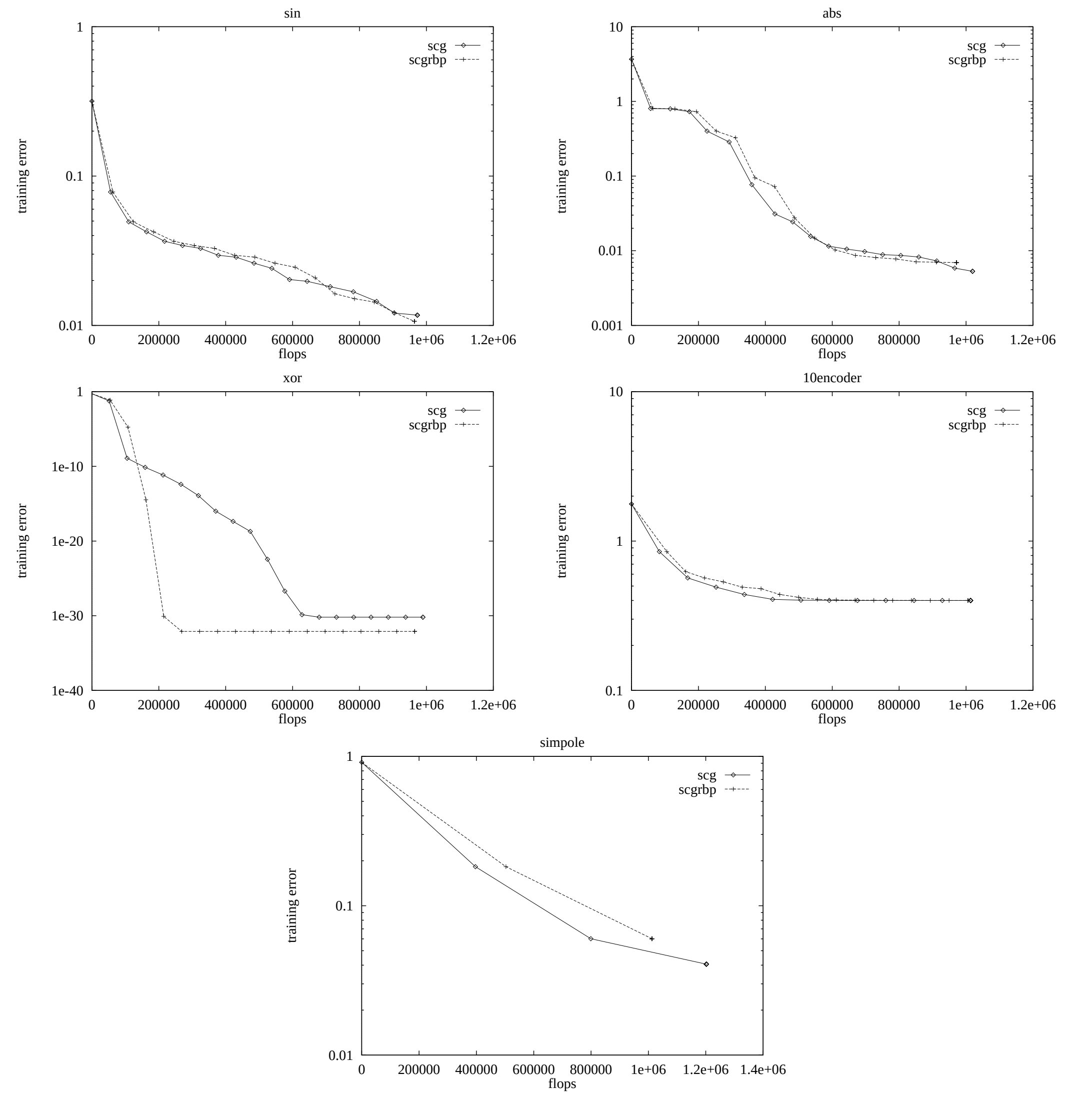


Figure 3.20: A comparison of scaled conjugate gradient with the original approximation used by Møller, and scaled conjugate gradient with RBackprop. The results of training the default network on each of the five tasks are shown.

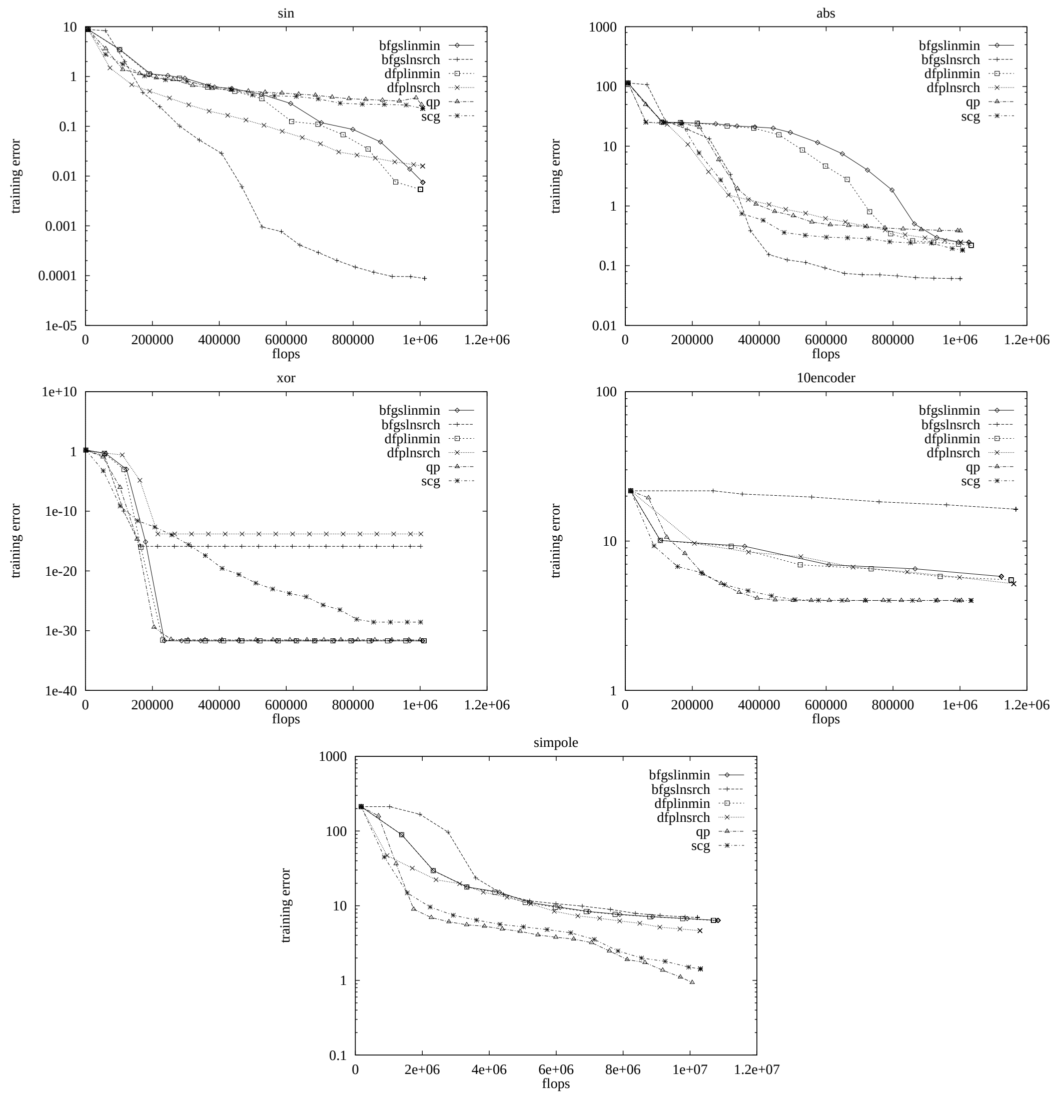


Figure 3.21: A comparison of two variable metric methods with two different line search schemes with scaled conjugate gradient and quickprop. The default network was trained on each of the five tasks.

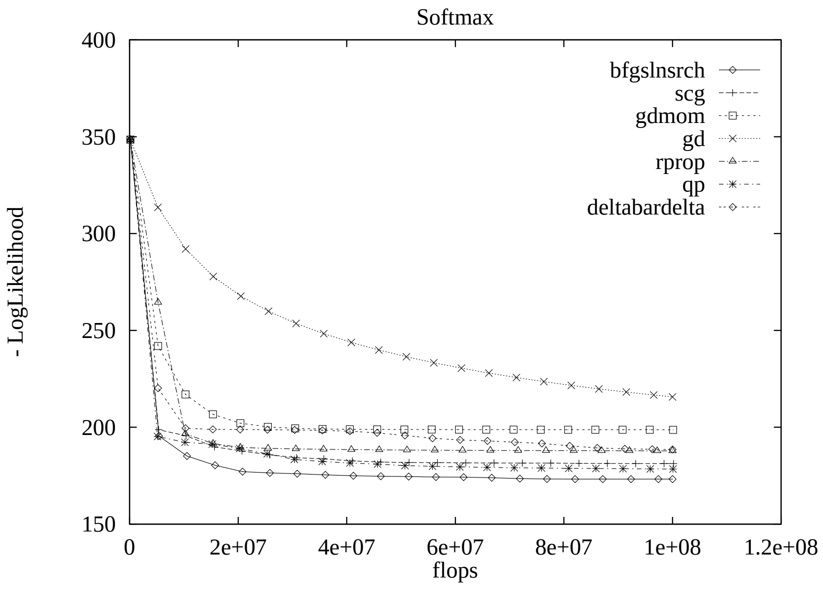


Figure 3.22: Performance of the optimizers on the classification task.

best of the optimization methods described here, and fast hardware, the computational demands of real-time control might be met. Beyond this, the issue of learning efficiently from the available data comes to the fore. These issues are considered in chapter 6.

3.10 Training without targets

Neural networks are normally trained by supervised methods. The methods rely on known input/output pairs. In a learning control problem, the input/output, or context/action, pairs may not be known. There is a way, however, to train neural networks efficiently without requiring the explicit input/output pairs needed for supervised learning (Munro 1987)

Figure 3.23 shows the idea. Two networks, networks 1 and 2, are combined to form a single network. Network 1 can be trained to control the input to network 2, so that the combination forms some desirable function, within the constraint of network 2 forming some other fixed function.

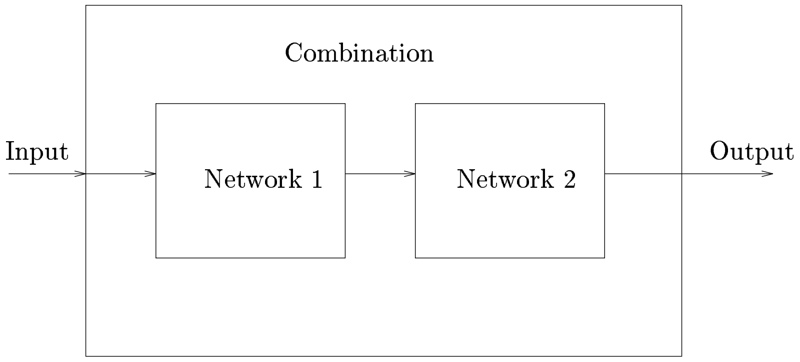


Figure 3.23: Two networks joined to form a combination. The mapping formed by network 2 is given. Network 1 is to be adapted so that the combination forms a suitable mapping.

Here is an example. Figure 3.24 shows network 2 trained to produce a sinusoid. The network was trained with supervised learning using an optimization technique (the Broyden-Fletcher-

Goldfarb-Shanno, or BFGS, algorithm (Press et al. 1992)) that used a gradient of the sum-squared error over the training data.

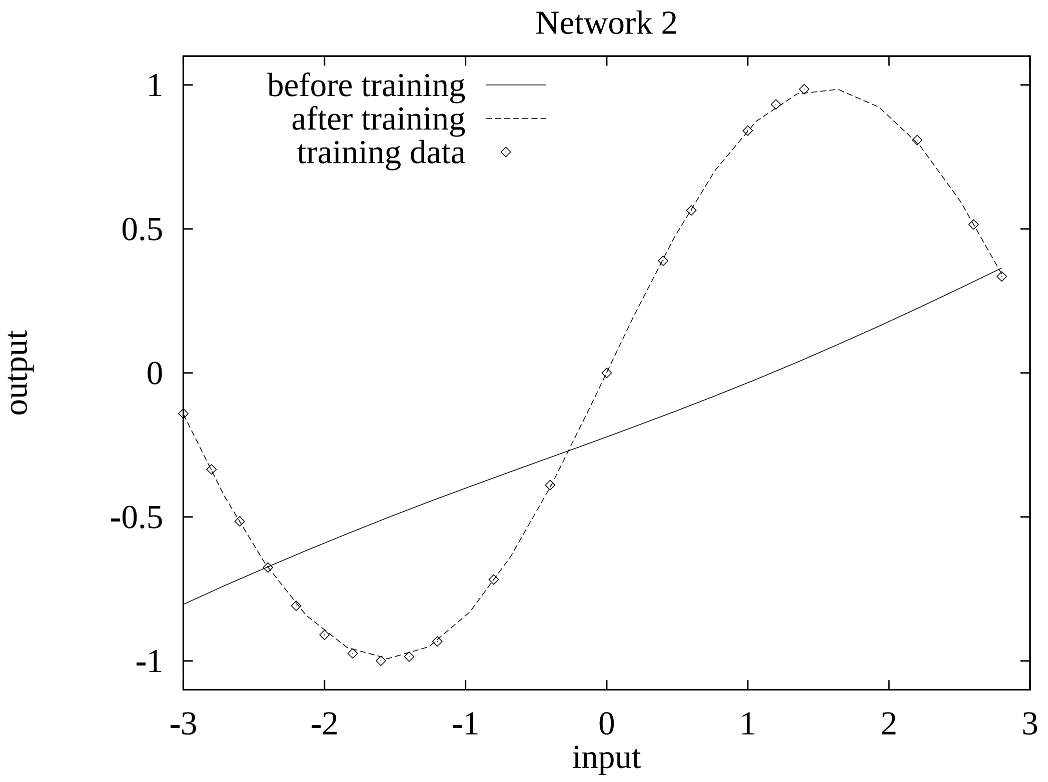


Figure 3.24: Network 2: a 1-5-1 network trained to form a sine wave. The training data are marked as points. The interpolating curves of the network before and after training are also shown.

Returning to the context of control, network 2 would be emulating a plant to be controlled. The network might be trained with input/output data gathered from a selection of control inputs to the plant.

The front-end network (network 1) forms an input to network 2 so that the output of the original network is appropriate for the new mapping. Figure 3.25 shows the combination of networks 1 and 2 forming a cosine wave. Network 2 is left unmodified and so continues to produce the sinusoid mapping given in figure 3.24.

The mapping across networks 1 and 2 is adjusted only by changing the weights of network 1. The optimization could be attempted using techniques that do not require gradient information, such as the simplex method of Nelder and Mead, or Powell's method (Press et al. 1992). These techniques will not be efficient, however, and will require long training times.

Network 1 can be trained more efficiently using gradient-based optimization methods. This requires the gradient of the objective function with respect to the weights of network 1 to be known. These derivatives can be found by applying the back-propagation algorithm (section 3.5) to network 2, which yields derivatives of the objective function with respect to the inputs of network 2. These derivatives can then in turn be back-propagated through network 1 to give the final derivatives of the objective function with respect to the weights of network 1.

The mapping across the two networks shown in figure 3.25 was generated by optimizing the weights of network 1 using the BFGS algorithm and the derivatives calculated using back-propagation through network 2.

Figure 3.26 shows the mapping that network 2 has formed in order that the combination of networks 1 and 2 forms a cosine. There are no explicit training data for this function.

In the context of control, network 1 would take the form of a controller for the plant. The rôle

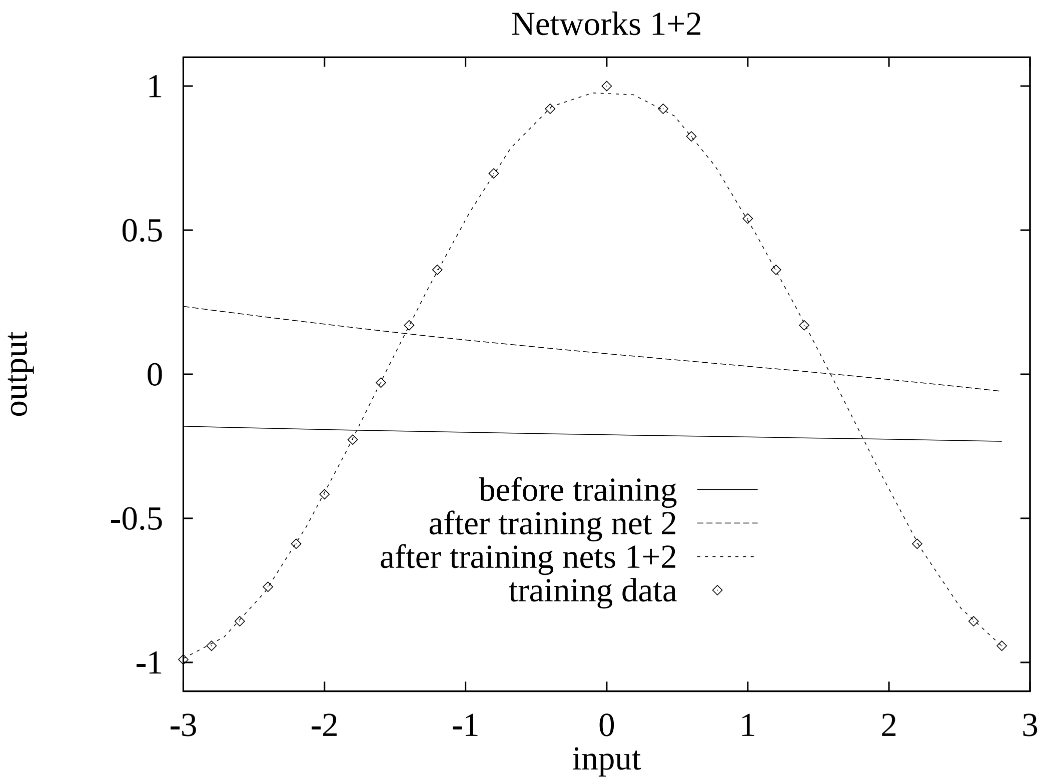


Figure 3.25: The mapping across both networks.

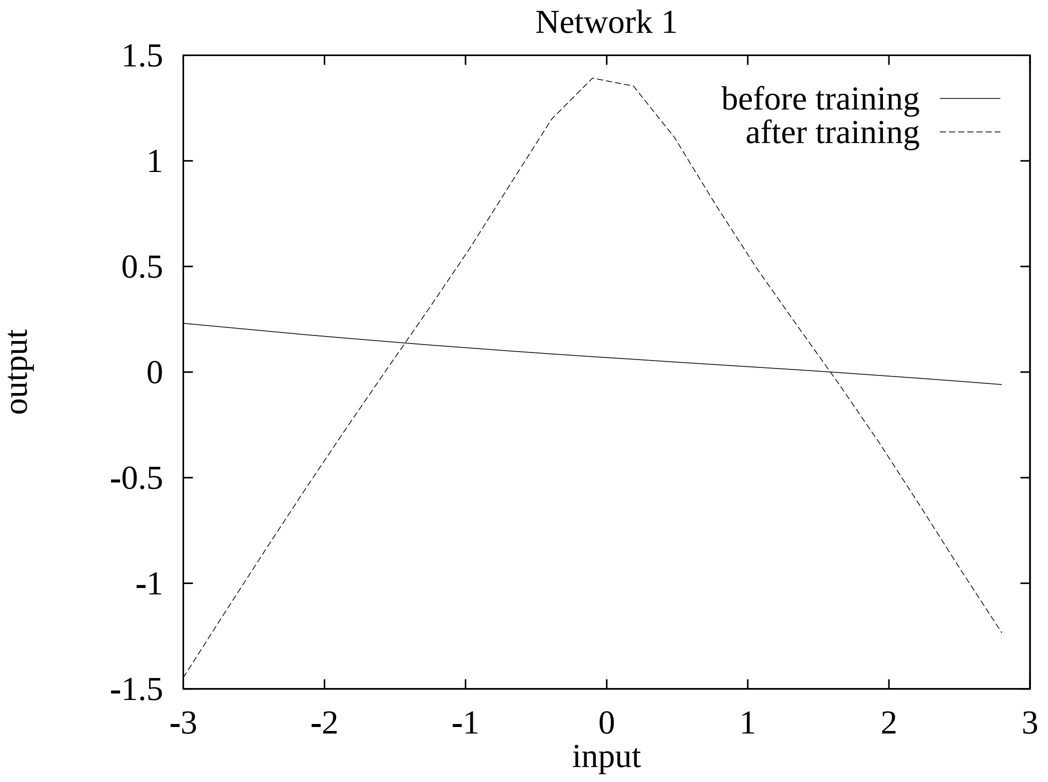


Figure 3.26: The mapping produced by the first of the two joined networks.

of network 2 is simply to make the optimization of network 1 more efficient. Network 2's outputs are not used explicitly. Network 1, the controller, can be trained without knowing explicitly the correct control actions for the sensory inputs.

The problem with this technique is that although network 2's outputs can be trained to emulate the plant, its gradients might not reflect the correct derivatives of the plant's output with respect to the plant's input. Since network 2 is only used for the effect of its gradients, the utility of this method in the general case is questionable.

3.11 Temporal differences and TD(λ)

Temporal difference methods for neural networks use the following TD(λ) weight update rule:

$$\boldsymbol{\delta}\mathbf{w} = \alpha(p(t+1) - p(t)) \sum_{k=1}^{t} \lambda^{t-k} \left.\frac{\partial p}{\partial \mathbf{w}}\right|_{k} \tag{3.64}$$

where $p(t)$ is a prediction of the long-term outcome at time t and α and λ are parameters of the update rule.

Tesauro has used TD(λ) to train a neural network to learn to play the game of backgammon (Tesauro 1991). The program plays to a high level, among the best in the world. It is the single most convincing demonstration of the utility of TD(λ) and the worth of reinforcement learning.

One advantage of TD(λ) in training network weights is that it gives an incremental update. At each time step, a weight update can be prescribed. This is not necessarily a reason for using the update rule before any others, however. For example, gradient descent, gradient descent with momentum and quickprop can also be used to perform an incremental update on-line.

Tesauro points out that the advantage of incremental updates may be of little value practically (Tesauro 1991). He supposes it is likely that if the application is tractable with TD(λ), it is also reasonable to store a sequence of reinforcements within a trial. This means that traditional algorithms might work better than TD(λ) at the cost of losing the incremental quality of TD(λ).

Tesauro's neural backgammon player played $300,000$ games during training. It is possible that traditional optimization techniques could find weights for the controller given such a large number of training examples.

3.12 Summary

This chapter has presented a review of feed-forward neural networks and some of the algorithms and techniques associated with them. Efficient training algorithms for neural networks have been described, useful in real-time environments.

Chapter 4

Continuous adaptive control

This chapter looks at some continuous adaptive controllers, in the form of neural networks.

Continuous state spaces present a challenge for dynamic programming control methods. Convergence is more difficult to prove. Nevertheless, dynamic programming methods have been used successfully in continuous state spaces (Anderson 1993, Bradtke 1993).

4.1 Neural network control

Werbos discusses a number of types of neural control (Werbos 1991). Six different ways to apply neural networks to control can be identified:

- Supervised control
- Direct inverse control
- Neural adaptive control
- Back-propagation through time
- Forward modelling
- Adaptive critic methods

These methods are outlined below. A more detailed exposition of neural networks for control is given by Hunt et al. (Hunt, Sbarbaro, Zbikowski and Gawthrop 1992).

4.1.1 Supervised control

Supervised control attempts to clone a pre-existing controller. The motivation for this kind of control rests in the assumption that the network is able to generalize from the training set. A neural network is given a training set of inputs and targets. The inputs are examples of readings from the plant's sensors, and the targets are control actions appropriate to those readings. The targets are generated from a pre-existing expert. Once the network is trained, it is expected to generalize so that presentation of a novel input vector results in a control output that approximates the output an expert would have given in the same situation.

A disadvantage of this approach is the requirement for a pre-existing controller. However, if computing control actions is a slow process, it might be worthwhile to train a network based on a training set from the expert, and use the neural network as a quicker look-table and generalizer.

The method is dependent on the network generalizing well, and so requires an appropriate training framework and network architecture. Generalization is discussed in section 3.2.1.

4.1.2 Direct inverse control

For some control problems, an inverse model of the plant may be learned directly. A training set can be generated from input/output data from the plant, using the outputs from the plant as inputs to the network, and the inputs to the plant as the corresponding targets for the network. Once the network is trained, the network maps plant outputs to the control actions that generate them. The network architecture and training schedule should be carefully designed for appropriate generalization on unseen inputs.

The use of inverse mappings for control is generally unsatisfactory, due to generalization errors that can occur between training examples. The problem with the method is demonstrated in figure 4.1. The output for the plant is well defined for each input. However, for a given demanded output, there are any number of inputs that could be used to produce it. The inverse model is a one-to-many mapping. A neural network will interpolate between the data, producing inappropriate control actions.

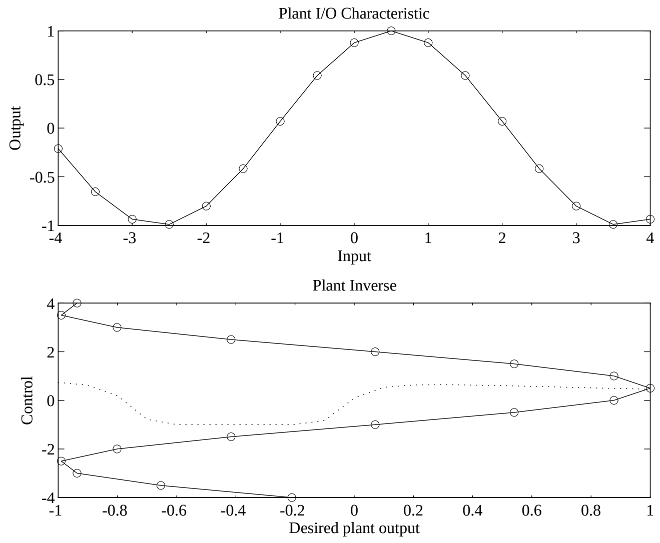


Figure 4.1: The direct inverse problem. The upper pane shows the input/output characteristic of the plant, sampled at the circles. The lower pane shows the inverse of the plant (solid line) and the interpolation of a 1-5-1 network trained with the samples (dotted line)

4.1.3 Neural adaptive control

Neural networks have been used as adaptive elements within conventional controllers (Tzirkel-Hancock 1992). This kind of use for neural networks is significant because stability can be proved in certain applications. By finding a suitable Lyapunov function, Tzirkel-Hancock proved the stability of an adaptive neural controller applied to a tracking control problem for an affine plant (Tzirkel-Hancock 1992). A rough outline of the scheme is given below.

A single-input, single-output affine plant is described by the following equations:

$$\dot{\mathbf{x}} = \mathbf{a}(\mathbf{x}) + \mathbf{b}(\mathbf{x})u \tag{4.1}$$

$$y = h(\mathbf{x}), \tag{4.2}$$

where u is the control input, y is the output, $\mathbf{x}$ is the state of the system, $\mathbf{a}$ and $\mathbf{b}$ are vector functions of the state $\mathbf{x}$ and h is an output function. The system is called affine because the control variable appears as a linear element in the plant dynamics. Mechanical robots are examples of affine systems.

An input/output relationship for an affine system exists, under certain conditions (Tzirkel-Hancock 1992), and has the form:

$$y^{(\gamma)} = f(\mathbf{x}) + g(\mathbf{x})u \tag{4.3}$$

where $f(\mathbf{x}) = L_a^{\gamma} h(\mathbf{x})$ and $g(\mathbf{x}) = L_b L_a^{\gamma-1} h(\mathbf{x})$. L is the Lie derivative operator and γ is the strong relative degree of the system.

If $\mathbf{a}(\mathbf{x})$, $\mathbf{b}(\mathbf{x})$ and $h(\mathbf{x})$ were known, then $f(\mathbf{x})$ and $g(\mathbf{x})$ could be found. Applying

$$u = \frac{-f(\mathbf{x}) + v}{g(\mathbf{x})}, \tag{4.4}$$

where v is another control law, linearizes the plant. A standard linear controller could then be used to control the system.

The rôle of neural networks in this scheme is to approximate the functions f and g for systems with unknown dynamics. The approximations are given by

$$\hat{f}(x) = f_0(x) + \boldsymbol{\theta}_f^T \mathbf{w}_f(x) \tag{4.5}$$

$$\hat{g}(x) = g_0(x) + \boldsymbol{\theta}_g^T \mathbf{w}_g(x) \tag{4.6}$$

where f_0 and g_0 are prior knowledge about the functions f and g, $\mathbf{w}_f(x)$ and $\mathbf{w}_g(x)$ are fixed weight parameters for the initial layer of each network, and $\boldsymbol{\theta}_f$ and $\boldsymbol{\theta}_g$ are network parameters for the second layer of each network, adapted during learning. The tracking error and its derivatives are transformed into an error suitable for adapting the networks' parameters.

4.1.4 Back-propagation through time

Nguyen and Widrow used two neural networks in a control scheme that learned to reverse a simulated articulated lorry into a loading bay (Nguyen and Widrow 1989, Nguyen and Widrow 1990). Figure 4.2 shows the task. One network formed a model of the lorry, another mapped the lorry's state into a control action (the angle of steer of the cab).

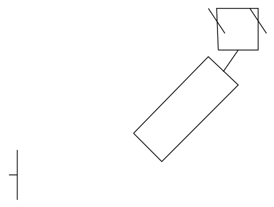


Figure 4.2: The articulated lorry and loading bay.

The networks were trained using a technique known as back-propagation through time (BTT). A plant identification procedure trained a model of the lorry using a training set comprising states and actions as inputs, and next-step states as targets. Then several trial dockings were attempted. For each step during a trial, the state of the lorry was passed to a controller network that produced a control action, subsequently applied to the plant. The state of the plant and the controller network were stored at each step. At the end of a trial, the activations were recalled to form a long network made up of one layer for each time step. The error between the final lorry state and the demanded lorry state was then propagated backwards through the network, adjusting the control network weights by gradient descent. The demanded lorry state was the state of the lorry when parked in the dock. Only the weights of the controller were adjusted; the weights of the model were left unchanged. Figure 4.3 shows the idea of the layers of the extended network.

Nguyen and Widrow used 25 hidden units in their model of the lorry scheme, and 25 hidden units for the controller network. They pointed out that at the time there was no theory to guide their choice of architecture other than rule of thumb and experiment. Later, Jenkins et al. showed

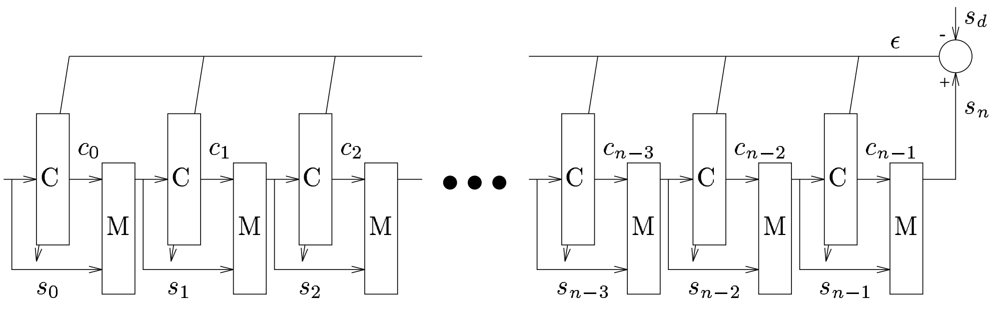


Figure 4.3: Back-propagation through time. C is a control network and M a neural network model. s_n is the lorry's state at step n, s_d the demand state at the dock, and c_n the control action at step n.

that the lorry reversing task can be completed with much smaller networks (Jenkins and Yuhas 1992). A control network with three inputs (cab angle, trailer angle and the vertical deviation from the centre of the dock) and two hidden units feeding into the control output (the angle of steer) is sufficient. Although Jenkins et al.'s approach of decomposing the problem to discover a simpler network architecture simplifies the networks, the other side of the argument is that more powerful networks can avoid the need to decompose the problem, essential if the problem is intractable.

Back-propagation through time requires a large amount of memory, since the network activations need to be saved over the whole of a learning sequence. It is only suited to tasks that are trial-based.

4.1.5 Forward modelling

A forward model of the plant avoids the generalization problem that can occur with an inverse model. Forward modelling relies on the idea of training without targets by joining networks, discussed in section 3.10.

Jordan and Jacobs used a neural network to form a forward model for the inverted pendulum task (Jordan and Jacobs 1990). A forward model translates proximal coordinates to distal coordinates. A proximal coordinate is 'near to' the controller, and is in the space of control actions accepted by the plant. A distal coordinate is an abstracted measure of the task, and cannot be directly set by the controller. A controller must control, through the proximal coordinates, the distal coordinate of the task.

A distal target for a pendulum balancing regulator is the length of time the pendulum is balanced. A controller must be given a proximal target, a target for its outputs, to satisfy the demand on the distal target. The mapping between distal and proximal target values is a way of expressing the problem of learning control.

The input vector $\mathbf{v}$ for Jordan and Jacob's forward model was a concatenation of a modified version of the state, and the control action from the controller:

$$\mathbf{v} = \begin{bmatrix} |x| \\ \text{sign}(x) \\ |\dot{x}| \\ \text{sign}(\dot{x}) \\ |\theta| \\ \text{sign}(\theta) \\ \left|\dot{\theta}\right| \\ \text{sign}\left(\dot{\theta}\right) \\ a \end{bmatrix}$$

where x is the trolley position, θ the pendulum's angle from the vertical and a the control action. The output of the forward model was an estimate of the reciprocal of the time to the pendulum's next collapse. Time was measured in discrete steps. The output was conveniently scaled between zero and one.

The forward model was trained using a temporal difference technique. The idea was to keep the model consistent with its predictions, and to ground the chain of predictions when a failure occurred. The model was trained at each time step. Let the output of the model at step i be given by $p(i)$. Then the estimated time to collapse at time i is given by $p^{-1}(i)$. An estimate for the time to collapse at the previous step is $p^{-1}(i) + 1$. A target value $t(i)$ for the prediction at step i is therefore given by:

$$t(i) = \frac{1}{p^{-1}(i+1) + 1}. \tag{4.7}$$

Training a forward model using targets given by equation 4.7 produces self-consistent predictions. The predictions are grounded when the pendulum collapses, since if the pendulum collapses at step k, the target for the prediction at $k-1$ is 1. A supervised learning scheme was used to train the forward model on the targets at each time step.

The controller network's input was the complete state information x, $\dot{x}$, θ and $\dot{\theta}$. There were no hidden units in the controller, so the control scheme was a linear state feedback scheme.

The controller could not be trained using supervised learning, because there were no explicit targets. This deficiency was met by the forward model. The forward model was not used to predict the time to failure. It was trained to predict time to failure, and then used to supply gradient information to train the neural network controller.

If output of the forward model were zero, the estimated time to collapse would be infinite. The difference between the output of the forward model and this ideal was the error e_c of the controller in the distal coordinate. The back-propagation algorithm was used to find the partial derivatives $\frac{\partial e_c}{\partial y}$ of e_c with respect to the activations of the nodes in the forward model. The value of $\frac{\partial e_c}{\partial y}$ at the action input to the forward model gave the gradient of the distal error with respect to the control action. The gradient was used to seed the back-propagation applied to the controller, and the controller was trained using gradient descent.

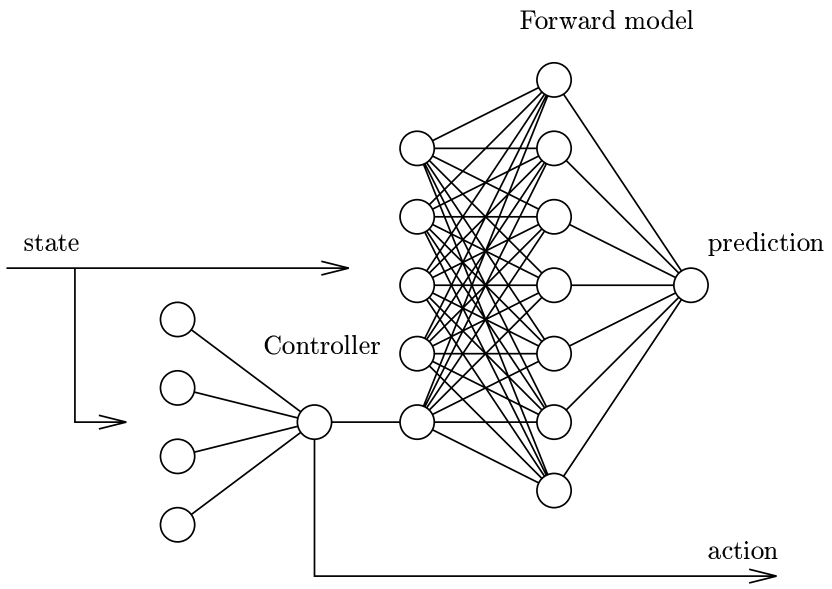


Figure 4.4: Forward modelling for the inverted pendulum. The state is fed into the controller and the forward model. The controller produces a control action, and the forward model makes a prediction of its the performance.

Jordan and Jacobs applied their forward modelling technique to the same plant as Barto et al. (Barto et al. 1983), save for neglecting the friction terms. They reported convergence after between 5000 and 30000 collapses.

Although the scheme worked, it required a large amount of training relative to Barto et al.'s learning controller (Barto et al. 1983). Relative to Barto et al.'s work, the technique required less prior knowledge of the task, and might be expected to need more experience to converge. However, Brody's analysis shows that the slow convergence is because the network is trained in a restrictive manifold of the state and action space, since the action is a function of the state (Brody 1992). This means that gradients off the manifold are arbitrary and result in poor convergence, or stop learning altogether. A discussion of Brody's work follows.

4.1.6 Brody's controller

Brody improves on Jordan and Jacobs' scheme by introducing a third network, a model that predicts the next state from the current state and action (Brody 1992). The extra network is shown in figure 4.5. The model is pre-trained using data from the whole of state and action space. The model is then frozen, and a control network and an evaluation network are then trained by back-propagating through the world model. The evaluation network maps the state of the plant into a measure of value for that state in terms of the control goals, and the world model and evaluation network act together like the forward model of Jordan and Jacobs (Jordan and Jacobs 1990).

Brody justifies his three-model approach by highlighting a problem of redundancy in training a forward model. His point rests on the fact that the control signal is a function of the state of the system. The forward model's exemplars have their inputs in a manifold of the state and action space defined by the mapping from state to action space. The model is free to allow any mapping it finds convenient in the area off this manifold. The gradients off the manifold will be arbitrary. Therefore, a controller that uses these gradients to adapt its output can perform poorly.

Although exploration around a manifold defined by a policy will occur as the policy changes, Brody describes how the gradient of the evaluation function may become small or zero during the learning phase. This will slow down, or stop, convergence to a suitable control policy.

One solution to the problems of forward modelling is to add noise to the control signal. This was the approach taken by Jordan and Jacobs (Jordan and Jacobs 1990). However, the task of the forward model is now more difficult, due to the need to cope with noise in the control function. Predictions of future performance become less certain.

Including Brody's world model avoids the arbitrary gradients off the control function's manifold, because the model is pre-trained with data from the whole of state and action space.

Clearly, Brody's scheme is advantageous if data from the whole of state and control space can be generated, since it makes the most of this prior knowledge. However, the need for such exemplars can also be seen as a drawback, as Werbos suggests (Werbos 1990).

Brody reports good results with his three-network scheme, converging to a working regulator for the same task that Jordan and Jacobs attempted in under 800 trials for 8 out of 11 experiments, most of those in under 300 trials (Brody 1992). The other three experiments were reported to have failed to converge. The results, although within the range that might allow the technique to be applied to hardware, unfortunately rely on off-line training of the world model with a suitably rich data set.

4.1.7 Actor/critic methods

The difference between adaptive critic methods and forward modelling is that the controller is not trained by differentiating the forward model, but by explicitly optimizing the critic's output over the control actions.

Anderson used two-layer neural networks as actor (controller) and critic (action quality evaluator) in a reinforcement learning framework (Anderson 1989). In the earlier ASE/ACE experiments of Barto et al. (Barto et al. 1983), the ASE was the actor and the ACE the critic. Anderson chose the inverted pendulum problem of Barto et al., without friction, to test his controller. He reported convergence after 6000 collapses of the pendulum (Anderson 1989).

The advantage of Anderson's networks over the ASE/ACE controller is that the need for a carefully prepared discretization is avoided — instead, the nodes of the network are fed the plant's

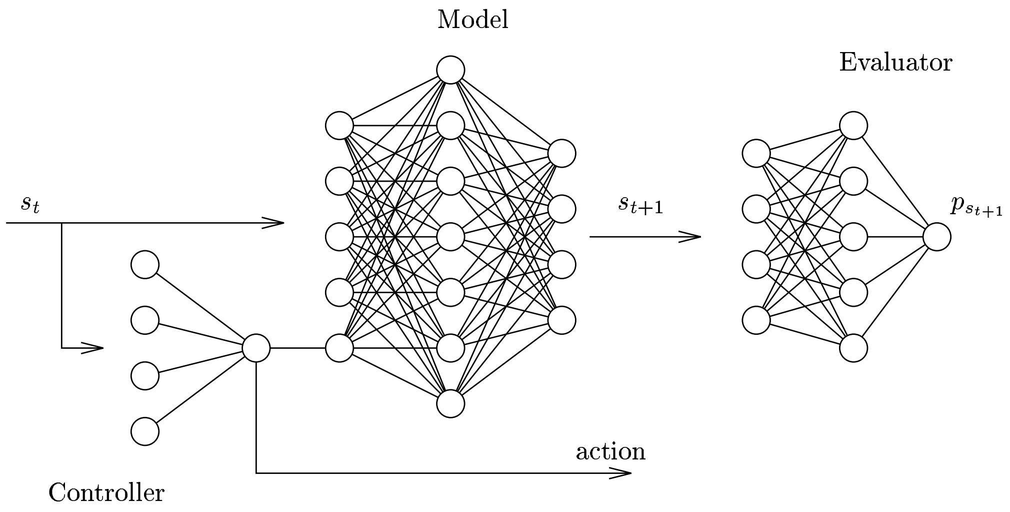


Figure 4.5: Brody's control scheme. The state at time t, s_t, is mapped into an action by the controller. The model predicts the next state s_{t+1} from the current state and action. The evaluation network predicts the quality $p_{s_{t+1}}$ of the next state. The difference between the performance and the desired performance can be back-propagated through the evaluator and the model to the controller.

state directly. A cost of this flexibility is in increased training time. Anderson also needed a suitable training schedule so the networks would gain enough experience of the plant dynamics. He reset the pendulum to random states after a collapse. This ensured the system learned to perform correctly for states away from the origin.

Anderson has also used multi-layer perceptrons to form Watkins' Q-function for the inverted pendulum (Anderson 1993). He describes a technique he calls 'hidden-unit restarting' to increase the learning speed, producing a factor of two improvement, training after 3000 collapses with hidden-unit restarting against 6000 without. The technique is designed to re-allocate hidden units that have become inactive during earlier learning.

4.2 Linear quadratic regulation and Q-learning

Bradtke has applied reinforcement learning to a linear quadratic regulation (LQR) problem (Bradtke 1993). His motivation is to move dynamic programming methods into continuous state and action spaces.

He considers a linear plant of the form:

$$\begin{aligned}
\mathbf{x}_{t+1} &= \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t) && (4.8)\\
&= \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t && (4.9)\\
\mathbf{u}_t &= \mathbf{U}\mathbf{x}_t && (4.10)
\end{aligned}$$

where $\mathbf{x}_t$ is the state of the system at time-step t, $\mathbf{u}_t$ is the control action, $\mathbf{A}$ and $\mathbf{B}$ are matrices that define the plant dynamics, and $\mathbf{U}$ is a linear state feedback controller.

Controlling the plant incurs a cost $r(\mathbf{x}_t, \mathbf{u}_t)$, which is a quadratic function of the state and control action:

$$r(\mathbf{x}, \mathbf{u}) = \mathbf{x}^T\mathbf{E}\mathbf{x} + \mathbf{u}^T\mathbf{F}\mathbf{u} \tag{4.11}$$

where $\mathbf{E} = \mathbf{E}^T > 0$ and $\mathbf{F} = \mathbf{F}^T > 0$.

Applying a control policy $\mathbf{U}$ over a period of time incurs a sequence of costs. The sequence is a function of the policy and the initial state. The value of a state $\mathbf{x}_t$, denoted by $v_{\mathbf{U}}(\mathbf{x}_t)$, is given by:

$$v_{\mathbf{U}}(\mathbf{x}_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{4.12}$$

where $0 \leq \gamma < 1$ is a discount factor. For the LQR problem, the value function $v_{\mathbf{U}}$ can be expressed as a quadratic function of the states:

$$v_{\mathbf{U}}\mathbf{x} = \mathbf{x}^T \mathbf{C}_{\mathbf{U}} \mathbf{x} \tag{4.13}$$

where $\mathbf{C}_{\mathbf{U}} > 0$ is a cost matrix for policy $\mathbf{U}$.

The optimal policy, $\mathbf{U}^*$, is the policy which minimizes the discounted sum of costs from each state.

Watkins' Q-function for this system is given by:

$$Q_{\mathbf{U}}(\mathbf{x}, \mathbf{u}) = r(\mathbf{x}, \mathbf{u}) + \gamma v_{\mathbf{U}}(\mathbf{f}(\mathbf{x}, \mathbf{u})) \tag{4.14}$$

where $\mathbf{u} = \mathbf{U}\mathbf{x}$. Q-learning is about learning this function. Bradtke makes a distinction between two forms of Q-learning. The first he calls 'optimizing Q-learning', which learns the Q-function of the optimal policy $\mathbf{U}^*$ directly. Its update rule is given by:

$$Q_{t+1}(\mathbf{x}_t, \mathbf{u}_t) = Q_t(\mathbf{x}_t, \mathbf{u}_t) + \alpha \left[r(\mathbf{x}_t, \mathbf{u}_t) + \gamma \min_{\mathbf{a}} (Q_t(\mathbf{x}_{t+1}, \mathbf{a})) - Q_t(\mathbf{x}_t, \mathbf{u}_t) \right] \tag{4.15}$$

where Q_{t+1} is the next approximation to the Q-function of the optimal policy $\mathbf{U}^*$. This is the form of Q-learning introduced by Watkins (Watkins 1989).

Bradtke refers to the second form of Q-learning as 'policy-based Q-learning'. This form of Q-learning attempts to find the Q-function for a particular policy $\mathbf{U}$. The update rule for this form of Q-learning is given by:

$$Q_{t+1}(\mathbf{x}_t, \mathbf{u}_t) = Q_t(\mathbf{x}_t, \mathbf{u}_t) + \alpha \left[r(\mathbf{x}_t, \mathbf{u}_t) + \gamma Q_t(\mathbf{x}_{t+1}, \mathbf{u}_{t+1}) - Q_t(\mathbf{x}_t, \mathbf{u}_t) \right] \tag{4.16}$$

where $\mathbf{u}_{t+1} = \mathbf{U}\mathbf{x}_{t+1}$.

Bradtke finds the Q-function for a particular policy over a series of trials. The policy is then improved, based on the estimate of the Q-function for the current policy. This involves a minimization over the control actions, which, for the LQR problem, can be solved analytically. The result is an update rule for the control matrix $\mathbf{U}$:

$$\mathbf{U}_{k+1} = -\mathbf{H}_{\mathbf{U}}^{22\,-1} \mathbf{H}_{\mathbf{U}}^{21} \tag{4.17}$$

where

$$\mathbf{H}_{\mathbf{U}} = \begin{bmatrix} \mathbf{H}_{\mathbf{U}}^{11} & \mathbf{H}_{\mathbf{U}}^{12} \\ \mathbf{H}_{\mathbf{U}}^{21} & \mathbf{H}_{\mathbf{U}}^{22} \end{bmatrix} \tag{4.18}$$

and

$$Q_{\mathbf{U}}(\mathbf{x}, \mathbf{u}) = \left[\mathbf{x}^T \mathbf{u}^T \right] \mathbf{H}_{\mathbf{U}} \begin{bmatrix} \mathbf{x} \\ \mathbf{u} \end{bmatrix}. \tag{4.19}$$

Bradtke estimates the matrix $\mathbf{H}_{\mathbf{U}}$ using recursive least squares.

Bradtke claims the sequence of policies generated by equation 4.17 to be provably convergent to the optimal policy.

The second Q-learning method Bradtke investigates uses optimizing Q-learning. The best policy after each update of the optimizing Q-function is found. This approach is found to be only locally convergent, that is, it will only converge to the optimal policy for parameters sufficiently close to the optimal solution. Bradtke highlights its problems by finding another Q-function that the optimizing Q-learning approach, applied to his LQR problem, also converges to.

The behaviour of the optimizing Q-learning method on the LQR problem suggests that the convergence results developed for finite state systems cannot be expected to apply to continuous state systems in general.

4.3 Conclusions

For a control scheme to be useful, it must cope with the demands of real hardware. Although the neural network solutions are more adaptive than the discrete approaches using 'boxes', they so far require too many training examples for their implementation on real hardware to be feasible. The adaptive heuristic critic approach learns quickly, in terms of the number of collapses of the pendulum before convergence, but requires some careful design in the controller specific to the task, nullifying part of the benefit of using a learning controller. Forward modelling with neural networks is more adaptive, requiring less task-specific knowledge. Unfortunately, the training time for this scheme is large, requiring many collapses before enough experience is gained for reasonable performance. In practice, the hardware is likely to wear out before convergence.

There is no robust, efficient learning controller for the continuous domain. Chapter 6 will, by casting the learning control problem as an optimization problem, attempt to find an efficient method for learning control in the continuous domain using the inverted pendulum as a benchmark problem.

Chapter 5

The inverted pendulum in hardware

This chapter describes an implementation of Barto et al.'s ASE/ACE controller (Barto et al. 1983) on a real rig, operating in real time. The demands of physical equipment required a return to the discrete methods of chapter 2, which learned the task, in simulation, in some hundreds of collapses.

5.1 The hardware

The testbed for the learning algorithm was a control rig designed for an undergraduate control course at Cambridge University Engineering Department. Details of the rig are given in appendix C.

The rig was based around a track 1 m in length, along which a carriage was free to move. The carriage was attached by wires to a torque motor mounted on one end of the track. A short, light aluminium rod was free to rotate about a pivot on the carriage. On the end of the rod was a heavy brass weight.

The rig was controlled by a C++ program on an Inmos T800 Transputer, connected to the apparatus via a custom interface designed and built within Cambridge University Engineering Department. The interface board carried four analogue to digital converters (ADCs) for the sensory data and a digital to analogue converter (DAC) for the control action.

Sensory information from the rig comprised angular position and velocity of the pendulum and linear position and velocity of the carriage (i.e. full state information). These were measured using potentiometers and tachometers on both the axis of the torque motor and the pendulum's pivot. The carriage position potentiometer was geared, since the torque motor took several rotations to move the carriage from one end of the track to the other.

5.2 Modifications to the original ASE/ACE controller

Modifications to the original ASE/ACE controller of Barto et al. were necessary because the real apparatus had different dimensions and masses to the simulation. Most significant was the shorter working track length (0.6 m against 4.8 m). The differences meant the original ASE/ACE controller failed to keep the real pendulum upright for more than about a second.

The controller successfully balanced the simulation using the masses of the physical apparatus and a 4.8 m track. However, reducing the simulated track length to 0.6 m defeated the controller, producing poor performance similar to that seen on the real rig. Comparing the real rig with the simulation suggested that the simulation was a good approximation. However, this is not necessarily important, since to test a learning controller the simulation need only be a suitably difficult control problem.

5.2.1 Assessing control feasibility

To determine the feasibility of controlling the pendulum with the available equipment, a number of steps were taken, helped by the existence of a linear state feedback control expression (see appendix E).

Linear state feedback

First it was verified that the T800 and interface were fast and accurate enough to balance the pendulum using simple state feedback. Four control gains were applied to the elements of the system's state to yield a proportional control torque from the torque motor. The result was a robust solution to the pendulum-balancing problem. It was possible to place a coffee mug on top of the pendulum and still observe stability.

The form of the linear state feedback expression is:

$$u = k_x x + k_{\dot{x}} \dot{x} + k_\theta \theta + k_{\dot{\theta}} \dot{\theta} \tag{5.1}$$

where u is the control action, x is the carriage position from the centre of the track, $\dot{x}$ is the carriage velocity, θ is the angle of the pendulum from the vertical and $\dot{\theta}$ is the pendulum angular velocity. The subscripted k's are the control gains.

Bang-bang control

To verify the viability of a bang-bang control scheme (the kind of control offered by the ASE/ACE controller), the linear state feedback control action from above was thresholded to give an output of fixed magnitude but variable sign. Stable control was observed, although not as robust as in the previous proportional case. Both the magnitude of the bang-bang control action and the sampling frequency were altered to discover the ranges of these parameters that led to stability. A sample frequency of 50 Hz and a control action magnitude of 0.08 Nm at the torque motor worked well, and were used for all the subsequent controllers.

The success of the bang-bang scheme verified the existence of a working control policy for a high-resolution boxes scheme. That is, a boxes scheme that partitioned state space to the resolution of the ADCs should have worked. Unfortunately, memory and processor speed preclude so many boxes. Thus the next step was to find a partitioning of state space with fewer boxes that still offered reasonable performance.

5.2.2 Finding a partitioning scheme

Successful partitioning schemes were found by trial and error. The search was initially fruitless, but was later successful when helped by two heuristic devices. The first avoided the need to wait for the controller to attempt to learn a policy before a partitioning could be rejected or accepted. The second helped to choose sensible partitionings to test.

Testing a candidate partitioning of the state space was speeded up by pre-filling each box of the partitioning with the control action that the bang-bang scheme would take at a candidate point in each box. For boxes of finite size, their centres were chosen, and for semi-infinite boxes (those at the edges of the partitioning), points were selected a suitable distance away from their inner edge.

Since each dimension of the state space could be dissected in many ways, the number of reasonable partitionings was large, and it was soon realized that a lot of them failed to work. A heuristic based upon 'pseudo switching lines', detailed below, helped to choose candidate partitioning schemes, and seemed to work well.

Pseudo switching lines

A switching line is a line through two-dimensional state space (or a hyperplane though multi-dimensional state space) which divides regions for which the bang-bang control actions are different. By setting the terms of any two state variables to zero in the state feedback equation (5.1), the relationship between the remaining state variables defines a rough approximation to a switching line, a 'pseudo switching line'.

It was reasoned that one way of choosing partitioning schemes would be relative to these pseudo switching lines. Two possibilities presented themselves: either the boxes themselves could be placed corner-to-corner to approximate the lines, like a string of diamonds, or the edges of the boxes could be set like a staircase through the lines. The two patterns are shown diagrammatically in figure 5.1.

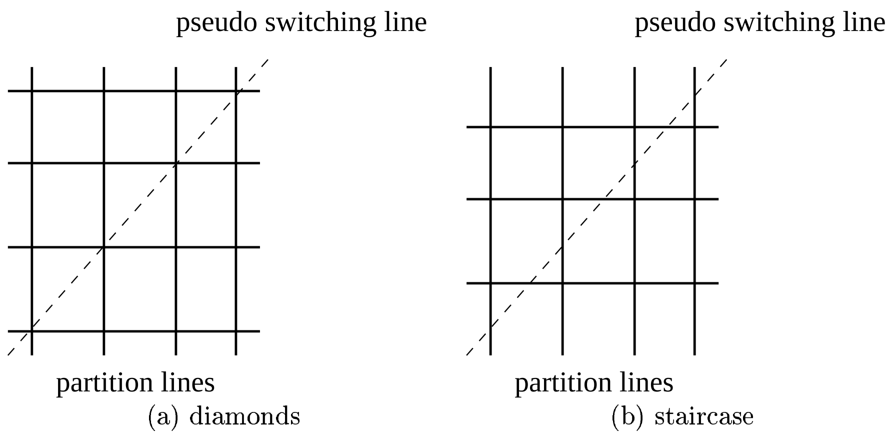


Figure 5.1: Partitions relative to pseudo switching lines

Three partitioning schemes were studied in detail. The first, with 750 boxes, used the staircase pattern. The second had 1296 boxes in a diamond pattern. The last scheme used 864 boxes, based on the diamond pattern but with two fewer partitions on the carriage position axis, the remaining carriage position partitions spread away from the centre.

The pseudo switching line heuristic is weak. Nevertheless, with it a number of partitioning schemes were eventually found to work with a pre-filled control policy. Knowing that for each of these partition schemes, at least one stationary control policy existed (i.e. the policy defined by the pre-filling procedure), it was expected that the ASE/ACE policy learning algorithm would find a control policy to balance the pendulum.

5.2.3 Processing speed

Real apparatus requires real time control. A feature of this work is that both control and learning were in real time. The sample frequency, the rate at which the controller was updated and the rate at which new control actions were issued were all 50 Hz. The earlier ASE/ACE experiments simulated the same rates (Barto et al. 1983).

Because the ASE/ACE algorithm required only coarse resolution of the system's state, real-time implementation was straightforward. The boxes scheme only required a small number of comparison operations between elements of the state vector and partitions of the state space axes. A control algorithm that needed to know distances between the plant's state and some datum points in state space would be more computationally demanding.

Too many boxes

The transputer was only able to process around 350 boxes within a sampling interval. Although the sampling could be reduced a little, runs with the pre-filled boxes showed that 50 Hz was better than, say, 45 Hz. The computational demands had to be slackened.

Most of the processing was needed to update the traces on the links between the ACE, ASE and decoder. Since the trace variables quickly become small, it was decided to operate the weight update scheme on only the last 200 boxes visited by using a circular queue. If a box was visited more than once in the last 200 samples, it would be entered in the queue more than once. A single pass through the queue would therefore update the parameters of that box more than once. Although the trace would decay faster, the reinforcement would be applied more often. These effects seemed to balance; a more involved rule was not deemed necessary.

An alternative solution would have been to run the algorithm in parallel, an opportunity that could still be realized using more than one transputer. However, one transputer and the adjustments in the algorithm described here were seen to give suitable results. Since the purpose of the work was to verify that the pendulum could be balanced, the parallelization project was seen as work for the future.

Control action delay

Delay between measuring the state of the pendulum system and delivering a control action might have had an effect on the ability of the controller to balance the pendulum. In order to gauge how significant the effect might be, a delay was built into a pre-filled controller. A delay of one sample period between measuring the state and applying the control action was seen to have little effect on the stability of the control. The delay in the learning controller could not be greater than a sample period, and so it was expected that the effect of any delay would not be serious. Significantly poorer performance was noticed as the delay was increased beyond one sample period.

The performance of the pre-filled controller with single sample delay was expected to be bettered both because the delay would be less than a full sample period, and because the learning controller could discover a control policy for the best performance (within the constraints of the state space partitioning). Once released from the fixed policy created for the pre-filled test run, the controller would be expected to find, if one existed, a modified policy that performed better, reducing the effect of delay.

5.3 Training procedure

Data were collected from a series of experimental runs. A run consisted of a long series of trial balances, and was either terminated after a fixed number of samples, or if performance was poor, after 500 trials. Before a run, the ASE was initialised with random weights, the ACE was initialised with zero weights, and the sensors were calibrated to compensate for drift in their values. (The sensors were not of high quality. Position, for example, was measured using geared potentiometers and not optical tracks.)

5.3.1 A control trial

A trial was started with the pendulum stationary, hanging down in the middle of the track. The pendulum was then flicked upright by a strong kick from the torque motor, which lifted the pendulum high enough for a linear state feedback controller to catch it upright and bring it smoothly back to the centre of the track. When the pendulum was upright, stationary and close enough to the centre of the track, the ASE/ACE controller was applied, and learning began.

During the ASE/ACE control period, the state of the system was sampled and fed through the decoder to the ASE, which returned an action that was immediately applied to the rig. In the remainder of the sample period, a reinforcement of zero was fed to the ACE, which consequently supplied an internal reinforcement (normally non-zero) to the ASE. All the learning thus took place within a sample period. Finally the processor waited until the next state sample was ready from the ADCs.

ASE/ACE control continued until the system had fallen out of its pre-set bounds (the carriage more than 0.3 m from the centre of the track, or the pendulum more than 12° from the vertical). At this point a park program took control of the pendulum, which moved it to the centre of the rig like a critically damped crane. With the pendulum at rest a reinforcement signal of -1 was sent to the ACE, and its subsequent output then passed on to the ASE. Although this stage could have been performed within a sampling period, due to hardware constraints the next stage could not: the output of trial data to the host system for logging. Once the data was transferred, the pendulum was flicked upright once again, and another trial started.

5.3.2 Reinforcement and trials

Barto et al.'s procedure (Barto et al. 1983) was not followed when the pendulum collapsed. Barto et al. had no box that corresponded to the state of a collapsed pendulum. The external reinforcement was always zero within any box. In this implementation, however, the learning scheme was changed to treat the reinforcement signal more generally. Not all tasks can be assumed to break down into trials, and reinforcement is not always discrete. State space partitions can be mis-aligned with reinforcement boundaries, so a box in state space might experience different values of external reinforcement over time.

The partitioning schemes presented here covered all of state space, and did not align with the out-of-bounds boundary in state space. The success of the algorithm under this distortion shows the algorithm's robust nature. Faster learning is expected with suitable alignment. Also, alignment is reasonable to expect since the reinforcement function is known before learning as the goal-directed specification of the control requirements. The idea of aligning partitions with the reinforcement boundary is shown in figure 5.2

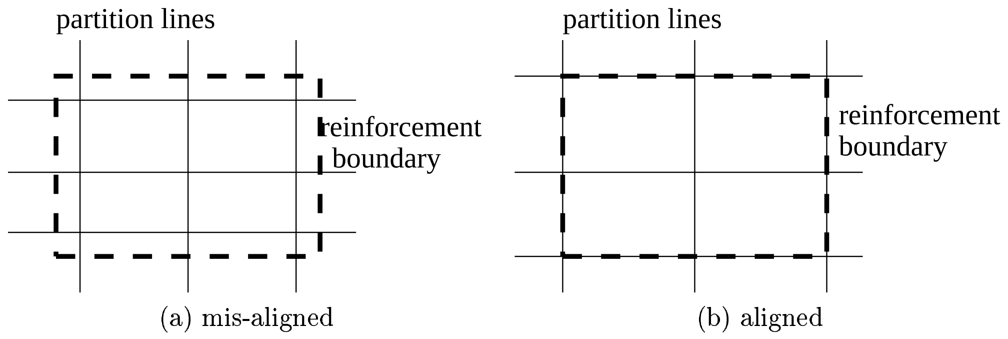


Figure 5.2: Reinforcement boundary and partition lines

5.4 Results

The performance of the controller was seen to improve as the time between collapses (the trial length) increased during learning. The best performance, a trial of 2100 seconds (35 minutes), was observed using the 864 box decoder after 255 collapses and 5043 seconds (84 minutes) of training time. However, the next longest trial was only half that length. These unusual runs are not seen on the plots below, since the smoothing procedure removes them to show the general trend. In comparison, a random policy would lead to the pendulum falling out of bounds after around 10 samples (0.2 seconds).

5.4.1 Learning rate

It was often noted that in simulation the controller would improve its performance dramatically over a short period of time. The same kind of behaviour was seen on the real rig. This is reflected in the roughly linear appearance of the performance plots on log scales.

Runs which started with longer periods of poor performance would lead to better results in the medium term, whereas runs which showed quicker convergence to reasonable performance took much longer to achieve better results. This observation might be explained by a kind of persistent excitation argument: poor initial controllers learn about their environment more quickly than controllers that make mistakes less often.

5.4.2 Variations in trial lengths

The results show a large variation in trial lengths, both within and between runs using the same controller. This variation is partly a function of the bang-bang control used, as can be seen with reference to figure 5.3, which shows runs for which learning was turned off after 200 collapses. In comparison to the other results where learning was not stopped, the figure also shows the positive effect of continued learning.

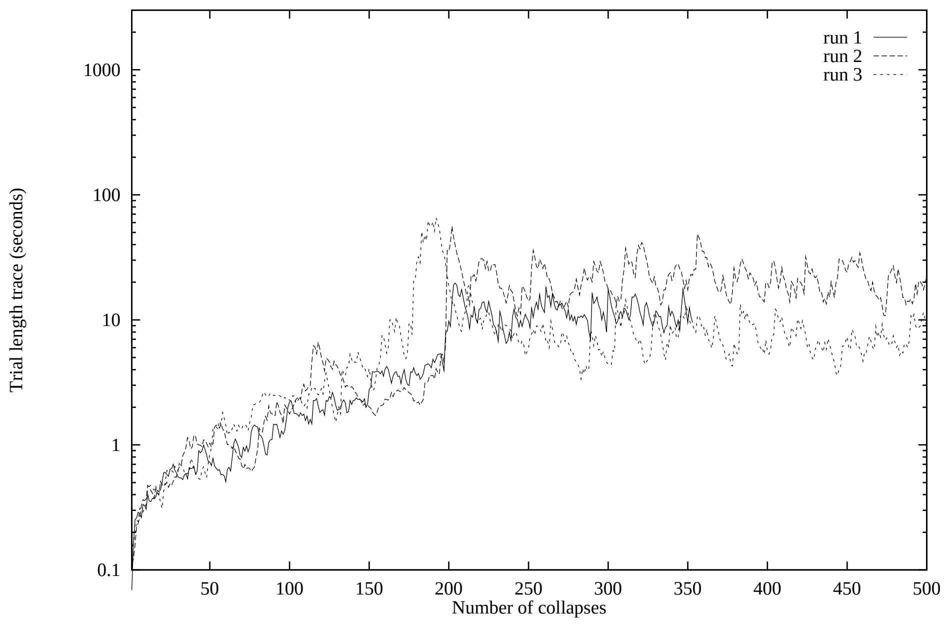


Figure 5.3: Three runs using the 864 box controller, with learning stopped after 200 collapses

Two of the runs in figure 5.3 show a poorer performance after 200 collapses (when learning was turned off) than that just prior to 200 collapses. This is because the ASE changes policy as it learns. Such changes may not always be for the best, and thus the performance can fall. With learning stopped, the policy selected just before learning was switched off becomes are permanent feature, along with its possibly poorer performance.

The other plots (where learning was not stopped) indicate a slightly larger variation in trial length as performance generally improves. An explanation of this might be that the number of control policies is large, but only a few perform well. Early in the learning phase, the controller changes between several similar policies, but since each is of poor performance, the variation in trial length is small. After learning has progressed, although some of the policies the controller invokes will be of much higher quality, poor policies may still linger. The general performance is better, but the variation is larger.

Randomization of the ASE's initial weights affected the initial conditions of a learning run, which helps to account for differences between runs. However, similar differences were also seen between runs starting with identical weights. Sensor drift and noise are therefore assumed to account for most of the variation between runs.

5.4.3 Smoothing the graphs

Graphs of the raw durations show a lot of variation between the lengths of trials, even after learning for some time. Thus in order to better discern a trend in the data, and to make a composite plot of several runs comprehensible, a filtered version of the trial lengths was plotted. The graphs show the value of a trace variable of trial length against the number of collapses. The trace variable t is defined as follows:

$$t(n) = (1 - \tau)t(n - 1) + \tau d(n) \tag{5.2}$$

where $d(n)$ is the length of the n'th trial. τ was set to 0.2 for all the plots. This exponential weighting of the results avoids the discontinuity of a binning procedure. The difference the trace makes may be seen in figure 5.4.

The results require a serious statistical analysis. Such an analysis would be of value to judge the comments made earlier on the controller's performance, and variations in that performance. The opportunity to carry out such analysis is left to the future. The results given here are expected to convey (1) that reasonable performance was achieved with the controller, and (2) that the configuration of the decoder (the state space partitioning) had an impact on that performance.

5.4.4 Changing the decoder specifications

Figure 5.5 shows the results of applying almost the same partitioning as the original simulation work. Because of the different track length, the carriage position partitions were moved to keep the same ratio with carriage position bounds (4.8 m in simulation, 0.6 m for the real apparatus). Reasonable performance was not observed with this partitioning, nor for any of several alternative partitioning schemes tried using 162 boxes.

Figure 5.6 is the result of a boxes scheme that attempted to re-create the rough switching lines outlined in section 5.2.2 using box edges in a staircase pattern.

Figure 5.7 shows the results of three runs with a 1296 box controller. This box scheme covered the pseudo switching lines in a diamond pattern.

Figure 5.8 shows three 400,000 sample runs using a decoder with 864 boxes. This decoder is similar to the 1296 decoder, but uses fewer boxes over the carriage position dimension (giving less resolution but more generalization). The lines in the plots are of different lengths because the 400,000 samples were distributed over the trials differently for each run.

The 750, 1296 and 864 decoders all led to successful learning and satisfactory control policies.

5.4.5 Comparison with the original ASE/ACE work

It is not possible to directly compare these results with the earlier results of Barto et al. (Barto et al. 1983) because of the difference in the plant parameters. Other differences include the reduced number of link weight updates to decrease the computational load, and more significantly the divergence in the treatment of the reinforcement signal through the mis-alignment of the state partitions and the reinforcement boundary.

5.5 Summary

This work verified the applicability of a reinforcement learning controller to a real hardware, real-time task. The task is too simple and contrived to be of any practical use in itself, but represents a step towards the solution of more complex problems.

5.5.1 State space partitioning

The partitioning scheme had to reconcile two conflicting demands. So the controller could use its experience in novel situations, the boxes in the decoder needed to be large and few. However, to

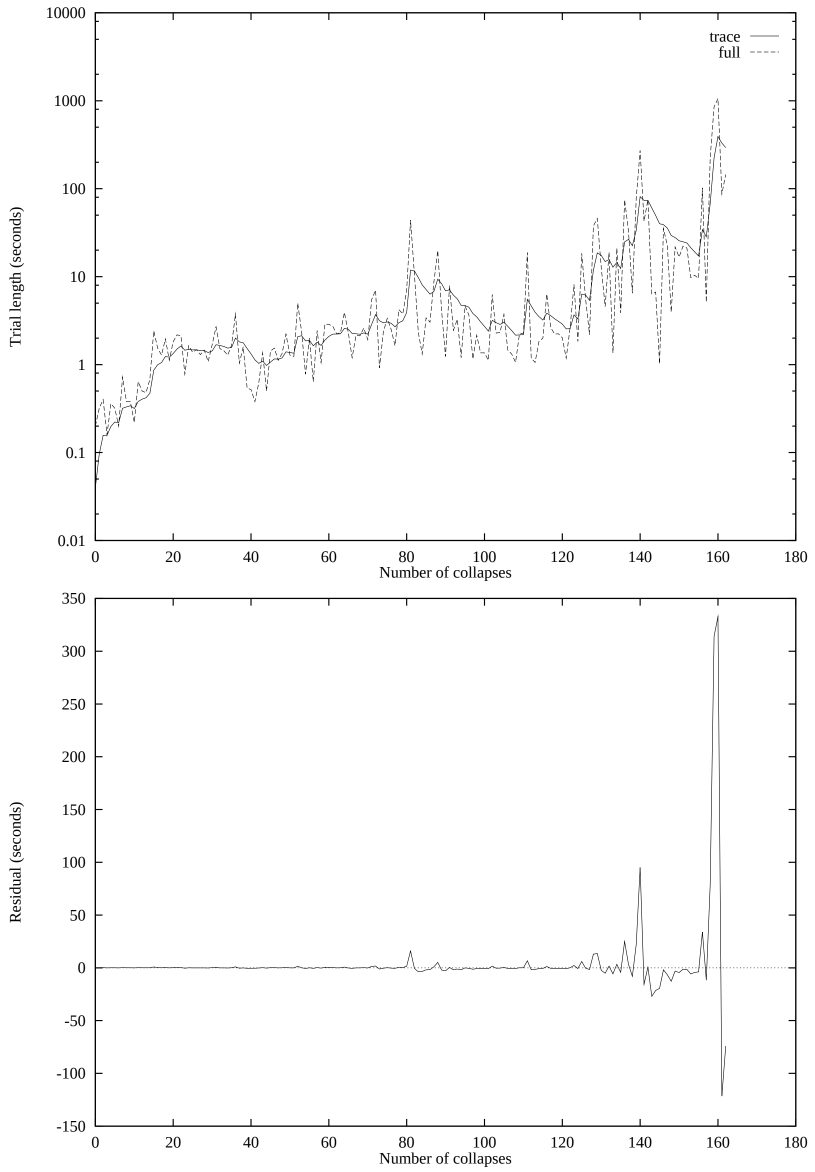


Figure 5.4: A typical 200,000 sample run using 864 boxes, showing the full data and its trace in the upper plot, and the residual between the two in the lower.

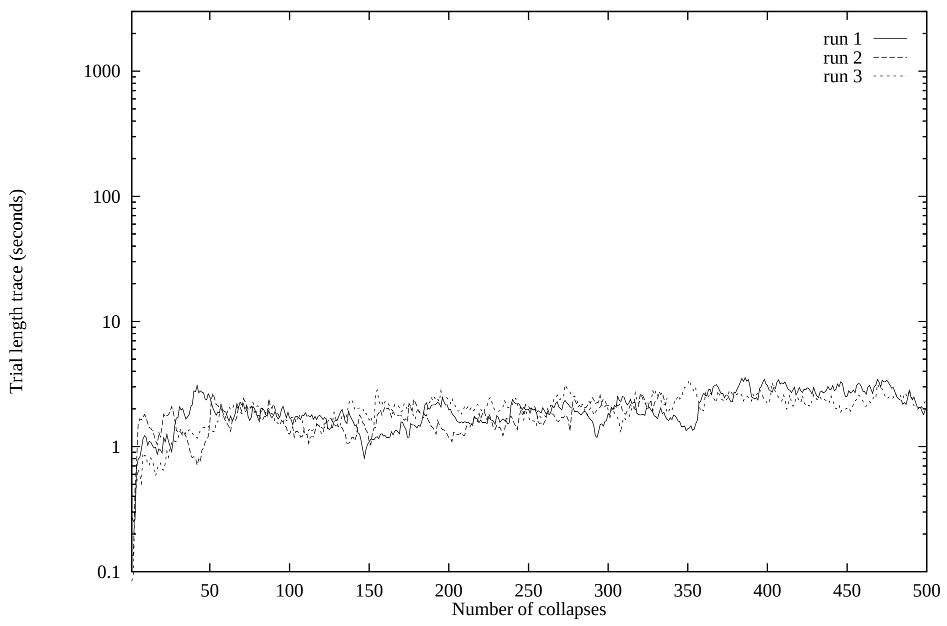


Figure 5.5: Runs using 162 boxes (trace). This shows the lack of performance of the original decoder on the hardware.

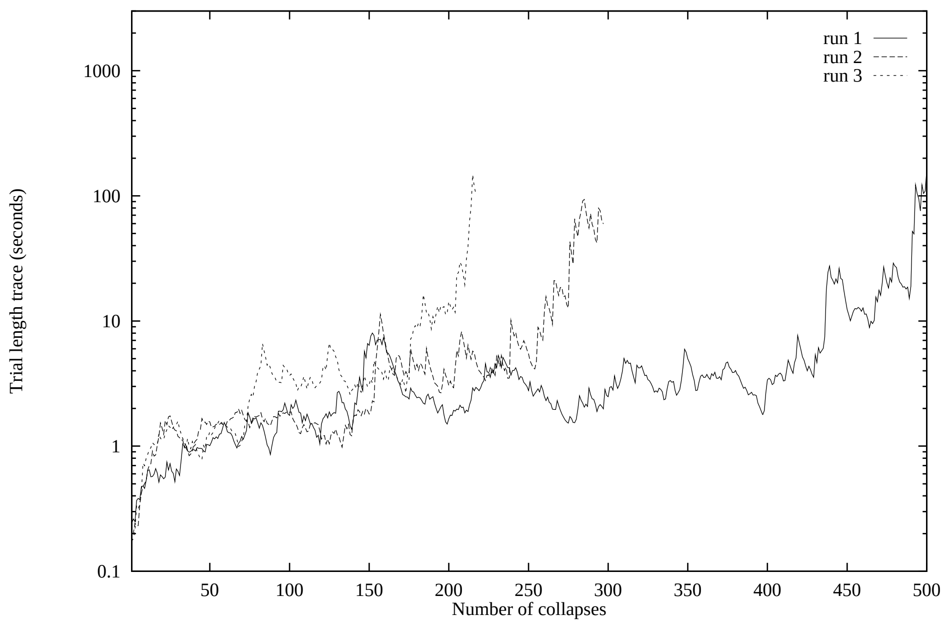


Figure 5.6: Runs using 750 boxes (trace)

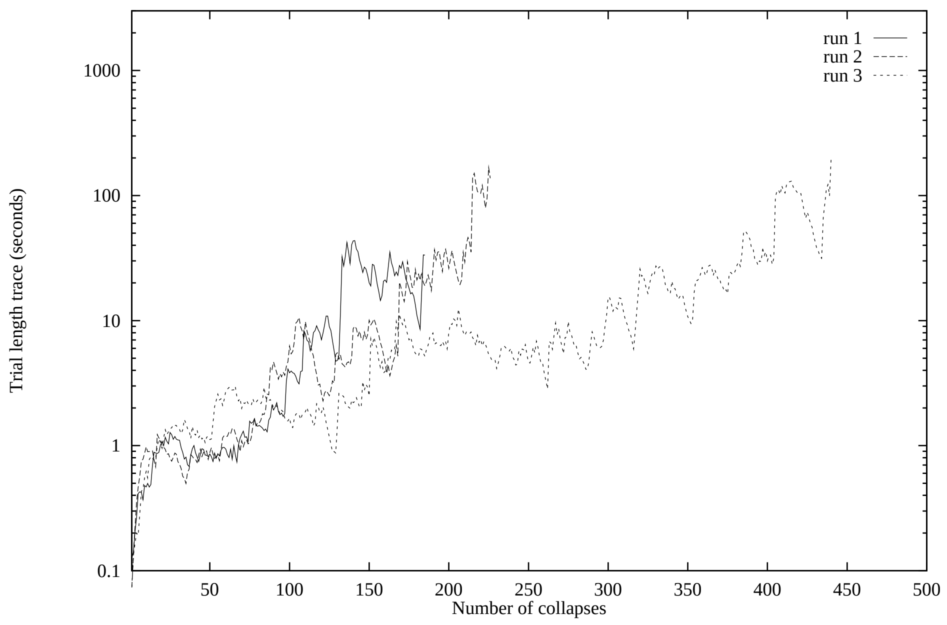


Figure 5.7: Runs using 1296 boxes (trace)

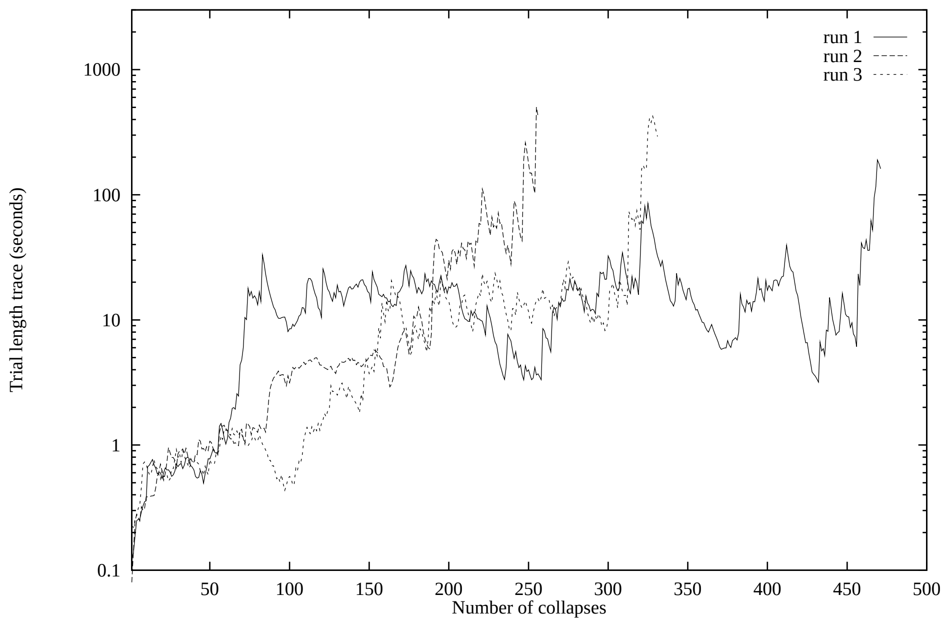


Figure 5.8: 400,000 sample runs using 864 boxes (trace)

resolve the state space sufficiently well for a working control policy, the boxes needed to be small and numerous.

The search for a suitable partitioning scheme would have been hard without the linearized state feedback control expression to choose, and especially to verify, various candidates. Unfortunately, the need for such prior knowledge largely defeats the purpose of learning. It is a consolation that, if it means a policy can be learned which works better than the pre-filled policy, learning might still be worthwhile. A better policy might be learned that took account of delays in the control loop, for example. The effect of delay was explored, however, in section 5.2.3, and was not found to be significant for this particular task.

Although aligning the state partitions with the reinforcement boundary might have improved the learning rate, it would have made no difference to the pre-filled test runs using various partition schemes. These tests suggest that successful partitioning schemes with less than around 1200 boxes are rare.

5.5.2 Exploration

The ASE/ACE controller must explore to find a good control policy. The original scheme of Barto et al. explored by including a noise term in its determination of control actions (Barto et al. 1983). However, the noise was small and did not have a significant effect — the controller worked just as well without it. This suggests that the policy learning task is not hard. The ASE/ACE scheme applied to the real rig also worked with no noise added to the control decision.

5.6 Conclusions

The demands of real hardware require a controller to make as much as possible of its experience.

Although the difficulty of partitioning the state space was overcome for this particular implementation, the problem still stands in the way of a more general utilization of policy learning schemes. Its solution requires more work, which is easier in simulation.

There are more powerful learning schemes available, such as the neural network schemes used by Jordan and Jacobs (Jordan and Jacobs 1990) and Anderson (Anderson 1989), that avoid the need for an explicit state space partitioning. Unfortunately, these schemes require a large number of training examples. Anderson's results showed that around 6000 failures were needed before performance improved (Anderson 1989), an order of magnitude greater than the number of failures necessary for the ASE/ACE scheme to give suitable performance. Although this might be expected of a control scheme that has more to learn, the thousands of examples necessary for the neural network solutions impose severe demands on any physical plant, motivating a search for a more efficient learning scheme.

A learning controller that is successful with more complex problems needs to be more powerful. Learning more control parameters efficiently requires marrying exploration of the plant with a utilization of the plant's resources, a problem known in the field of control as dual control (Feldbaum 1961) and in artificial intelligence as the two-armed bandit problem (Berry and Fristedt 1985).

Chapter 6

Neural network policy optimization

The previous chapter described a successful application of a reinforcement learning controller to an inverted pendulum task on hardware. The application would have been more straightforward if the controller were more adaptive. Such controllers exist, demonstrated by Jordan and Jacobs (Jordan and Jacobs 1990), and Anderson (Anderson 1989, Anderson 1993), but their results in simulation indicate that they require too many control trials to be feasible on real equipment. This motivates an attempt to make the continuous controllers more efficient.

This chapter describes some simple learning controllers based on neural networks, and brings together the ideas of learning and searching under the concept of optimization. The schemes attempt to regulate the inverted pendulum. The basis of the schemes is a return to a simple strategy — a search in policy space. The task is to find a policy with the maximum quality, the policy being a continuous function parametrized by a finite set of real values. Williams proposed this approach (Williams 1992).

The schemes must find a good regulator from a broad class of continuous regulators in as few trials as possible. If the number of trials could be kept within several hundred, an on-line learning scheme without a carefully designed discretization strategy could be applied to a real inverted pendulum rig.

Reinforcement learning problems are normally solved using dynamic programming. Although successful in a small number of finite dimensions, moving to continuous spaces makes dynamic programming difficult. Generally, the convergence proofs valid in finite space no longer apply, because the proofs require finite state and action spaces. There are, however, methods that can be applied to continuous spaces in the restricted linear-quadratic regulation domain (Bradtke 1993).

The approach to continuous spaces in reinforcement learning so far has been to adapt dynamic programming methods, moving to more heuristic approaches. Barto et al. cover a range of dynamic programming variants for large but finite state control problems (Barto, Bradtke and Singh 1991). It may be, however, that the problems of applying dynamic programming in continuous spaces mean that alternative optimization methods become more efficient. Thus, one approach to learning control in continuous spaces is to apply standard optimization routines for continuous spaces to a search for a good control policy.

6.1 Global optimization

Learning control can be formulated as a global optimization problem — how to find the parameters for a controller that minimize the reinforcement cost. Unfortunately global optimization is in general an impossible problem (Törn and Žilinskas 1987). There is a simple local condition for verifying a local minimum (a positive-definite Hessian), but such a condition does not exist for a global minimum.

There are some unusual conditions under which the global optimization problem is viable:

1. The optimum of the objective function, f^*, is known, and some value of x is known that generates f^*. Then x is a global optimum.
2. The number of local minima is known, and all the local minima have been found.
3. An exhaustive search of the space has been completed.

Two optimization strategies that attempt the global optimization problem are simulated annealing and genetic algorithms.

Simulated annealing is based on an analogy with nature. It attempts to find solutions in the way in which nature finds low-energy configurations for crystals as they cool. Practically, simulated annealing methods use a random element in their decision whether to take a step in parameter space or not. There is a non-zero probability that a minimization procedure will take a step to a higher point on the objective function. The probability is determined a parameter of the algorithm known as the temperature. As the temperature varies over time, it is hoped that the algorithm jumps out of local minima to find the global minimum. A problem with the technique is selecting an appropriate annealing schedule.

Genetic algorithms are a variant of a random search. The objective function is evaluated at a number of points in parameter space. A new set of points is then found, based on mutations of representations of the original set of points. The objective function is evaluated at each of these new points. The population of points is periodically reduced, discarding the poorer points.

6.2 Learning control as an optimization problem

The exciting element to learning control is contained in optimization. In learning control, it is hoped to form a mechanism that learns to do better through its experience over time. Optimization methods have the same task. Over time, they gain experience of their environment — the objective function — and learn to produce better results, finding the point in parameter space with an extreme value for the objective function.

This chapter formulates the standard learning control problem of regulating the inverted pendulum as an optimization problem.

The strategy in optimization research has traditionally been to hunt constantly for the optimum, and largely avoid forming complex intermediate models. Few strategies for multi-dimensional non-linear optimization use anything more complex than a localized quadratic approximation. Approaching the problem of optimizing the parameters of a controller by using straightforward optimization strategies without building a complex intermediate model falls within the mainstream of optimization ideology.

6.2.1 An objective function for the inverted pendulum task

The quality of regulation for the inverted pendulum is measured by the time the pendulum remains upright, starting from a state which, if left with no regulator, would rapidly lead to its collapse.

Let the duration of regulation, from a starting state $\mathbf{s}_0$, using a regulator with parameters $\mathbf{p}$, be given by $d(\mathbf{p}, \mathbf{s}_0)$. For the tests carried out here, $\mathbf{s}_0$ was set as follows:

$$\mathbf{s}_0 = \begin{bmatrix} \theta \\ \dot{\theta} \\ x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} \frac{\pi}{180}\text{ rad} \\ 0\text{ rad s}^{-1} \\ 0\text{ m} \\ 0\text{ m s}^{-1} \end{bmatrix} \tag{6.1}$$

i.e. the Barto start state given in table B.1, save for an initial pendulum angle of 1°.

With fixed initial state $\mathbf{s}_0$, the objective function $f(\mathbf{p})$ was set to the reciprocal of the regulation duration. The reciprocal was chosen so that a minimization of the objective function would give parameters that maximized the regulation duration.

$$f(\mathbf{p}) = \frac{1}{d(\mathbf{p}, \mathbf{s}_0)} \tag{6.2}$$

Thus the task for all the following optimization schemes is to find $\mathbf{p}^*$, given by:

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} f(\mathbf{p}) \tag{6.3}$$

The objective function may be sampled at a point by applying a fixed set of control weights and monitoring the regulation performance (finding the time to collapse of the pendulum).

The problem is a reinforcement learning problem because a collapse gives only a qualitative measure of the performance of the regulator. It does not indicate in what way the regulator could be changed to give improved performance. This necessitates a learning controller that explores. The framework is not a general solution to reinforcement learning problems, however, because it relies on repeated trials and a binary reinforcement.

6.2.2 Simulation parameters

The neural network regulators were tested on an inverted pendulum simulation using the parameters of Barto et al.'s original simulation (Barto et al. 1983). The decision to use the Barto parameters was swayed by the intention to produce results that could be compared with those of other authors, specifically Jordan and Jacobs (Jordan and Jacobs 1990), Brody (Brody 1992), and Anderson (Anderson 1989, Anderson 1993).

A review of the inverted pendulum regulation task is given by Geva and Sitte (Geva and Sitte 1993). They point out that the task is not difficult, especially in the form most commonly attempted, that is, with the Barto parameters. They show that with relatively straightforward prior analysis of the problem, they can limit a search for the parameters of a linear controller to the positive hypercube of the four-dimensional parameter space. They concentrate on the problem from a control perspective, and discuss the quality of regulation as well as giving parameters for the task that make good controllers more difficult to find.

The view taken here is that reinforcement learning is concerned with the optimization problem of finding suitable parameters given uninformative prior knowledge. The task is to find a learning controller that is efficient with its experience. The Barto parameters are still appropriate for this task.

For comparison, both the Barto and the Geva parameters are given in appendix B. To avoid converging to a policy that gave only zero control actions, the perturbed Barto state state given in table B.1 was used for the experiments to follow.

6.2.3 The regulator

The regulation task may be solved with a linear state feedback regulator. A good review of the quality of regulation that can be achieved by this method is given by Geva and Sitte (Geva and Sitte 1993).

The purpose of attempting learning control for the inverted pendulum regulation problem is more than finding a good regulator, however. It is finding a good regulator from weak prior knowledge of the problem. Weak prior knowledge about the form of the regulator means that the space from which the regulator is to be selected is large.

A neural network, with the state of the plant as input, and no hidden layers, can form a linear state feedback regulator. It is possible, though, that a linear regulator may not have been powerful enough for the problem. It is also still feasible that a non-linear regulator might be more robust. We might therefore wish to search a space of linear and non-linear regulators. However, the task undertaken here is to compare parameter optimization using traditional techniques with the learning controllers of other authors. Jordan and Jacob's used a linear regulator (Jordan and Jacobs 1990), which has once again motivated the choice of regulator here.

A block diagram for a linear regulator is shown in figure 6.1.

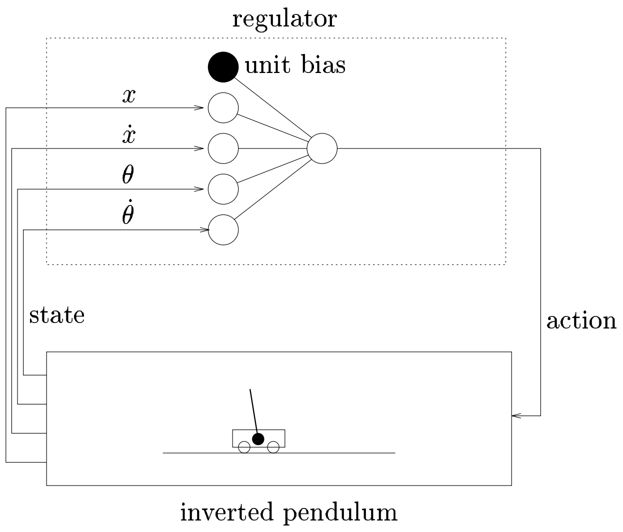


Figure 6.1: A linear regulator for the inverted pendulum.

This work is concerned with finding the parameters for a continuous regulator. Finding the parameters is an optimization problem. A comparison with a random search will compare the worth of the various optimization strategies.

6.3 Optimization strategies

All but the last of the optimization schemes below are detailed in Numerical Recipes (Press et al. 1992), and so only an outline of each of these schemes is given here. Common among the schemes is their lack of reliance on gradient information of the objective function.

6.3.1 Random search

The simplest search strategy is a random search. It is a baseline against which other optimization schemes may be compared.

The parameter space for the neural network regulator is a 5-dimensional space of real values. $50,000$ samples were taken, using a Gaussian distribution with zero mean and a standard deviation of 10.

The algorithm is as follows:

1. Select the regulator's weights at random
2. Apply the controller and assess the performance of the controlled plant
3. Return to step 1

6.3.2 Powell's method

Powell's method of optimization is a variation of performing a line minimization along each of the dimensions of the parameter space in turn. The technique is possible without gradient information if a line minimization procedure that does not need gradients is available. A line minimization method such as that described in section 3.6.2, that successively finds closer bounds on a minimum, is suitable.

The insight of Powell's technique is to find conjugate search directions. The advantage of conjugate search directions has been stated already in section 3.7.1.

The concept of performing a line search in the parameter space of the regulator at first seems inefficient in terms of a learning control method. However, the principle of this chapter is that optimization methods have been designed to find optima efficiently, and that a learning controller based on a conventional optimization algorithm might be efficient.

6.3.3 Nelder and Mead's simplex algorithm

A simplex is a polyhedron with N+1 vertices in an N-dimensional space, so, for example, a simplex in a 2-D space is a triangle. The simplex algorithm works by changing the shape of a simplex in the parameter space of the regulator. An initial simplex in the regulator's weight space is defined by the tips of the unit vectors in the parameter space, and the origin. This simplex is then scaled by a factor of 2 and translated by subtracting 1 from each element. This is so as to leave the simplex sitting over the origin.

The next step is to evaluate the objective function at each corner of the simplex. Each evaluation is a trial balance, ending in a collapse.

The algorithm then attempts to change the shape of the simplex so that the corners sit at better points in parameter space. It does this by reflecting, scaling, and shrinking. These operations are detailed in Numerical Recipes (Press et al. 1992).

6.3.4 A simulated annealing variant of the simplex algorithm

A variation of the simplex algorithm can be made in an attempt to avoid local minima and capture a global minimum.

The simplex algorithm maintains the values of the objective function at each vertex of its simplex in parameter space. The algorithm details how the simplex might be deformed, in search of a lower point on the objective function's surface. A deformation is normally accepted only if the evaluation at the new point is lower than the current lowest vertex.

The algorithm can be changed so that there is a probability, related to the value of the new point in relation to the others, that the new point will be accepted. It is hoped the algorithm might leave local minima and find deeper minima.

The simulated annealing technique used here attempted a block of minimizations at a fixed temperature. The temperature was then reduced, and more minimizations were tried. Block sizes of 10 and 20 were tried. The temperature was given by $exp(-i)$, where i is the number of previous block minimizations.

6.3.5 Predictor network search

This scheme uses a second neural network (the regulator itself is the first neural network) to model the relationship between the regulator's parameters (the weights of the first network) and the regulator's performance. The model is trained with data produced by trials on the plant.

Using the back-propagation algorithm (section 3.5), derivatives of predicted regulation quality with respect to the regulation parameters can be calculated. The derivatives allow for an efficient search for the optimum regulation parameters. Any optimization technique that requires only point and gradient information may be used to find candidate control weights. Quickprop (section 3.8.3) was used for the results reported here.

One motivation for the predictor network approach was to avoid the non-stationary objective function inherent in TD(λ). The targets for TD(λ) optimization change with time, producing conflicting input/target patterns.

A problem with the forward-modelling approaches described in section 4.1.5 is that while the forward model is trained as a function value approximator, only the derivatives of the model are used. These derivatives, as Brody has pointed out, might be in arbitrary directions (Brody 1992). This problem hinders the progress of training using the forward modelling technique.

The scheme described here is not like the forward modelling method in this respect. The derivatives of the approximated objective function are used solely for a search for the maximum of the approximated function. The derivatives are appropriate for this task. There is no need for the gradients to be meaningful with respect to the real plant. It may be that the optimum found using the derivatives is not optimum in practice, because the objective function is a poor approximation to the true objective function. In this case, the mismatch can be appropriately handled by considering the error bars on the approximation function.

The predictor network algorithm is as follows:

1. Select parameters for the regulator by finding parameters with maximum predicted quality.
2. Control the inverted pendulum with the regulator until it collapses.
3. Train the predictor network using the parameters of the regulator as input, and the duration of the trial as the target.
4. Return to step 1.

By fixing the control policy throughout a trial, the performance of the policy is accurately determined. Were the policy to be changed in a trial, the quality of the policies would not be clear.

By using a parametrized policy, rather than state and action, as input to the predictor network, the predictor is never given conflicting training data. A fixed policy will always give the same performance. The quality of a state and a single control action will have a quality that depends upon future control actions. Using states and actions individually can lead to conflicting training data.

The concept behind the predictor network scheme is shown in figure 6.2. The figure shows four snapshots from ten stages of the scheme finding the maximum of the function

$$-0.01 \cosh x + \sin x. \tag{6.4}$$

At each stage, the network is trained on the currently available samples of the objective function, and the maximum of the regularized interpolant is found using a gradient-based optimization procedure. This maximum is the candidate optimum for the next stage. Both the output of the network, and the regularized output of the network, are shown. Regularization of the network's output is discussed below.

The input of the interpolator is the set of regulation parameters, in the form of a vector of weights for the regulation network. The output of the prediction network is a representation of the time to failure of the inverted pendulum system when regulated by the control network.

Exploration

Before sufficient data about the plant have arrived, the predictor model will be poor. Parameter choices from the model are likely to give poor regulation. The model will be better given a sufficiently rich data set from which to train.

One way to generate a rich data set is to use random choices of regulation parameters for an initial period of some fixed length. The solution encompasses the idea that initially, information about the problem is scarce and the predictions will be poor. Later, when more information is available, the predictions are more likely to be accurate and a random search is less efficient. However, the solution is not elegant. It does not specify how long the exploration phase should be.

An alternative strategy that avoids choosing an arbitrary point at which to stop random searching is to use a measure of confidence in the performance prediction to determine how much random exploration to make. This solution encompasses the idea of using a random search when information about the problem is scarce.

For the final results shown later, the data set used to initialize the the simplex algorithm was chosen. Randomized regulation parameters were not used.

Without a random element, only the parameters that optimize the prediction surface will be evaluated. If the prediction surface happens to be accurate for those parameters, the surface will not change and convergence will have stopped. This problem is illustrated in figure 6.3.

The results suggest that this problem has been avoided for the task chosen, but it is inevitable that the problem will arise in other cases, since the global optimization problem is not tractable. Experience with other tasks will tell if the problem is significant in practice.

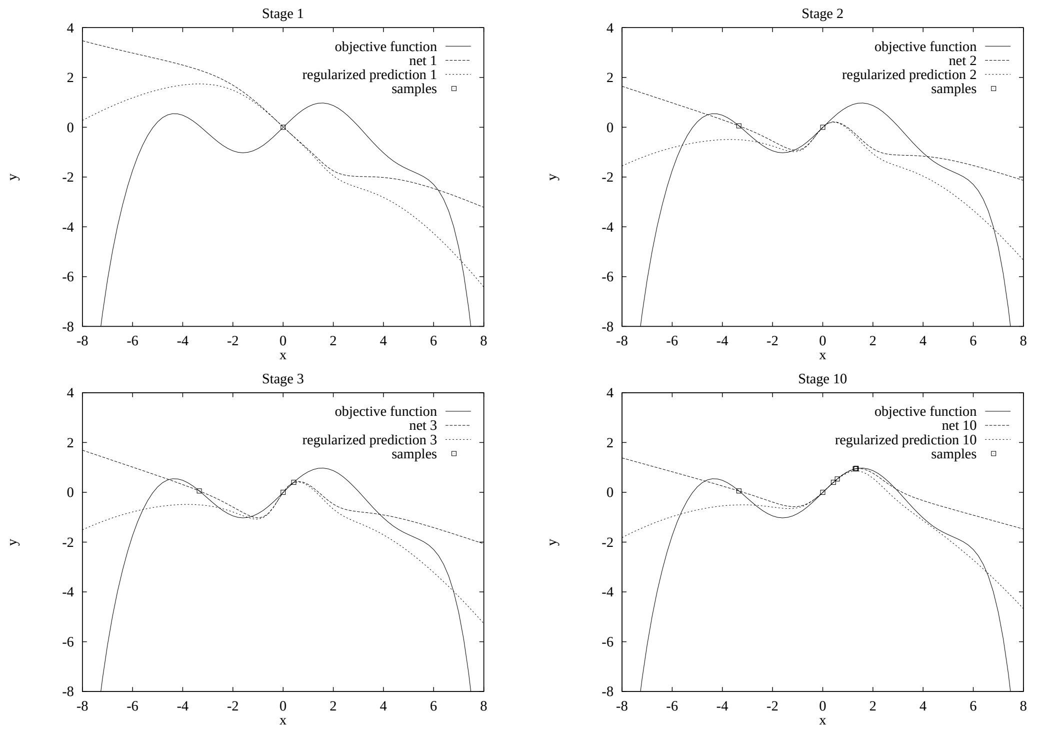


Figure 6.2: An illustration of the concept behind the predictor network optimization method. The first three and the last of ten iterations to maximize the function $-0.01 \cosh x + \sin x$ are shown.

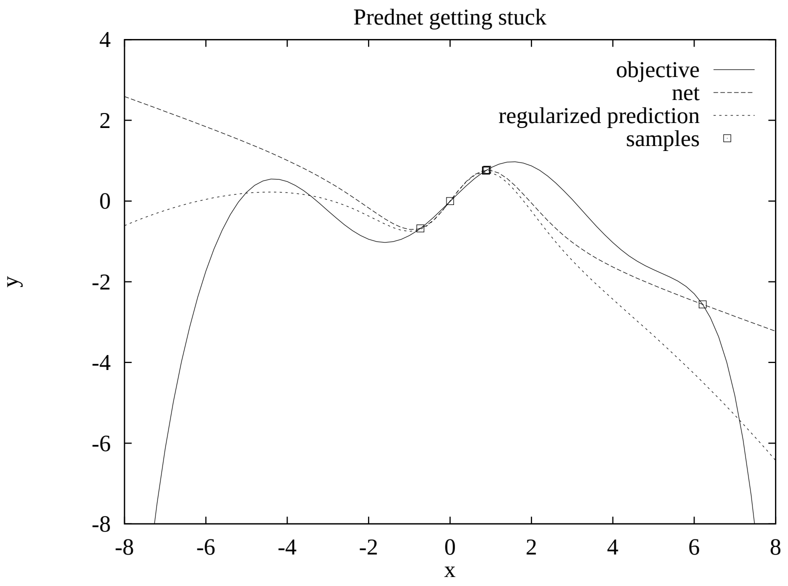


Figure 6.3: One way in which a lack of exploration can stop convergence using the predictor network. The approximation at the approximator's maximum is accurate, so samples of the maximizing parameter will not change the approximation. The algorithm has stopped.

A further exploration strategy that could be applied to this task would be to randomize the regulation parameters by an amount based on the current performance of the regulator. Adding noise in inverse proportion to the regulator's performance avoids the problem of getting stuck in a local minimum after the algorithm has 'cooled down' after an annealing schedule.

Representing the regulation performance

The duration of the regulation may be represented in several ways. The representation should be such that the model of the performance distribution is appropriate. Some alternatives are:

1. A linear output representing the duration directly.
2. A linear output representing log duration.
3. A log-sigmoid representing the reciprocal of the duration.

Representation 1 wastes half the output range since negative durations are not possible. Alternatives 2 and 3 avoid this problem. Representation 2 was used for the experiments attempted here. Representation 3 was by Jordan and Jacobs (Jordan and Jacobs 1990).

Optimistic or pessimistic predictions?

Whether the predictor is optimistic or pessimistic in areas which are unexplored will affect the optimization procedure. Barto's ASE/ACE controller (section 2.3) was optimistic, since each virgin state predicted no reinforcement cost, and states that were previously occupied were always updated towards a negative reinforcement prediction. This bias led to an exploration of all possible actions for each discrete state visited.

The reliance on an optimistic prediction function is not appropriate in continuous spaces, since the set of alternative actions is infinite. In practice, an optimistic prediction function in a continuous policy space will lead to a non-converging search into policies with elements of large magnitude.

The form of the prediction surface in unexplored regions is related to the asymptotes of the neural network interpolation. The asymptotes depend on the network's connectivity and its output function. A layered connectivity will give bounded asymptotes, since the input to the final layer will be bounded by the saturating hidden layer. If the output layer is not linear, but a sigmoid, the output will saturate for large input. A fully connected network with linear output nodes has unbounded asymptotes.

The problem with the asymptotes is that finding regulation parameters with the greatest predicted performance might lead to finding unnaturally large regulation parameters.

One solution to this problem might be to use a radial basis function approximator. The asymptotes could be controlled so that it may be consistently optimistic or pessimistic for points far from the centres of the basis functions.

Another solution is to modify the quality function to penalize extreme policies. A form of regularization, introduced in section 3.2.1 to penalize over-fitting in data regression, can be used to reduce the value of the prediction function for extreme values of the policy. This is the basis of the method used here. A modified quality function $\tilde{q}$ was defined:

$$\tilde{q} = q - \frac{\gamma}{2} \sum_i p_i^2 \tag{6.5}$$

where q is the original prediction of quality from the predictor network, a representation of the duration of a regulation trial; p_i is the i'th regulation parameter; and γ is the regularization parameter. The gradient of this objective function is given by:

$$\frac{\partial \tilde{q}}{\partial \mathbf{p}} = \frac{\partial q}{\partial \mathbf{p}} - \gamma \mathbf{p}. \tag{6.6}$$

This gradient can be used to find the parameters that maximize the objective function $\tilde{q}$. The inclusion of the regularization term avoids the parameters growing too large when finding an extremum of the prediction surface. γ was set to 5×10^{-4} for the results to follow.

Searching for the maximum of the prediction surface

The maximum of the prediction surface is another global optimization problem. However, in this case the search is helped through access to the training data for the prediction surface. The maximum of the surface will be close to the maximum datum in the data set. Finding the maximum of the training set is not difficult, because the set is finite. Furthermore, in an on-line scheme, keeping track of the maximum is a quick, incremental procedure.

The predictor network method starts a search for the parameters with the greatest predicted performance from the best parameters in the training set. From the start point, gradients of the prediction surface are used to find the maximum of the function. The predictor network is acting as an interpolator between the points in the training data. The training data are the accumulated experience of the learning controller.

A final detail of the search was the inclusion of a simulated annealing schedule to select the point in the training data from which to start the search. Zero mean Gaussian noise was added to the performances in the training set before choosing the maximum. This meant that parameters with performances close to the maximum might get chosen in place of the maximum. The variance of the noise exponentially decreased with the number of trials. The simulated annealing approach was motivated by an analysis of the predictor network optimization method using a single-input, single-output function. The example suggested that it would be worthwhile to occasionally search from a point that was not the best in the training set.

6.4 Training procedure

Each run comprised a series of trial periods of regulation. For each trial, the system state was set to the perturbed Barto values given in table B.1. No noise was introduced to the system. Applying zero force to the system led to a short trial, because the initial system state was not at an equilibrium point.

The state of the system was passed to the regulator, which produced a control action. The current state and control action were updated using equations B.24 and B.23, and Euler integration over a simulated time-step of 0.02 seconds.

When the state of the pendulum fell out of the prescribed bounds, given in table B.2, the trial was terminated and the reciprocal of the duration was passed as the result of the evaluation of the objective function at the current point in the regulator's parameter space. A minimum of the objective function corresponded to a maximum balancing duration.

Each run was terminated after a set of parameters was found that kept the pendulum upright for more than half an hour of simulated time, or when the algorithm converged to a local optimum, or when a suitably large (generally 5000) number of collapses had occurred.

The initial regulation parameters for the Powell search were chosen at random from a unit variance zero mean Gaussian.

6.5 Results

Figure 6.4 shows a histogram of the regulation performance through randomly sampling the parameter space of the continuous regulator. This acts as a benchmark for the alternative optimization schemes. The graph shows the frequency of controllers that balanced the pendulum for the duration indicated on the abscissa. The sampling was Gaussian with zero mean, and standard deviation 10. From $50,000$ samples in the parameter space, three were still balancing the pendulum after the cut-off period of 2000 seconds, or 33 minutes.

The results of the simplex algorithm without simulated annealing are given in figure 6.5. This method failed to converge to a satisfactory policy. Figure 6.6 shows the progress of a simplex

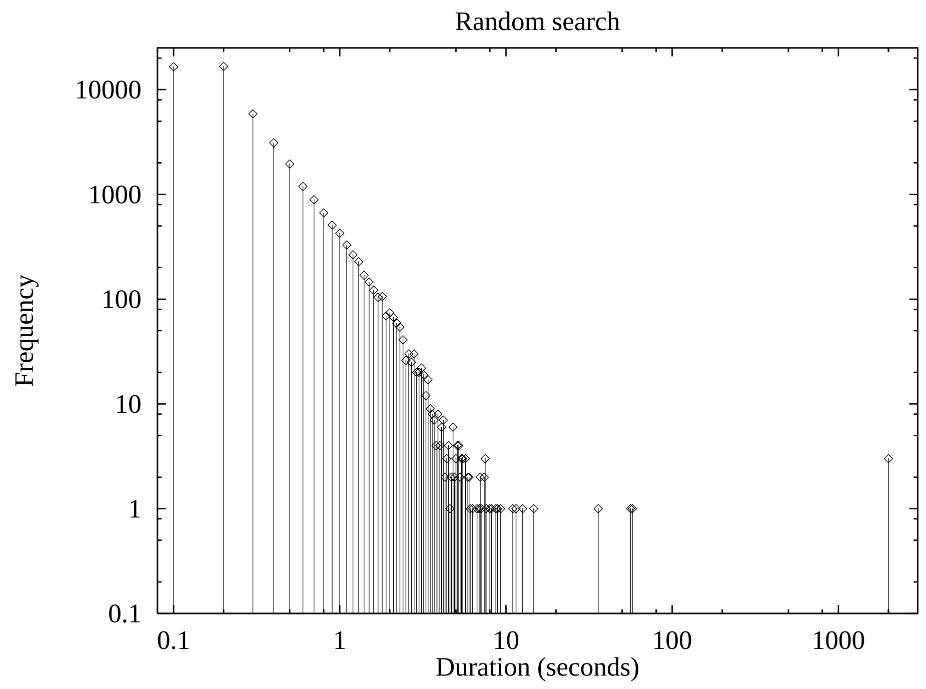


Figure 6.4: Results of a random search in the regulator's parameter space. The histogram shows the results from 50, 000 samples in the regulator's parameter space. Trials were stopped after 2000 seconds. It was assumed that otherwise they would have continued indefinitely.

algorithm with simulated annealing. This algorithm found a set of regulation parameters that gave a satisfactory control policy.

Figure 6.7 shows the progress using Powell's method to search for good regulator parameters. The progress of the line searches can be seen in the periodic structure of the results.

Figure 6.8 shows the progress of the neural network prediction procedure. The predictor network had 5 hidden log-sigmoid nodes, a single linear output node and layered connections, giving 36 parameters. It was trained using BFGS with lnsrch (see section 3.9.4) on a history of regulator parameters and their performances. Once there were as many examples as parameters of the predictor network, the history was kept at a constant size by removing the oldest entries.

The predictor network method produced the most efficient search of the regulator's parameter space.

6.6 Summary

This chapter has presented an alternative method of approaching learning control, using optimization techniques in a parametrized policy space. The techniques do not use gradient information from the objective function, because such information is not available in the reinforcement learning context. Thus, in terms of standard optimization, the number of function evaluations may seem large. However, in terms of standard reinforcement learning strategies, thousands of trials, where one trial is equivalent to one evaluation of the objective function, is not unusual.

Four optimization strategies were tried. Of these, a variant of the simplex method using simulated annealing and the predictor network method were successful. Their performance compares with Brody's controller (Brody 1992), but because they avoid the need for a pre-trained world model, they are closer to the goal of a generally applicable learning controller.

The methods used are a step backwards in terms of the mainstream reinforcement learning methods, because they rely on a trial-based problem, and update after each trial. They do this for the benefit of a stationary objective function. If the requirement for a trial-based problem can be

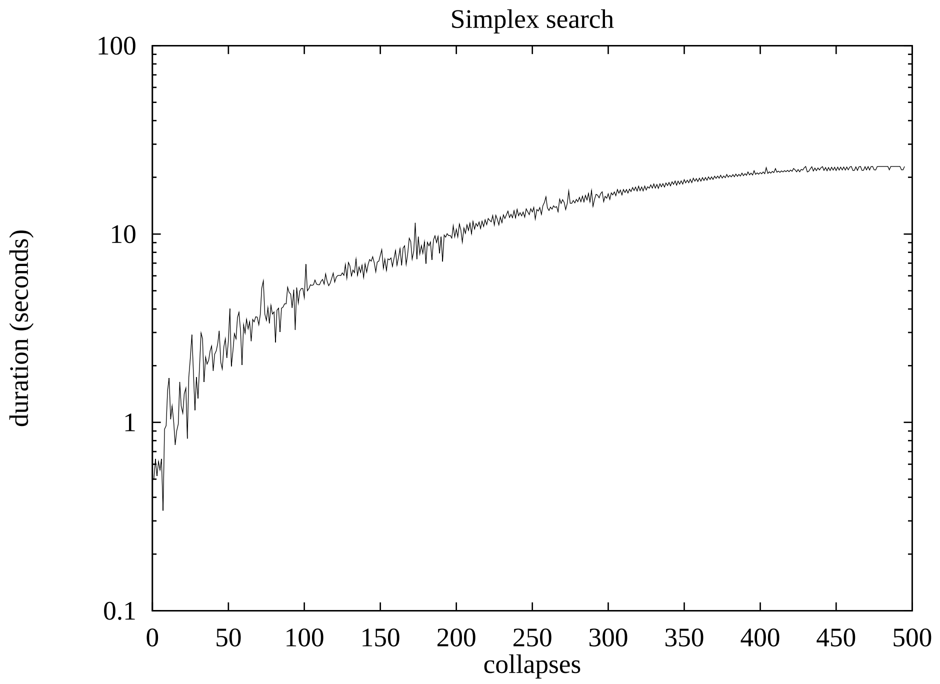


Figure 6.5: Results of the simplex search without simulated annealing. This search failed to find a satisfactory regulator.

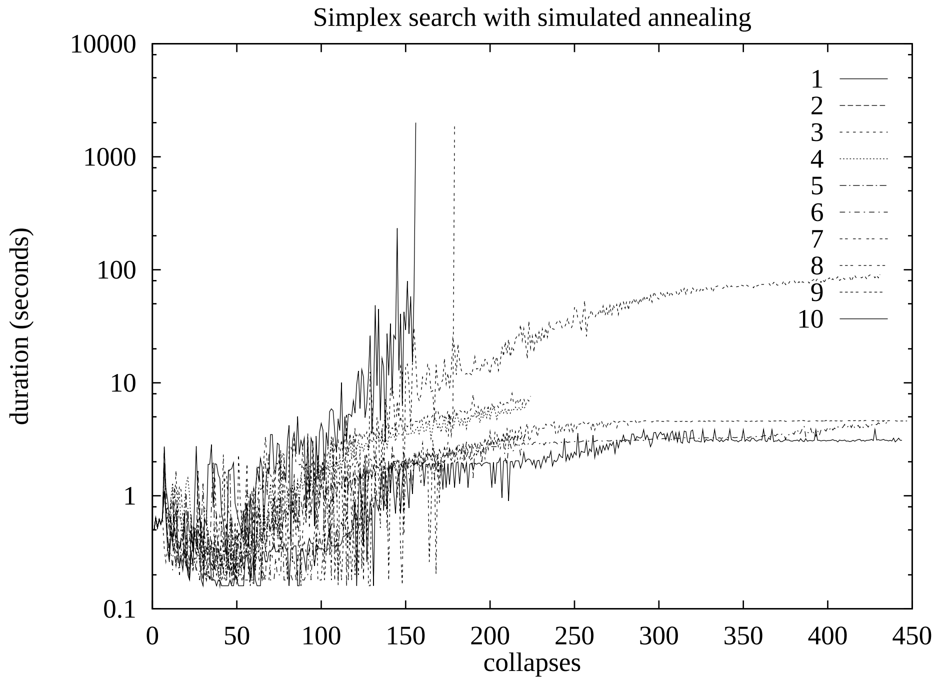


Figure 6.6: Results of a simplex search with simulated annealing in the regulator's parameter space. The first five runs perform 10 iterations at each temperature in the annealing schedule. The second five perform 20 iterations per temperature setting. Different seeds for the random number generator were used for the runs.

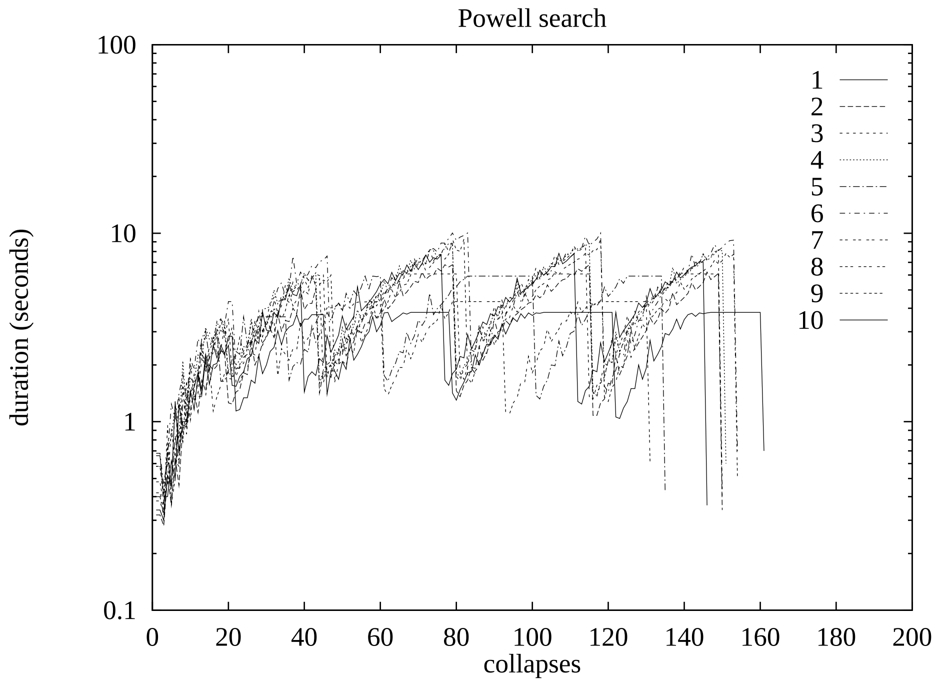


Figure 6.7: Results of searches using Powell's method in the regulator's parameter space.

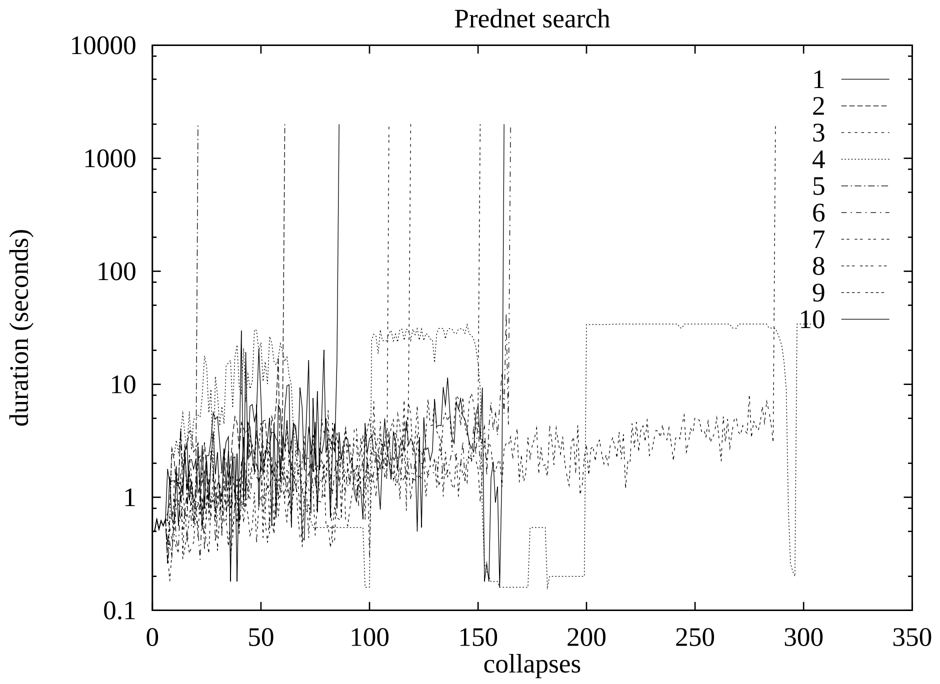


Figure 6.8: Results of the neural-network modelling optimization procedure. The runs differ only in the value of the initial seed of the random number generator. This method was the most efficient learning controller tried.

avoided, the scope of the algorithms is widened.

The predictor network method might be extended towards non-trial based reinforcement problems, by updating the prediction surface with intermediate results.

Chapter 7

Summary

This study has attempted to go beyond the field of traditional adaptive control, confined mostly to system identification. Assumptions about the environment have been made as broad as possible, and the paradigm of reinforcement learning has been used to give a general framework for learning control. The methods have attempted to learn to give improved control performance through experience.

The main thread of the work has been to study the inverted pendulum problem, firstly in simulation, and then on real apparatus. The results of this study showed insufficient adaptability, and not processing demands, to be the problem. Continuous regulators were needed because they avoided the problem of insufficient adaptability, but results of such regulators in simulation showed that they would require more experience to learn than could reasonably be gained on real equipment. The work then attempted to find a learning scheme for continuous regulators for the inverted pendulum task that learned with fewer trials than currently reported in the literature. Of the schemes attempted in chapter 6, a simulated annealing variant of Nelder and Mead's simplex algorithm, and a new optimization scheme based on a connectionist model, found good regulation parameters in as many trials as might reasonably be expected to be feasible on hardware.

The conclusion of the work is that learning control may be considered as an optimization problem in control policy space, and that using such an approach can lead to efficient learning. Optimization methods have been developed to search spaces efficiently, and these may be applied to finding learning controllers for general control problems.

7.1 Real-time constraints

Chapter 3 presented some results of training neural networks using alternative optimization techniques to those commonly in use. Because the results were calibrated in floating-point operations, rather than by the number of passes through the training set, the algorithms could be ranked in efficiency in terms of computational effort rather than the number of training epochs. When neural networks are applied to real-time problems, algorithms that are expedient in time and computational effort will be needed. The chapter showed that algorithms that are more complex than gradient descent with momentum are worth their computational overhead.

Chapter 8

Future work

As discussed in section 6.1, the global optimization problem is not solvable except under restrictive conditions. Reinforcement learning is a global optimization problem for which these conditions will not hold in general. The reinforcement learning problem is therefore left broadly unsolvable. This need not, however, mean that expedient use of computational resources in optimizing parameters cannot be done. Using computer resources for optimization in policy space is a general way to do useful future work.

Work can continue in learning control in two directions. Firstly, the methods of learning control may be developed. Secondly, learning controllers might be applied to real problems.

8.1 Extending learning control methods

This thesis suggests that in moving from the finite domain into the continuous domain, learning control might advance by leaving behind its traditional reliance on dynamic programming algorithms. Alternative optimization strategies developed for continuous domains might be useful in machine learning. The success of the simplex algorithm and the predictor network algorithm in chapter 6 would suggest that the application of these algorithms to other learning control tasks might be worthwhile.

8.1.1 The predictor network

The idea of modelling control performance, and choosing parameters based on the predictions of the internal model, is attractive: the predictor network approach is firmly based in the continuous domain, necessary for real problems; its inference of which policies to attempt next is explicit; its choices are based on its whole experience, summarized by the predictor network model. The method needs to be investigated further.

Although one motivation for the predictor network was to avoid a non-stationary objective function for the predictor, it is clear that the method is limited by a need for repeated trials. Temporal difference methods avoid this problem, at the cost of a non-stationary objective function. It may be possible to combine temporal difference methods with the predictor network approach to allow a learning controller to update the control policy without reference to trials. This is likely to require augmenting the input to the predictor with the current state, or the observed context. It is possible, however, that resorting once again to a non-stationary objective function will remove the benefit of the scheme.

Further analysis of the rôle of the regularization parameter should be carried out. This parameter embodies a tendency to be risk-averse. In the appropriate representation of performance, this parameter should be redundant. As an example, consider:

$$q\prime = q \exp^{-\frac{1}{2}\alpha\sigma^2} \tag{8.1}$$

where q is the value of the interpolant; σ^2 is its variance, discussed in section 3.2; and α is a constant with appropriate dimensions. An example of this function is shown in figure 8.1. The graph shows an interpolant and its error bars for a partial sinusoid. The error bars widen as the data becomes more sparse, as might be expected. The error bars also widen as the network extrapolates beyond the data. This widening of the error bars comes about from a higher variance and a lower probability of the mean interpolant. Further away from the training data, the regularized interpolant is closer to zero.

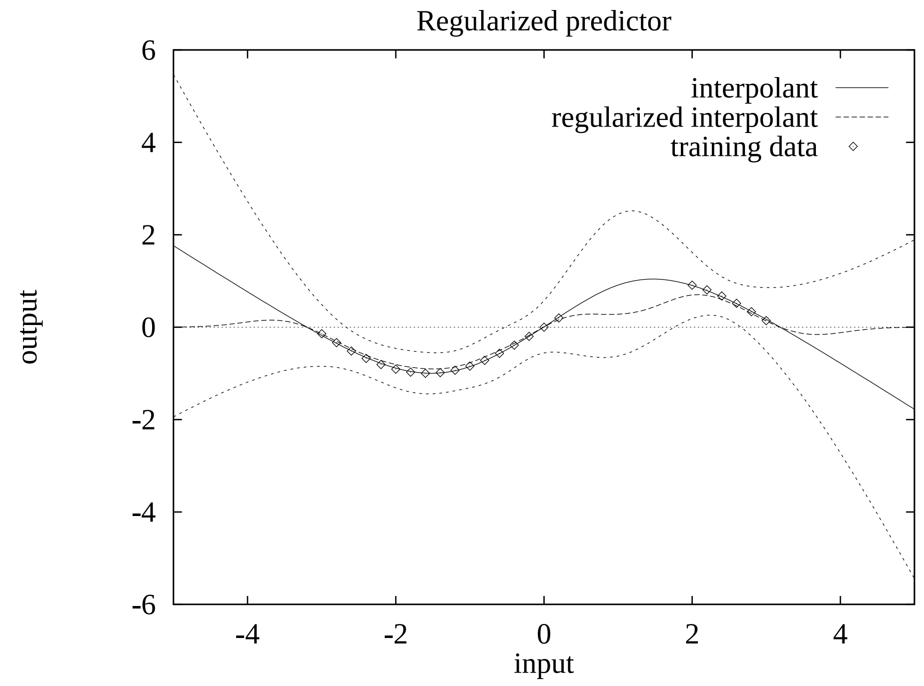


Figure 8.1: The mean value of an interpolant, using the error bars as a measure of the probability of the interpolant. Where the error bars widen, the expected value of the interpolant falls.

In the context of predicting the regulation quality, using the regularized interpolant $q\prime$ would avoid the problem of extreme values of the regulation parameters being predicted a high value of quality. A regularization parameter would not be necessary.

8.1.2 Incomplete state information

Both the predictor network and the simplex search with simulated annealing should be able to find control policies for regulating the inverted pendulum that use incomplete state information. The regulators might use as input the current and previous values of the pendulum angle θ and trolley position x. The learning control framework developed in chapter 6 is unaffected by incomplete state information.

Imagine a controller given a plant to control, but given incomplete state information and no time history of the inputs. One can speculate that after placing a memory device beside the plant, the learning control methods of chapter 6 might be used to find a controller that controlled the memory and the plant to overcome the problem of incomplete state information.

8.2 Machine learning applied to real problems

Learning control will come of age only when it is successfully demonstrated on impressive problems. Unfortunately, the problems that have been attacked so far have fallen short of this goal, although Tesauro's backgammon player is an encouraging development.

Tzirkel-Hancock has used neural networks in conventional control paradigms as adaptive elements (Tzirkel-Hancock 1992). His scheme is provably stable, but is limited to affine control problems (problems that are linear in the control variables). Although this class of problem includes robots, it excludes some simple plants such as the ball-and-beam control problem, where the control problem is to position a ball which is free to roll along a pivoting beam.

Nevertheless, embedding neural networks as adaptive elements in a provably stable control scheme is a way of widening the application of neural networks to real-world problems with reduced risk to plant and equipment.

The methods of chapter 6 might be used to extend learning control to other problems. The methods could be applied to the hardware described in chapter 5. It should be possible to formulate a problem based on flicking a pendulum upright from an initially stationary, hanging position. Because the methods are in the continuous domain, the need for a hand-made discretization of the state space for the problem is not necessary.

8.2.1 A candidate problem for learning control

Vehicle suspension systems (Sharp and Crolla 1987) offer a class of problems that may be tackled by learning control. A semi-active suspension system has a variable damping factor for the shock-absorber. The factor may be adjusted by varying the size of an orifice in the piston of the shock-absorber.

Controlling the ride that comes from semi-active suspension is difficult because linearizing the effect of the damping factor is not reasonable, as it is a friction term.

A learning control neural network approach could be attempted for this non-linear problem. The techniques of chapter 6 might be applied to finding parameters for a non-linear neural network controller that optimized an objective function encompassing a measure of smoothness of ride.

Appendix A

RBackprop

RBackprop is an algorithm for calculating the product of the hessian $\mathbf{H} = \frac{\partial^2 f}{\partial \mathbf{w}^2}$ of a surface $f(\mathbf{w})$ (equation 3.21) and an arbitrary vector $\mathbf{v}$ for neural networks. The name of the algorithm comes from the notation used by Pearlmutter in his exposition of the technique (Pearlmutter 1993). The algorithm is significant because not only does it give an exact result, but it is also computationally cheap, especially if the gradient has already been computed (as is the case, for example, in conjugate gradient descent). It can be used to find the exact Hessian for a network by taking multiple products of vectors that extract consecutive columns of the Hessian.

Let $\mathbf{w}$ be a point in weight space, r a small scalar and $\mathbf{v}$ an arbitrary vector in weight space. Then

$$\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+r\mathbf{v}} = \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}} + r \left.\frac{\partial^2 f}{\partial \mathbf{w}^2}\right|_{\mathbf{w}} \mathbf{v} + \mathcal{O}\left(r^2\right) \tag{A.1}$$

where $\mathcal{O}\left(r^2\right)$ represents terms of order r^2 and higher.

Substituting $\mathbf{H} = \left.\frac{\partial^2 f}{\partial \mathbf{w}^2}\right|_{\mathbf{w}}$, rearranging and dividing by r gives:

$$\mathbf{H}\mathbf{v} = \frac{\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+r\mathbf{v}} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}}{r} + \mathcal{O}\left(r\right). \tag{A.2}$$

The exactness comes from taking the limit as $r \to 0$:

$$\mathbf{H}\mathbf{v} = \lim_{r\to 0} \frac{\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+r\mathbf{v}} - \left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}}}{r} = \frac{\partial}{\partial r}\left.\left(\left.\frac{\partial f}{\partial \mathbf{w}}\right|_{\mathbf{w}+r\mathbf{v}}\right)\right|_{r=0}. \tag{A.3}$$

The term on the right hand side is the key to the algorithm. Defining the following differential operator $\mathcal{R}_v\left\{.\right\}$, from which RBackprop gets its name:

$$\mathcal{R}_v\left\{f(\mathbf{w})\right\} = \frac{\partial}{\partial r} f(\mathbf{w}+r\mathbf{v})\Big|_{r=0} \tag{A.4}$$

gives us $\mathbf{H}\mathbf{v} = \mathcal{R}_v\left\{\frac{\partial f}{\partial \mathbf{w}}\right\}$. Thus applying the operator to the gradient produced by the back-propagation algorithm gives the result we are looking for. Since the operator is a normal differential operator, it can be applied to all the steps in the back-propagation algorithm.

The algorithm is as follows. In the equations below, the vector $\mathbf{v}$ has been indexed in the same way as the weight vector. The forward pass:

$$\mathcal{R}_v\left\{x_i\right\} = \sum_j (w_{ji}\mathcal{R}_v\left\{y_i\right\} + v_{ji}y_j) \tag{A.5}$$

$$\mathcal{R}_v\left\{y_i\right\} = \mathcal{R}_v\left\{x_i\right\}\sigma_i'(x_i) \tag{A.6}$$

and the backward pass:

$$\mathcal{R}_v\left\{\tfrac{\partial f}{\partial w_{ij}}\right\} = y_i\mathcal{R}_v\left\{\tfrac{\partial f}{\partial x_j}\right\} + \mathcal{R}_v\{y_i\}\frac{\partial f}{\partial x_j} \tag{A.7}$$

$$\mathcal{R}_v\left\{\tfrac{\partial f}{\partial x_i}\right\} = \sigma_i'(x_i)\mathcal{R}_v\left\{\tfrac{\partial f}{\partial y_i}\right\} + \mathcal{R}_v\{x_i\}\sigma_i''(x_i)\frac{\partial f}{\partial y_i} \tag{A.8}$$

$$\mathcal{R}_v\left\{\tfrac{\partial f}{\partial y_i}\right\} = \frac{d^2 f}{dy_i^2}\mathcal{R}_v\{y_i\} + \sum_j(w_{ij}\mathcal{R}_v\left\{\tfrac{\partial f}{\partial x_j}\right\} + v_{ij}\frac{\partial f}{\partial x_j}) \tag{A.9}$$

where $\frac{d^2 f}{dy_i^2}$ is the second direct derivative of f with respect to the activation y_i.

The algorithm has a number of variables in common with back-propagation, a feature that can be used to speed up the calculation at a point in weight space for which the error gradient has already been found using backprop.

Appendix B

Inverted pendulum simulations

This appendix gives the details of the inverted pendulum simulations used to demonstrate the preceeding learning controllers.

Among the work done on the inverted pendulum system, the commonest value for the plant parameters are those introduced by Michie and Chambers (Michie and Chambers 1968), popularized by Barto et al. (Barto et al. 1983). These parameters are identified here as the 'Barto' parameters.

More recently, Geva and Sitte have attempted to re-cast the inverted pendulum problem in a form more appropriate to the current ability of learning controllers (Geva and Sitte 1993). The parameters they prescribe are identified here as the Geva parameters.

For experimental results to be compared with those of others, the Barto parameters should be used. Unfortunately, as Geva and Sitte remark, these parameters do not present a particularly difficult problem. The Geva parameters present a more difficult problem. However, little work has been done with these parameters.

B.1 System start state

The initial state of the inverted pendulum system for a trial varies between experimenters. The Barto and Geva start states are given in table B.1. Other experimenters sometimes used a randomized start state around the Barto start state. Jordan and Jacobs claim this is a bonus — the need for a pre-existing controller that can reset the state to a definite point is avoided (Jordan and Jacobs 1990). However, the requirement for random start states can also be a hindrance, since a sampling strategy that gives rich data may not be obvious.

The perturbed Barto state state was introduced for the disturbance-free control experiments in chapter 6 to avoid the upright equilibrium state.

B.2 Stopping criteria

An inverted pendulum is deemed to have collapsed once its state falls outside a prescribed region in state space. The allowed region in state space for the Barto and Geva parameters are given in

State variable	Barto	Perturbed Barto	Geva
θ	0 rad	1°	0.1 rad
$\dot{\theta}$	0 rad s^{-1}	0 rad s^{-1}	0.2 rad s^{-1}
x	0 m	0 m	1 m
$\dot{x}$	0 m s^{-1}	0 m s^{-1}	1 m s^{-1}

Table B.1: The Barto, perturbed Barto and Geva start states.

Barto	Geva
$-12^\circ < \theta < 12^\circ$	$-90^\circ < \theta < 90^\circ$
$-\infty^\circ\ \mathrm{s}^{-1} < \dot{\theta} < \infty^\circ\ \mathrm{s}^{-1}$	$-\infty^\circ\ \mathrm{s}^{-1} < \dot{\theta} < \infty^\circ\ \mathrm{s}^{-1}$
$-2.4\ \mathrm{m} < x < 2.4\ \mathrm{m}$	$-2.4\ \mathrm{m} < x < 2.4\ \mathrm{m}$
$-\infty\ \mathrm{m\ s}^{-1} < \dot{x} < \infty\ \mathrm{m\ s}^{-1}$	$-\infty\ \mathrm{m\ s}^{-1} < \dot{x} < \infty\ \mathrm{m\ s}^{-1}$

Table B.2: The Barto and Geva state bounds.

table B.2. They differ only in the range of the pendulum angle θ.

B.3 Derivation of the equations of motion

The inverted pendulum system is shown in figure B.1. The analysis below carries the moment of inertia of the pendulum through to the final equations, so that the same equations can be used for both a pendulum of distributed mass, and a pendulum with point mass at the end. The real pendulum system discussed in chapter 5 is best modelled as having point mass at its end.

The coordinate frame is shown in the figure. $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are mutually orthogonal unit vectors.

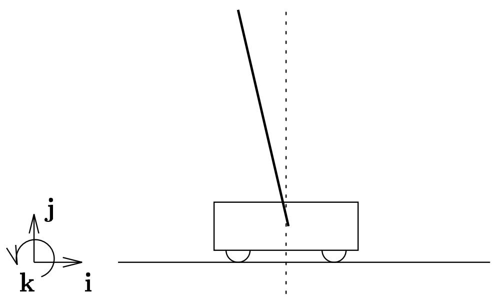


Figure B.1: The inverted pendulum system.

Consider the pendulum and trolley separately. Figure B.2 shows a pendulum AB. $\boldsymbol{\rho}$ is the position vector of the centre of mass of the pendulum, a distance l from point A, and $\mathbf{x} = x\mathbf{i}$ is the position vector of point A. $\mathbf{e}$ is a unit vector that points along AB. m is the mass of the pendulum. θ is the angle of the pendulum from the vertical, with positive angle indicated. $\mathbf{F}$ is the force on the pendulum from the trolley at the hinge. $\mathbf{g} = -g\mathbf{j}$ is the acceleration due to gravity. $m\ddot{\boldsymbol{\rho}}$ is the d'Alembert force due to the acceleration of the pendulum, and acts through the pendulum's centre of mass. μ_p is the friction coefficient about the hinge.

The geometry of the system gives us

$$\boldsymbol{\rho} = \mathbf{x} + l\mathbf{e}. \tag{B.1}$$

The forces give us

$$\mathbf{F} + m\mathbf{g} - m\ddot{\boldsymbol{\rho}} = 0. \tag{B.2}$$

The torques about A are given by:

$$-\mu_p\dot{\theta}\mathbf{k} - I\ddot{\theta}\mathbf{k} + ml\mathbf{e}\times\mathbf{g} - ml\mathbf{e}\times\ddot{\boldsymbol{\rho}} = 0 \tag{B.3}$$

where I is the moment of inertia of the pendulum about its centre of mass. For a pendulum with point mass, $I = 0$. For a pendulum of uniformly distributed mass, $I = \frac{ml^2}{3}$.

The forces on the trolley, shown in figure B.3, satisfy

$$-M\ddot{\mathbf{x}} - \mathbf{F} + \mathbf{U} + \mathbf{R} + M\mathbf{g} - \mu_r\mathrm{sign}\left(\dot{x}\right)\mathbf{i} = 0 \tag{B.4}$$

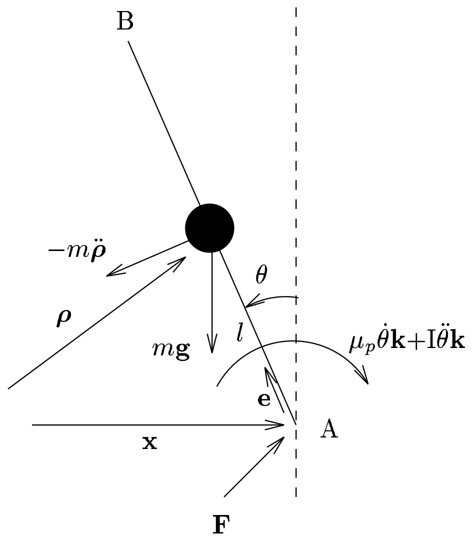

B
$-m\ddot{\boldsymbol{\rho}}$
θ
$\boldsymbol{\rho}$
$m\mathbf{g}$
l
$\mu_p\dot{\theta}\mathbf{k}+\mathrm{I}\ddot{\theta}\mathbf{k}$
$\mathbf{e}$
$\mathbf{x}$
A
$\mathbf{F}$

Figure B.2: The pendulum dynamics.

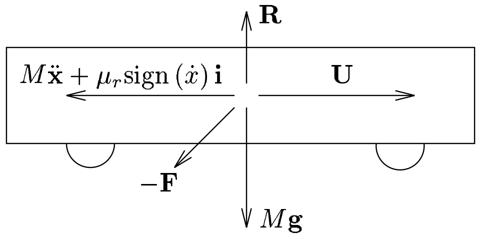

$\mathbf{R}$
$M\ddot{\mathbf{x}} + \mu_r \mathrm{sign}\,(\dot{x})\,\mathbf{i}$
$\mathbf{U}$
$-\mathbf{F}$
$M\mathbf{g}$

Figure B.3: The trolley forces.

where M is the mass of the trolley, $\mathbf{U} = U\mathbf{i}$ is the control force on the trolley, $\mathbf{R} = R\mathbf{j}$ is the reaction on the trolley from the ground, and $-\mu_r \text{sign}(\dot{x})\,\mathbf{i}$ is the rolling resistance on the trolley as a function of its linear velocity $\dot{x}$.

Equations B.1–B.4 contain the dynamics of the inverted pendulum system. Some manipulation will yield a more convenient form.

F, the force between the pendulum and the trolley, may be eliminated by adding equations B.4 and B.2:

$$-M\ddot{\mathbf{x}} + \mathbf{U} + \mathbf{R} + M\mathbf{g} - \mu_r \text{sign}(\dot{x})\,\mathbf{i} + m\mathbf{g} - m\ddot{\boldsymbol{\rho}} = 0. \tag{B.5}$$

R, the reaction from the track on the trolley, may be eliminated from B.5 by resolving along i:

$$-M\ddot{x} + U - \mu_r \text{sign}(\dot{x}) - m\ddot{\boldsymbol{\rho}}.\mathbf{i} = 0. \tag{B.6}$$

$\ddot{\boldsymbol{\rho}}$ may be eliminated by differentiating B.1 twice:

$$\ddot{\boldsymbol{\rho}} = \ddot{\mathbf{x}} + l\ddot{\mathbf{e}} \tag{B.7}$$

and substituting into B.3 and B.6:

$$\begin{aligned}
-\mu_p\dot{\theta}\mathbf{k} - I\ddot{\theta}\mathbf{k} + ml\mathbf{e}\times\mathbf{g} - ml\mathbf{e}\times\ddot{\mathbf{x}} - ml^2\mathbf{e}\times\ddot{\mathbf{e}} &= 0 &\text{(B.8)}\\
-M\ddot{x} + U - \mu_r \text{sign}(\dot{x})\,\mathbf{i} - m\ddot{x} - ml\ddot{\mathbf{e}}.\mathbf{i} &= 0. &\text{(B.9)}
\end{aligned}$$

It now remains to express **e** in equations B.8 and B.9 in terms of θ. The following results will be useful:

$$\begin{aligned}
\dot{\mathbf{e}} &= \dot{\theta}\mathbf{k}\times\mathbf{e} &\text{(B.10)}\\
\ddot{\mathbf{e}} &= \ddot{\theta}\mathbf{k}\times\mathbf{e} + \dot{\theta}\mathbf{k}\times(\dot{\theta}\mathbf{k}\times\mathbf{e}) &\text{(B.11)}\\
&= \ddot{\theta}\mathbf{k}\times\mathbf{e} - \dot{\theta}^2\mathbf{e} &\text{(B.12)}\\
\mathbf{e}\times\ddot{\mathbf{e}} &= \mathbf{e}\times(\ddot{\theta}\mathbf{k}\times\mathbf{e} - \dot{\theta}^2\mathbf{e}) &\text{(B.13)}\\
&= \ddot{\theta}\mathbf{k}. &\text{(B.14)}
\end{aligned}$$

Since $\mathbf{e} = -\sin\theta\mathbf{i} + \cos\theta\mathbf{j}$,

$$\begin{aligned}
\mathbf{e}\times\mathbf{g} &= g\sin\theta\mathbf{k} &\text{(B.15)}\\
\mathbf{e}\times\ddot{\mathbf{x}} &= -\ddot{x}\cos\theta\mathbf{k} &\text{(B.16)}\\
\ddot{\mathbf{e}}.\mathbf{i} &= (\ddot{\theta}\mathbf{k}\times\mathbf{e}).\mathbf{i} - \dot{\theta}^2\mathbf{e}.\mathbf{i} &\text{(B.17)}\\
&= \ddot{\theta}\mathbf{k}.(\mathbf{e}\times\mathbf{i}) + \dot{\theta}^2\sin\theta &\text{(B.18)}\\
&= \ddot{\theta}\mathbf{k}.(-\cos\theta\mathbf{k}) + \dot{\theta}^2\sin\theta &\text{(B.19)}\\
&= -\ddot{\theta}\cos\theta + \dot{\theta}^2\sin\theta. &\text{(B.20)}
\end{aligned}$$

Substituting into equations B.8 and B.9:

$$\begin{aligned}
-\mu_p\dot{\theta}\mathbf{k} - I\ddot{\theta}\mathbf{k} + mlg\sin\theta\mathbf{k} + ml\ddot{x}\cos\theta\mathbf{k} - ml^2\ddot{\theta}\mathbf{k} &= 0 &\text{(B.21)}\\
-M\ddot{x} + U - \mu_r \text{sign}(\dot{x}) - m\ddot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta &= 0. &\text{(B.22)}
\end{aligned}$$

Hence:

$$\begin{aligned}
\ddot{\theta} &= \frac{\frac{-\mu_p\dot{\theta}}{ml} + g\sin\theta + \frac{\cos\theta}{M+m}\left\{U - \mu_r \text{sign}(\dot{x}) - ml\dot{\theta}^2\sin\theta\right\}}{l\left\{\frac{I}{ml^2} + 1 - \cos^2\theta\frac{m}{M+m}\right\}} &\text{(B.23)}\\
\ddot{x} &= \frac{U - \mu_r \text{sign}(\dot{x}) + ml\left(\ddot{\theta}\cos\theta - \dot{\theta}^2\sin\theta\right)}{M+m}. &\text{(B.24)}
\end{aligned}$$

Parameter	Barto	Geva
Pendulum mass m	1.0 kg	0.1 kg
Pivot to pendulum mass centre l	0.5 kg	0.5 kg
Inertia I	$ml^2/3$	$ml^2/3$
Pivot friction μ_p	2×10^{-6} kg m^2 s^{-1}	0 kg m^2 s^{-1}
Trolley mass M	1.0 kg	1.0 kg
Rolling friction μ_r	5×10^{-4} kg m s^{-2}	0 kg m s^{-2}
Gravity g	9.81 m s^{-2}	9.81 m s^{-2}
Euler integration step	0.02 s	0.02 s

Table B.3: The Barto and Geva parameters for the equations of motion.

k_x	1 N m^{-1}
$k_{\dot{x}}$	2 N s m^{-1}
k_θ	-19 N
$k_{\dot{\theta}}$	-7 N s

Table B.4: Linear state feedback gains for the simulated system with the Barto parameters, and the sign convention given in figure B.1. The values justify the scope of the search space for the random search of section 6.3.1.

B.4 The equation parameters

The Barto and Geva simulation parameters differ over their inclusion of friction (Barto et al. 1983). Barto et al. justify the parameters by appealing for realism. Geva and Sitte suggest that the friction effects are negligible in the equations, and make the problem easier by introducing damping (Geva and Sitte 1993). The Barto and Geva parameters for the equations of motion are given in table B.3.

Appendix C

Hardware

Figure C.1 shows the configuration of the experimental apparatus, which was designed and built within Cambridge University Engineering Department. Table C.1 details some constants for the hardware. An important feature of the apparatus was its robustness, a great advantage during the long experiment runs.

The force required to move the carriage along the track at constant low velocity ranged between 1 and 3 N, an indication of the size and non-uniformity of the friction along its length.

To protect the equipment, a hardware cut-out on the rig stopped the carriage smashing into the end-stops of the track. Unfortunately, once the cut-out was activated, the rig was beyond the control of the transputer, so a safety margin was introduced into the software to ensure the plant stayed within safe operating bounds. This further reduced the available length of track, and explains why only 0.6 m of the 1 m track was used for training.

Figure C.2 shows a picture of the apparatus. The analogue power amplifier and sensor amplifiers are in the box behind the track on the left, the transputer and interfaces are in the box on the right. The scale is indicated by a 300 mm ruler under the pendulum in the centre of the track. The pendulum is shown held upright by a linear state feedback controller.

Symbol	Description	Value
L	Working track length	0.80 m
l	Pivot to pendulum centre of mass distance	0.125 m
a	Pulley radius	0.016 m
m	Pendulum mass	0.230 kg
M	Trolley mass	0.700 kg
I	Motor shaft's moment of inertia	80×10^{-6} kg m^2
$\mathrm{k_m}$	Torque motor constant	0.080 Nm A^{-1}
$\mathrm{k_a}$	Amplifier constant	-0.50 A V^{-1}

Table C.1: Physical constants of the apparatus

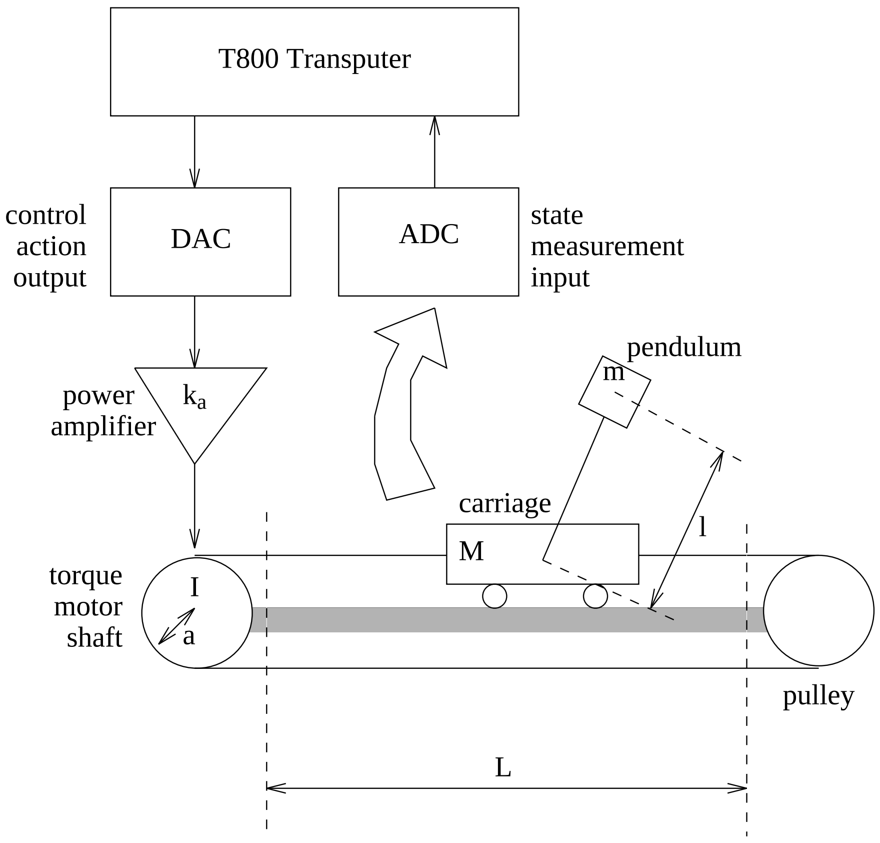


Figure C.1: Hardware configuration

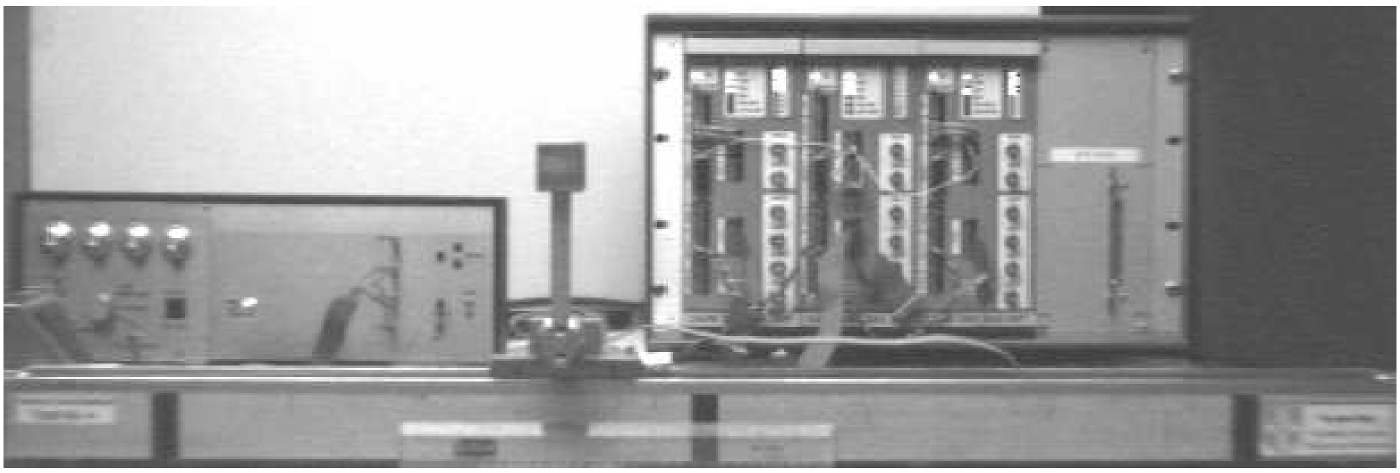


Figure C.2: The inverted pendulum hardware.

Appendix D

Decoder partitions

Several state space partitioning schemes were used in the experimental runs on the pendulum hardware. Table D.1 details the partitions used in the first balancing attempt, which failed to work. This partitioning differs from the original work of Barto et al. by only a linear scaling of the carriage position partitions (the simulated system had carriage bounds at +/- 2.4 m, the real apparatus had carriage bounds at +/- 0.3 m) (Barto et al. 1983).

Tables D.2, D.3 and D.4 give the partition positions for the 1296, 864 and 750 decoders discussed in chapter 5.

Number of boxes	162
Pole position partitions	
1	$-6°$
2	$-1°$
3	$0°$
4	$1°$
5	$6°$
Pole velocity partitions	
1	$-50°$ s^{-1}
2	$50°$ s^{-1}
Carriage position partitions	
1	-0.1 m
2	0.1 m
Carriage velocity partitions	
1	-0.5 m s^{-1}
2	0.5 m s^{-1}

Table D.1: 162 box decoder parameters

Number of boxes	1296
Pole position partitions	
1	$-6°$
2	$-2°$
3	$0°$
4	$2°$
5	$6°$
Pole velocity partitions	
1	$-68°$ s^{-1}
2	$-23°$ s^{-1}
3	$0°$ s^{-1}
4	$23°$ s^{-1}
5	$68°$ s^{-1}
Carriage position partitions	
1	-0.22 m
2	-0.07 m
3	0 m
4	0.07 m
5	0.22 m
Carriage velocity partitions	
1	-0.27 m s^{-1}
2	-0.09 m s^{-1}
3	0 m s^{-1}
4	0.09 m s^{-1}
5	0.27 m s^{-1}

Table D.2: 1296 box decoder parameters

Number of boxes	864
Pole position partitions	
1	$-6°$
2	$-2°$
3	$0°$
4	$2°$
5	$6°$
Pole velocity partitions	
1	$-68°$ s^{-1}
2	$-23°$ s^{-1}
3	$0°$ s^{-1}
4	$23°$ s^{-1}
5	$68°$ s^{-1}
Carriage position partitions	
1	-0.15 m
2	0 m
3	0.15 m
Carriage velocity partitions	
1	-0.27 m s^{-1}
2	-0.09 m s^{-1}
3	0 m s^{-1}
4	0.09 m s^{-1}
5	0.27 m s^{-1}

Table D.3: 864 box decoder parameters

Number of boxes	750
Pole position partitions	
1	$-6°$
2	$-2°$
3	$0°$
4	$2°$
5	$6°$
Pole velocity partitions	
1	$-45°$ s^{-1}
2	$-11°$ s^{-1}
3	$11°$ s^{-1}
4	$45°$ s^{-1}
Carriage position partitions	
1	-0.15 m
2	-0.04 m
3	0.04 m
4	0.15 m
Carriage velocity partitions	
1	-0.18 m s^{-1}
2	-0.05 m s^{-1}
3	0.05 m s^{-1}
4	0.18 m s^{-1}

Table D.4: 750 box decoder parameters

Appendix E

Linearized state feedback

Analysis of the pendulum system shown in figure E.1 gives the following equations, ignoring friction:

$$
\begin{aligned}
l\ddot{\theta} &= g\sin\theta - \ddot{x}\cos\theta && \text{(E.1)}\\
\left\{\frac{M}{m} + \frac{I}{ma^2} + 1\right\}\ddot{x} &= \frac{T}{ma} - l\ddot{\theta}\cos\theta + l\dot{\theta}^2\sin\theta && \text{(E.2)}
\end{aligned}
$$

where l is the length of the pendulum, θ is the angle of the pendulum from the vertical, g is the acceleration due to gravity, x is the position of the carriage along the track, M is the mass of the carriage, m is the mass of the pendulum, a is the radius of the torque motor shaft, I is the moment of inertia of the torque motor about its axis, and T is the torque supplied by the motor. Signs for the quantities are in the sense depicted in figure E.1.

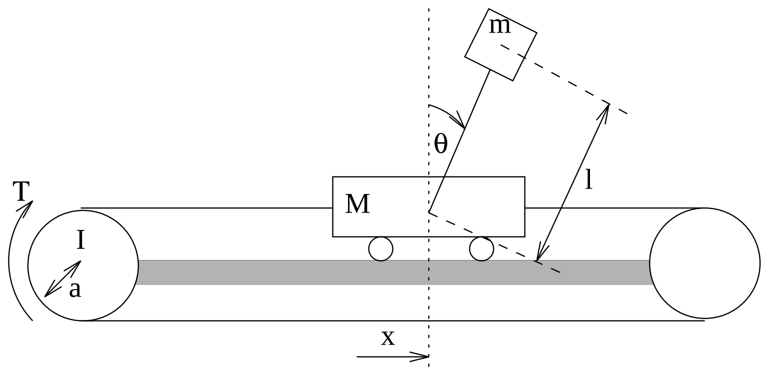


Figure E.1: The pendulum system dynamics.

Linearizing equations (E.1) and (E.2) for small values of θ gives

$$
\begin{aligned}
l\ddot{\theta} &= g\theta - \ddot{x} && \text{(E.3)}\\
\left\{\frac{M}{m} + \frac{I}{ma^2} + 1\right\}\ddot{x} &= \frac{T}{ma} - l\ddot{\theta} && \text{(E.4)}
\end{aligned}
$$

which may be re-arranged and expressed in matrix form:

$$
\frac{d}{dt}\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{g}{J-1} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{gJ}{l(J-1)} & 0 \end{bmatrix}\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ma(J-1)} \\ 0 \\ \frac{1}{lma(J-1)} \end{bmatrix} T \tag{E.5}
$$

k_x	1.15 N
$k_{\dot{x}}$	0.93 N s
k_θ	2.4 N m
$k_{\dot{\theta}}$	0.21 N m s

Table E.1: Linearized state feedback gains

where

$$J = \left\{ \frac{M}{m} + \frac{I}{ma^2} + 1 \right\}. \tag{E.6}$$

Linear state feedback fixes the torque in equation (E.5) to be a linear combination of the elements of the state vector $\mathbf{x}$, where

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} \tag{E.7}$$

Thus T is set to

$$T = \begin{bmatrix} k_x \\ k_{\dot{x}} \\ k_\theta \\ k_{\dot{\theta}} \end{bmatrix}^T \mathbf{x} \tag{E.8}$$

and equation (E.5) can be expressed in the form

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}. \tag{E.9}$$

The positions of the eigenvalues of $\mathbf{A}$ in the complex plane characterize the stability of the linearized system. So long as the eigenvalues are in the left half plane the linear system is globally asymptotically stable. A range of values for the elements of $\mathbf{k}$ results in suitable eigenvalue placement. The values of $\mathbf{k}$ chosen to set the decoder partitions of chapter 5 are given in table E.1.

Bibliography

Albus, J. (1975). A new approach to manipulator control: The cerebellar model articulation controller, *Transactions of the ASME* pp. 220–227.

Anderson, C. W. (1989). Learning to control an inverted pendulum using neural networks, *IEEE Control Systems Magazine* pp. 31–37.

Anderson, C. W. (1993). Q-learning with hidden-unit restarting, *Advances in Neural Information Processing Systems 5*, Morgan Kaufmann, San Mateo, California. to appear.

Åström, K. and Wittenmark, B. (1989). *Adaptive Control*, Addison-Wesley series in electrical engineering: control engineering, Addison-Wesley, Reading, Massachusettes.

Barto, A., Bradtke, S. and Singh, S. (1991). Real-time learning and control using asynchronous dynamic programming, *Technical report*, Department of Computer Science, University of Massachusetts, Amherst MA 01003, U.S.A.

Barto, A. G., Bradtke, S. J. and Singh, S. P. (1993). Learning to act using real-time dynamic programming. Submitted to AI Journal, special issue on Computational Learning Theories of Interaction and Agency.

Barto, A. G., Sutton, R. S. and Watkins, C. J. (1990). Learning and sequential decision making, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, The MIT Press, Cambridge, Massachusetts, pp. 539–602.

Barto, A., Sutton, R. and Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems, *IEEE Transactions on Systems, Man and Cybernetics* **SMC–13**(5): 834–846.

Beale, E. (1971). A derivation of conjugate gradients, *in* F. Lootsman (ed.), *Numerical Methods for Non-linear Optimisation*, Academic Press, pp. 39–43. Conference sponsored by the Science Research Council, University of Dundee, Scotland.

Bellman, R. (1957). *Dynamic Programming*, Princeton University Press, Princeton, New Jersey.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching, *Communications of the ACM* **18**(9): 509–517.

Berry, D. A. and Fristedt, B. (1985). *Bandit Problems: sequential allocation of experiments*, Monographs on statistics and applied probability, Chapman and Hall, London.

Bertsekas, D. P. (1987). *Dynamic Programming — Deterministic and Stochastic Models*, Prentice-Hall.

Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic estimation, *in* C. Giles, S. Hanson and J. Cowan (eds), *Advances in Neural Information Processing Systems 5*, Morgan Kaufmann, San Mateo, California. to appear.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition, *in* F. F. Soulié and J. Hérault (eds), *Neurocomputing*, Vol. F68 of *NATO ASI Series*, Springer-Verlag, pp. 227–236.

Brody, C. (1992). Fast learning with predictive forward models, *in* J. Moody, S. Hanson and R. Lippmann (eds), *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann.

Fahlman, S. E. (1988). An empirical study of learning speed in back-propagation networks, *Technical Report CMU-CS-88-162*, Carnegie Mellon University.

Feldbaum, A. (1961). Dual-control theory I, *Automation and Remote Control* **21**(9): 874–880. Translation of “Avtomatika i telemekhanika”, Academy of Sciences, Moscow, USSR, September 1960.

Geva, S. and Sitte, J. (1993). A cartpole experiment benchmark for trainable controllers, *IEEE Control Systems* **13**(5): 40–51.

Hecht-Nielsen, R. (1990). *Neurocomputing*, Addison-Wesley Publishing Company, Inc., Reading, Massachusettes.

Hertz, J. A., Palmer, R. G. and Krogh, A. S. (1991). *Introduction to the Theory of Neural Computation*, Addison-Wesley, 350 Bridge Parkway, Redwood City, CA 94065, U.S.A.

Howard, R. (1960). *Dyanmic Programming and Markov Processes*, M.I.T. Press, Cambridge, Massachusettes.

Hunt, K. J., Sbarbaro, D., Zbikowski, R. and Gawthrop, P. J. (1992). Neural networks for control systems: a survey, *Automatica* **28**(6): 1083–1112.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation, *Neural Networks* **1**: 295–307.

Jang, J.-S. R. (1992). Self-learning fuzzy controller based on temporal back-propagation, *IEEE Transactions on Neural Networks* **3**(5): 714–723.

Jenkins, R. E. and Yuhas, B. P. (1992). A simplified neural-network solution through problem decomposition: The case of the truck backer-upper, *Neural Computation* **4**: 647–649.

Jordan, M. I. and Jacobs, R. A. (1990). Learning to control an unstable system with forward modeling, *in* D. Touretzky (ed.), *Advances in Neural Information Processing Systems 2*, Morgan Kaufmann, San Mateo, California.

Kadirkamanathan, V. (1991). *Sequential Learning in Artificial Neural Networks*, PhD thesis, Cambridge University Engineering Department, Trumpington Street, Cambridge, England.

Khalid, M. and Omatu, S. (1992). A neural network controller for a temperature control system, *IEEE Control Systems Magazine* pp. 58–64.

MacKay, D. (1991). *Bayesian Methods for Adaptive Models*, PhD thesis, California Institute of Technology, Pasadena, California. Also in Neural Computation, 1992, vol. 4: no. 3, pp 415–447, 448–472; no. 5, pp 698–714.

MacKay, D. (1994). Bayesian non-linear modelling for the 1993 energy prediction competition, *in* G. Heidbreder (ed.), *Maximum Entropy and Bayesian Methods, Santa Barbara 1993*, Kluwer, Dordrecht.

Mel, B. W. (1989). Murphy: A connectionist approach to vision-based robot motion planning, *Technical report*, Department of Computer Science and Center for Complex Systems Research, Beckman Institute, University of Illinois, 405 North Mathews Street, Urbana, IL 61801, U.S.A.

Michie, D. and Chambers, R. (1968). Boxes: An experiment in adaptive control, *in* E. Dale and D. Michie (eds), *Machine Intelligence*, Vol. 2, Oliver and Boyd, pp. 137–152.

Miller III, W. T. (1989). Real-time application of neural networks for sensor-based control of robots with vision, *IEEE Transactions on Systems, Man and Cybernetics* **SMC-19**: 825–831.

Møller, M. (1993). A scaled conjugate gradient algorithm for fast supervised learning, *Neural Networks* **6**: 525–533.

Moore, A. M. and Atkeson, C. G. (1992). An investigation of memory-based function approximators for learning control, *Technical report*, MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA 02139.

Moore, A. W. (1991). Knowledge of knowledge and intelligent experimentation for learning control, *Proceedings of the International Joint Conference on Neural Networks, Seattle.*

Moore, A. W. (1992). Fast, robust adaptive control by learning only forward models, *in* J. Moody, S. Hanson and R. Lippmann (eds), *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann.

Munro, P. W. (1987). A dual back-propagation scheme for scalar reinforcement learning, *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Seatle, WA.

Narendra, K. S. and Thathachar, M. A. (1989). *Learning Automata: An Introduction*, Prentice Hall International, London.

Nguyen, D. and Widrow, B. (1989). The truck backer-upper: An example of self-learning in neural networks, *Proceedings of the International Joint Conference on Neural Networks, Volume 2*, Washington D.C., pp. 357–363.

Nguyen, D. and Widrow, B. (1990). Neural networks for self-learning control systems, *IEEE Control Systems Magazine* pp. 18–23.

Pearlmutter, B. A. (1993). Fast exact multiplication by the Hessian, To appear in *Neural Computation.*

Press, W. H., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T. (1992). *Numerical Recipes in C*, 2 edn, Cambridge University Press.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The RPROP algorithm, *in* H. Ruspini (ed.), *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, pp. 586–591.

Robinson, A. (1994). An application of recurrent nets to phone probability estimation, *IEEE Transactions on Neural Networks.* To appear in March.

Robinson, A. and Fallside, F. (1987). The utility driven dynamic error propagation network, *Technical Report CUED/F-INFENG/TR.1*, Cambridge University Engineering Department, Trumpington Street, Cambridge, England.

Rumelhart, D. E. and McClelland, J. L. (1986). *Parallel Distributed Processing*, Vol. 1, The MIT Press, Cambridge, Mass., U.S.A.

Rumelhart, D. E., Hinton, G. and Williams, R. (1986). Learning internal representations by error propagation, *in* D. E. Rumelhart, J. L. McClelland and the PDP Research Group (eds), *Parallel Distributed Processing*, Vol. 1, The MIT Press, Cambridge, Mass., U.S.A., pp. 318–364.

Samuel, A. (1963). Some studies in machine learning using the game of checkers, *in* E. Feigenbaum and J. Feldman (eds), *Computers and Thought*, McGraw-Hill, New York, pp. 71–105.

Schiffmann, W., Joost, M. and Werner, R. (1992). Optimization of the backpropagation algorithm for training multilayer perceptrons, *Technical report*, University of Koblenz, Institut für Physics, Rheinau 3–4, D-5400 Koblenz, Germany.

Sharp, R. and Crolla, D. (1987). Road vehicle suspension system design — a review, *Vehicle System Dynamics* **16**: 167–192.

Tesauro, G. (1991). Practical issues in temporal difference learning, *Technical report*, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

Tham, C. K. and Prager, R. W. (1992). Reinforcement learning for multi-linked manipulator control, *Technical Report CUED/F-INFENG/TR 104*, Cambridge University Engineering Department, Trumpington Street, Cambridge, England.

Thodberg, H. H. (1993). Ace of Bayes: Application of neural networks with pruning, *Technical report*, The Danish Meat Research Institute, Maglegaardsvej 2, DK-4000 Roskilde, Denmark.

Thrun, S. B. (1992). The role of exploration in learning control, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, Van Nostrand Reinhold, Florence, Kentucky 41022.

Törn, A. and Žilinskas, A. (1987). *Global Optimization*, Lecture notes in computer science, Springer-Verlag.

Tzirkel-Hancock, E. (1992). *Stable Control of Nonlinear Systems using Neural Networks*, PhD thesis, Cambridge University Engineering Department, Trumpington Street, Cambridge, England.

van der Smagt, P. P. (1991). A real-time learning neural robot controller, *in* T. Kohonen, K. Mäkisara, O. Simula and J. Kangas (eds), *Artificial Neural Networks*, Elsevier Science Publishers B.V. (North Holland), pp. 351–356.

van Luenen, W., de Jager, P., van Amerongen, J. and Franken, H. (1993). Limitations of adaptive critic control schemes, *Technical report*, Control Laboratory, Department of Electrical Engineering, University of Twente, P.O. Box 217, 7500 AE Enschede, Netherlands. submitted to the International Conference on Artificial Neural Networks, Amsterdam, Netherlands, September 93.

Watkins, C. J. (1989). *Learning from Delayed Rewards*, PhD thesis, King's College, Cambridge.

Werbos, P. J. (1990). Consistency of H.D.P. applied to a simple reinforcement learning problem, *Neural Networks* **3**: 179–189.

Werbos, P. J. (1991). An overview of neural networks for control, *IEEE Control Systems Magazine* pp. 40–41.

Williams, P. M. (1991). A Marquardt algorithm for choosing the step-size in backpropagation learning with conjugate gradients, *Technical Report CSRP 229*, University of Sussex at Brighton, England.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning, *Machine Learning* **8**: 229–256.

Williams, R. J. and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks, *Neural Computation* **1**: 270–280.