Recurrent Neural Networks and Adaptive Motor Control

Paul Ian Miller

Submitted in partial fulfillment of the requirements for
the degree of

Doctor of Philosophy

at

University of Stirling

September 1997

Department of Psychology
University of Stirling
Scotland

Acknowledgements

Thanks to Iain Paterson for too many things to enumerate: discussions about all aspects of science, help with maths and programming, writing up, and curing a really bad slice. A more genuine and generous person you will not meet.

Thanks to Peter Hancock and Kevin Swingler for helpful discussions and advice; to Will Goodall for being Will Goodall; to Toni Zawadski for enthusiastic discussions of everything, and to Manoli Stamatakis for his kindness and encouragement.

Paul Toombs has helped with numerous aspects of this PhD, from Latex and Postscript nightmares, to programming and pointers to literature. But more than that, Paul has been a huge source of encouragement, enthusiasm and discussion, about the work, and everything else. When I needed intelligent conversation, and an infectious sense of humour, he was always there.

I would also like to thank Leslie Smith for help with many aspects of this thesis, and for insightful discussions on the complexities of recurrent networks.

Thanks to Ben Craven for all our discussions - Ben has the ability to get to the fundamentals of things, whether it's Maths, Science or Poker. Ben has made me constantly question and rethink the assumptions implicit in any research, and I thank him for that.

There are many people who have made my time at Stirling more enjoyable, but special thanks go to Roisin Ash, Trish Carlin, Graham Dyson, Iain Paterson and Dario Floreano, just for being the lovely people they are.

Thanks to Mike Burton for comments on an earlier draft, for much support and encouragement, and for being the perfect boss.

All the programming for this work was done with the support of Billy Corgan.

Finally, I must thank Fiona. Fiona has given all a person can give - unconditional love, kindness, encouragement, understanding and a lot of laughs. To have done this while finishing her own PhD makes her what she is: my best friend. Cheers, doll.

Contents

Abstract

This thesis is concerned with the use of neural networks for motor control tasks. The main goal of the thesis is to investigate ways in which the *biological* notions of motor programs and Central Pattern Generators (CPGs) may be implemented in a neural network framework. Biological CPGs can be seen as components within a larger control scheme, which is basically modular in design. In this thesis, these ideas are investigated through the use of modular recurrent networks, which are used in a variety of control tasks

The first experimental chapter deals with learning in recurrent networks, and it is shown that CPGs may be easily implemented using the machinery of backpropagation. The use of these CPGs can aid the learning of pattern generation tasks; they can also mean that the other components in the system can be reduced in complexity, say, to a purely feedforward network. It is also shown that incremental learning, or 'shaping' is an effective method for building CPGs. Genetic algorithms are also used to build CPGs; although computational effort prevents this from being a practical method, it does show that GAs are capable of optimising systems that operate in the context of a larger scheme. One interesting result from the GA is that optimal CPGs tend to have unstable dynamics, which may have implications for building modular neural controllers.

The next chapter applies these ideas to some simple control tasks involving a highly redundant simulated robot arm. It was shown that it is relatively straightforward to build CPGs that represent elements of pattern generation, constraint satisfaction, and local feedback. This is indirect control, in which errors are backpropagated through a plant model, as well as the CPG itself, to give errors for the controller.

Finally, the third experimental chapter takes an alternative approach, and uses direct

control methods, such as reinforcement learning. In reinforcement learning, controller outputs have unmodelled effects; this allows us to build complex control systems, where outputs modulate the couplings between sets of dynamic systems. This was shown for a simple case, involving a system of coupled oscillators. A second set of experiments investigates the use of simplified models of behaviour; this is a reduced form of supervised learning, and the use of such models in control is discussed.

1 Neural Networks and Control

1.1 Introduction

This thesis in concerned with the use of neural networks for motor control tasks. Research in this area typically draws from two bodies of work: studies of the control strategies used in biological systems, and nonlinear control theory. The first of these is the subject of the next chapter; this chapter attempts to survey the field of nonlinear motor control, and neural networks. Biological systems are of interest to robotics and motor control theorists, for the simple reason that animals are the only existence proof that difficult motor control tasks can be solved at all. Understanding the functioning of biological systems may be essential in building a computational theory of motor tasks, and is certainly a rich source of information about possible architectures for control problems.

In contrast to most research in neural networks, biological systems are often highly structured, with sets of components performing different roles, and interconnections co-ordinating their behaviour. The main goal of this thesis is to investigate the design and use of such structured systems in neural networks and motor control. The goal is not to model the fine-scale detail of individual systems, but rather to capture the broad structure seen across a wide range of species. These structured controllers are implemented using existing neural net methods for learning control, and applied to some simple, simulated motor control tasks.

The use of neural nets in control draws from several fields - Artificial Intelligence, Statistics, Linear and Optimal Control Theory. Each of these areas has their own history and literature, and a full treatment of the combined field would encompass a vast amount of literature. Consequently, this literature review attempts to give a conceptual overview

of the area.

A control system acts by sending control signals to an object, referred to as the 'plant', which then acts on its environment. The control signals are generally a function of some inputs, which may represent the current state of the plant, as well as information specifying the desired plant behaviour. The field of control systems has a rich theoretical history, which begins with the study of linear dynamic systems, and feedback controllers.

1.2 Linear Control and Adaptive Methods

Linear control refers to the use of linear feedback elements to control a system, itself assumed to be linear. The plant is a dynamic system, i.e. a system described by a set of linear differential equations (or difference equations for systems that are discrete in time). Linear control theory is an area with an enormous amount of literature, which reflects the advanced state of development of linear systems theory in general.

If the plant is linear, then it may be compactly described by a matrix operation:

$$\dot{y} = Au(t) + By(t) \tag{1.1}$$

or its equivalent in discrete time. y is the plant state, u is the control input, and the matrix A is the transfer function of the plant, and B relates the control inputs the changes in plant state (Barnett and Cameron, 1985).

The dots are taken to mean the derivative with respect to time. 'Plant state' refers to all the variables which make up the plant; knowledge of their values allows the future behaviour to be predicted. The transfer function describes how the plant's rate of state change depends on its current state.

One of the interesting results from this formulation is that the entire behaviour of the system is given by the eigenvectors of A and B; these are called the 'modes' of the system. A linear feedback system is achieved by making the control inputs u a linear function of the output y. This can only produce another linear system $\dot{x} = Cx(t)$, and the eigenvectors of this new system are given entirely by the eigenvectors of the plant itself, and the linear mapping of the controller. These two facts are related - if the characteristics of the plant are known (i.e. its eigenvectors), then a controller may be *designed* to produce

the desired behaviour, assuming such a controller exists. As linear differential equations may be analytically solved, the behaviour of the coupled controller-plant system may be determined analytically. This is only true if the feedback loop remains constant, but most of the interest in control lies in the ability of the controller to *adapt* its parameters so as to improve performance. This may be because the characteristics of the plant change in time, and it is desired that the controller is able to track those changes. It may also be seen as an attempt to automate the design of the controller. More significantly, the use of adaptive methods corresponds to the desire to control plants that are only partly specified, implying that a controller cannot be designed with analytic methods. This clearly becomes more important as the complexity of the plant increases.

If the parameters of the controller are allowed to vary in time, then the combined controller-plant system will in general be nonlinear. Most of the research effort in adaptive control has been directed towards building systems that are known to be convergent, as instability is always a key issue in nonlinear dynamic systems. Instability corresponds to nonconvergence; this may mean the system wanders away from its desired state; more seriously, it may refer to the system making larger and larger oscillations. Historically, much of the work on adaptive control was conducted in the area of autopilots, and regulators for industrial plants. In such cases, instability is a real danger to the system. This is also true, to a lesser extent, to the control of robots - instability and oscillations always cause excessive wear and poor performance on the control task.

There are a wide variety of methods for adapting the controller's parameters, some making use of models of the plant, and others using direct methods - see (Widrow and Stearns, 1985) for an excellent introduction. One important idea to come out of adaptive methods is the notion of *intelligent control.* Intelligent control may be understood by considering the simple example of a human learning to drive an unfamiliar car. Part of this problem lies in the fact that the 'weight' of the controls are unknown - the car may have power steering, in which case small turning forces should be used, or the brakes may be power assisted. Learning to drive the car is a process of adaptive control, as the control actions have to be matched against the weights of the steering and brakes. Now imagine the driver switches to a different vehicle - for example, a truck. The weights of the controls

are likely to be completely different, so an adaptive process is again involved. The key point about intelligent control concerns what happens if the driver now returns to the car. In adaptive control, the driver now has to re-adapt to the car, as the first process of adaptation has been undone by learning to drive the truck. *Intelligent Control* refers to the (desired) ability to recognise different situations, and select the best control policy for the current circumstances. This involves such problems as pattern recognition and intelligent decision making, and so starts to approach the notion of intelligent behaviour in humans.

The use of adaptive and intelligent methods find their greatest utility when the assumptions of linearity break down - i.e. in the control of nonlinear systems.

1.3 Nonlinear Systems: Feedforward and Feedback Control

Control of nonlinear systems is substantially more difficult than linear control. This reflects the fact that generally, nonlinear differential equations cannot be analytically solved, but have to be treated on a case-by-case basis. The methods for building nonlinear controllers are correspondingly more complex. One obvious strategy is to approximate the full nonlinear plant with a linear system, and then proceed as before. The success of this approach clearly depends on the quality of the approximation. For example, it may be known that the plant is convergent, so that its dynamics are such that it always converges to a fixed point. In that case, its behaviour in the immediate region surrounding the equilibrium point may be nearly linear. This corresponds to taking a Taylor series expansion of the system's transfer function, and neglecting all but the first term. The magnitude of the discarded terms gives a measure of the accuracy of the approximation. Clearly, this approach is only going to be applicable in some cases, and it may be necessary to have more than one model to approximate the behaviour of the plant in different parts of its state space.

A general nonlinear system is described by the generalisation of 1.1:

$$\dot{y} = F(u(t), y(t))$$

where F is some nonlinear function, u is again the control inputs, and y is the state

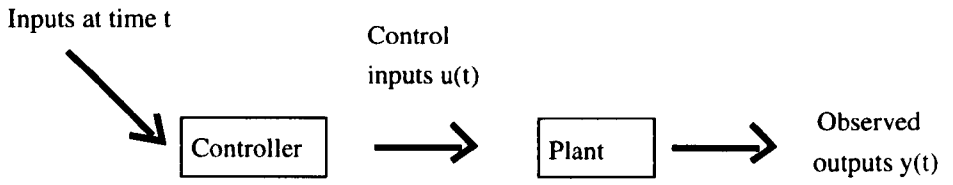

FIGURE 1.1. A general control task

of the plant. This is a completely general framework for describing dynamic systems. Different forms of the transfer function F give rise to plants with widely varying dynamics, from systems that converge onto fixed points, those that show oscillatory behaviour, even chaotic systems. This lack of a *priori* knowledge of the plants dynamics makes the control task a very difficult one.

1.4 Learning Control

Fig 1.1 shows a generic situation for learning control. As before, the plant is the system to be controlled, and the learning system plays the part of the controller. The controller gives an output drawn from set of allowed control actions. The state of the plant goes into some measurement process, which then gives the available information about the plant's behaviour. This measurement process may be noisy; it may also be lossy in the more general sense that some aspects of the plant state may not be able to be measured at all. In the case of feedback control, the inputs to the controller will include some information about the time-varying plant state. Feedforward (open-loop) control implies that control is made without reference to the plant state. The task is usually defined by a desired state for the plant (time series of desired states); these are called setpoints.

As with linear systems, feedback control may be used with nonlinear plants. A feedback controller takes as its input the discrepancy between the desired and actual plant state, and maps this onto the controls. This makes the system behave somewhat as a spring, with the desired state acting as the spring's equilibrium position. As springs are notorious for exhibiting oscillations, it is usual to include a damping term. There is often an additional integrative term that is designed to compensate for steady-state offsets in the plant state,

for example, the effect of gravity on a robot arm. These three terms correspond to the PID controller (Proportional, Integrative, Damped).

The main attraction of purely feedback techniques is their simplicity - the controller continuously seeks to remove the discrepancy between the current and desired state, without detailed knowledge of the dynamics of the plant. The main disadvantage of purely feedback control is that this is an error-driven strategy, so the plant needs to measurably deviate from the desired state in order for the controller to do anything at all. The speed at which corrections are made is given by the *gain* of the controller, i.e. the ratio of its outputs to its inputs. A high gain is necessary to quickly eliminate errors, but a high gain will also result in instability if there are measurement delays. This fact is particularly relevant for biological control systems, where most sensory inputs are subject to a transmission delay.

By contrast, a purely feedforward controller produces outputs without reference to the current state of the plant. How does the controller know which outputs to produce? One way is to define the controller as an *inverse plant model.* The plant has been described as a nonlinear mapping, from control inputs to changes in state (or equivalently, to the state at the next time step). Ideally, the controller would take some coding of the desired state of the system, and output the controls that cause the plant to move into the desired state:

$$u = G(y(t), y^*(t))$$

so the controller takes the current state y, and the desired state y^*, and outputs the corresponding control. In effect, this requires inverting the transfer function of the plant: $G(.) = F^{-1}(.)$. Finding the inverse dynamics is not straightforward, and usually involves the use of a *forward* plant model, i.e. an approximation of the transfer function F. Nevertheless, this framework gives a way of understanding the role of the controller as that of *undoing* the effects of the plant dynamics.

The way in which the controller is actually used depends on the control task. In the case of regulation, the controller is required to move the plant to a setpoint. In the more general case of tracking, the controller's task is to move the plant through a series of target states. As the controller does not receive feedback about the ongoing plant state,

the controller produces a time series of controls from the initial plant state, and a time series of desired states.

There are several reasons why purely feedforward control could prove to be unsatisfactory:

- It may be impossible to get a sufficiently accurate inverse model of the system, and the deviation of the produced path from the target may be a rapidly increasingly function of time (due to non-linearities in the dynamics of the system).
- Even assuming that our model is perfectly accurate, there may be error in the measurement of the initial state of the system, i.e. at $t = t_0$.
- Any external objects that cause the system to deviate from its path will not be corrected; a robot arm brushing against a surface will introduce errors that cannot be eliminated.

It would seem that a combination of feedforward and feedback strategies is the most promising approach, as the weaknesses of one method are often matched by the strengths of the other. A great deal of current research effort is devoted to this problem.

Learning control implies a kind of optimisation process, so there is some index of performance to be maximised (or some cost function to be minimised). The controller itself is a mapping from inputs to outputs, possibly with dynamic elements, and as such there will be a set of parameters that define the actual mapping produced. In linear feedback control, these parameters are simply the feedback gains; in the case of a neural network, these would be the weights (and possibly time constants - more about this later). The goal of learning control is to adjust the controller parameters so as to improve the performance on the control task. This perspective makes apparent the connection with other learning problems, particularly neural net problems. However, learning control is not the same as fitting curves to data. This is due to the fact that a dynamic system is involved, and we will generally be concerned with the stability of the combined controller/plant system. Such stability issues are seen to be paramount in the linear control literature, and are no less important in the general case of nonlinear control, although the powerful mathematical framework of linear systems theory is no longer available.

Possibly the most important issue when designing a learning controller concerns the nature of the training information. Although the problem has been formulated as optimising some index of performance, the value of this performance measure is not immediately informative about how to change the parameters. This is the *credit assignment* problem: given the index of performance, in what way should the control actions be modified so as to improve performance? There are generally two approaches to adapting the parameters. *Supervised Learning* assumes the existence of target, or desired control actions, and these are used to decide how to change the parameters so as to improve performance. In this case, the performance index is some measure of how close the actual and desired outputs are. However, the gradient of this performance with respect to the parameters is assumed to be known, or can be estimated. The other approach is that of *Reinforcement Learning*, in which no gradient information is available.

1.4.1 Distal Learning

One extremely common variant of supervised learning is that of *Distal Learning* problems. Here target information is available, but in a different co-ordinate system to that of the controller outputs. The most common situation is for the targets to be defined in the space of the plant, and not the controller's outputs. An obvious example of this situation, one to be examined in more depth later, would be the task of controlling a robotic arm. In this case, the task is likely to be defined as time series of target values, or 'setpoints' for the endpoint of the arm. Given that the controller's output will be defined in the space of the joint angles of the arm (or possibly a set of torques), there is a change of co-ordinates involved. This problem is typically approached by the use of models, which approximate the mapping between controller output and the space of the target values, and the model is used to transform the targets back to the controller outputs.

In *Reinforcement Learning*, the only information available is the performance index itself. This gives some measure of the appropriateness of the control outputs, but no information about how to change the parameters. This situation is rather similar to marking a students work with just a grade - e.g. giving a grade B- does not tell the student which were the strengths and weaknesses of the essay, or how to improve the

grade in future. Also note that even this case is made slightly easier than will generally be the case, as the student can guess that a B- is roughly in the middle of the range of possible grades. In general, the performance index is just a number on an unknown scale, and so it will not be known advance whether a rating of 0.2 is good or bad. In this case, optimising the controller will usually involve some element of random exploration in order to determine how the performance varies with control actions.

1.4.2 Adapting the Controller

Controller optimisation is simplest in the supervised learning case, in which the desired outputs are directly available. This information would come from a teacher in the form of a expert, or *operator*, usually human. The operator is observed while performing the control task, and supplies a set of training data for the controller. It now remains to adapt the controller in such a way that the learned system shows good generalisation, in the sense of interpolating and extrapolating to novel inputs. Obviously, the main limitation of this approach is its dependency on an existing expert, which will not be available for most difficult control tasks. In that case, the learning task will inevitably involve distal or reinforcement learning. In both cases, there is a gap between the training data and the optimal control actions.

In distal learning, this gap is bridged through the use of models. Consider again the case of the robot arm, in which the controller's outputs map to the joint angles of the arm, but the targets are defined in the space of the Cartesian position of the endpoint. A *forwards* model approximates the mapping *from* the joint angles *to* the endpoint position; an *inverse* model maps endpoints to angles. As an inverse function, this may not be well-defined, in the sense that there could be many sets of joint angles that produce the same endpoint position. Whether the model is forwards or inverse, this model may be used to optimise the controller. In conventional control theory, a plant model usually specifies the parameters of the controller directly, in the sense that the controller parameters may be analytically obtained from the parameters of the plant model. This design approach is only possible in some cases, and a learning approach has to be taken for more complex cases.

In contrast, *learning* control uses the model to obtain training data for the controller, which is optimised with some learning process, such as gradient descent. The gradient of the performance with respect to the plant output is related to its gradient with respect to the controller output; the relation between the two is just the gradient of the plant output with respect to the controller output. This may be easily seen if the behaviour of the plant can be described by a static function, which is approximated with a feedforward net $net1$:

$$net1: \quad y = F(u;\theta)$$

where y is the estimated plant response, u is the control input, and θ are the parameters of the model. In general y and u are vectors. The plant model is optimised by the procedure of *system identification* (Chen and Billings, 1992; Yamada and Yabuta, 1993; Pham and Lin, 1993; Qin et al., 1992). This refers to the choice of a set of control inputs, chosen to be representative of the kind of inputs the plant is expected to receive in use, and also chosen to be informative about all the plants modes of behaviour. Observation of the plant response gives a set of training data for the plant model, which is trained with supervised learning. This is *offline* learning, so the plant model is trained to some criterion of accuracy, and then its weights are frozen.

System identification is a research area in its own right, and an in-depth discussion is beyond the scope of this thesis. However, there are a few points that should be made. Firstly, the structure of the model often reflects the degree of knowledge of the functioning of the plant. The use of neural networks is often described as a 'black box' method, in that there is no explicit representation of prior knowledge of the plant. In contrast, many engineering applications make use of knowledge of the physics of the plant in deciding the model structure. For example, some of the mathematical relations that describe the system can be derived from physical laws or known systems of equations (conservation of mass, energy, momentum; equations describing fluid flows, for instance), or can be obtained by empirical studies (relating, for instance, a power law relation between the frictional force and the speed of flow in a pipe). Black box methods are most useful when the structure of the plant is difficult to define or measure, and the aim is to model the input-output relation without accounting for the process.

The other main caveat concerns the collection of training data. The goal of the training set is to represent data about the behaviour of the plant in such a way that all the modes of the plant are explored, or at least those that are likely to be seen in real use. Such a set is called *persistently exciting*. The problem is that building a persistently exciting set is a highly non-trivial task in itself.

If we assume that training information is available in the space of the plant output, then an error function may be defined:

$$E = \frac{1}{2}(y - y^*)^T(y - y^*)$$

where y is again the plant output, and y^* denotes the targets. Suppose the controller is implemented as another network, *net*2, and the plant and controller are arranged in series, so that the output of *net*2 forms the input to *net*1:

$$z \overset{net2}{\rightarrow} u \overset{net1}{\rightarrow} y$$

$$net2: \quad u = G(z, \phi)$$

The targets at the model output give rise to the usual error derivatives familiar in supervised learning:

$$\nabla_\theta = \frac{\partial y^T}{\partial w}(y^* - y)$$

which is used to adapt the model weights during system identification. In order to adapt the controller, the following error derivatives are used:

$$\nabla_u = \frac{\partial y^T}{\partial u}(y^* - y)$$

so the simple chain rule is all that is required to get training information for the controller. The term $\frac{\partial y^T}{\partial u}$ is just the vector extension of a first derivative; it is matrix of partial derivatives called a Jacobian, where the element $J_{ij} = \frac{\partial y_i}{\partial u_j}$.

This means that the plant model has to be in a form that can be differentiated; neural networks are an obvious example. In fact, neural nets make use of the chain rule as a way of solving the credit assignment problem for the hidden units. This fact was used by (Jordan and Rumelhart, 1992) to show how distal learning may be implemented using

just the machinery of backpropagation (Rumelhart et al., 1986), as the backpropagation procedure calculates a particular factorisation of the transpose of the Jacobian. Rumelhart *et al* also show that this is a general method for finding inverses of functions, even if such inverses are not well-defined, and this is used to develop controllers for systems with excess degrees of freedom.

It is worth spending a little time to understand exactly how this works. The controller is placed in series with the plant model, so that the outputs of the controller form the input to the model. In effect there is now one network, in which some of the weights have a learning rate of 0: these are the weights that comprise the plant model. If this combined net is now trained on the identity function, then the controller will converge on the inverse of the plant model. In general, this is just what is required of a feedforward controller. Although the weights of the plant model are not adapted in this process, we still have to backpropagate through the model to instantiate a particular Jacobian. One interesting property of this approach is that it is able to find an inverse, although the inverse may not be well-defined. To see how this happens, consider a simple case in which the forwards model is the mapping $y = sin(x)$, in the range $[0 : \pi/2]$. This function is 2 to 1, so the inverse has two possible branches. Now consider the use of distal learning to approximate the inverse, starting at two different places on the curve. The direction in which learning proceeds is given by the error derivatives, which are simply the errors at the output multiplied by the state-dependent Jacobian. The *sign* of the Jacobian is just given by the slope of the forwards function, which is positive in one half of $y = sin(x)$, and negative on the other half. This difference in sign ensures that solutions which start on one half of the curve stay there, as do solutions on the other half. Incidentally, it is worth noting that the process of multiplying the error derivatives by the Jacobian has the effect of distorting their values - for example, they will go to zero as the two solution branches approach each other (i.e. as the slope of the forwards function approaches 0).

Distal Learning chooses one inverse over the other in a way that is dependent on the initial state of the net. It is not too difficult to see how the learning could be biased towards one inverse through the use of additional terms in the error function, or the use of regularisers (Poggio et al., 1985). The important point to note is that the choice of

inverse is stable, in that the learning does not switch between different inverses.

This has been described as an *offline* procedure, in which all the learning for the plant model is performed before the controller is adapted. In general, offline learning means training the plant model until some accuracy criterion is reached. The setting of this criterion is left to the experimenter, and there is no easy way to decide what this value should be. Some control tasks require a more accurate model than others, and the relationship between the complexity of the control task and the required model accuracy is unknown in general.

An alternative is to use *online* learning, in which the plant model is adapted at the same time as the controller. This approach is not without problems however; in the early stages of learning, the controller will be trained with uninformative information, as the plant model will be inaccurate. Some kind of compromise between these approaches is probably best, with the plant model being trained to at least some degree of accuracy before the controller is optimised. The plant model would then learn at the same time as the controller, although probably with a lower learning rate. This was the approach taken in (Jordan and Rumelhart, 1992).

There are several ways in which this scheme may be extended, and an exploration of this will be used in much of the experimental work in this thesis. Firstly, the general idea of backpropagating through networks is well-defined in the case of recurrent, or dynamic networks. Backpropagation for recurrent nets is discussed below; the main difference is that the error derivatives are now time-dependent quantities that have to be integrated. Secondly, the network does not have to be a model at all, and it was shown in (Jordan, 1988) how to use distal learning to implement forms of constraint satisfaction. The basic idea is that a constraint can often be represented as a many-to-one mapping, and inverting that mapping is a way of finding an output which satisfies the constraint.

A simpler approach to learning inverses is *Direct Inverse Learning*. Again, observation of the behaviour of the plant in response to a set of control inputs gives a set of training data, consisting of pairs of input/output values. Direct inverse learning proceeds by simply reversing the pairs of data points, so that the observed plant output is given as input, and the corresponding control inputs as the desired controller output. Kuperstein has

suggested this method as a way of learning simple hand-eye co-ordination (Kuperstein, 1988; Kuperstein, 1992). The effectiveness of this method is strongly dependent on the characteristics of the plant - if the transfer function of the plant is many to one, then the inverse is not well defined, and this method may not work. This is due to the fact that one input to the controller will be trained to associate with more than one target output, and the actual output produced after learning will be in the convex closure of the training actions. If the inverse of the desired plant state is not a convex set, the controller output may not be in the inverse image.

One final approach is Feedback-error learning (Kawato et al., 1987; Gomi and Kawato, 1993), which is closely related to distal learning. In this formulation, there are two controllers: a fixed linear feedback element, and an adaptive nonlinear feedforward controller. If the feedforward controller *was* an inverse of the plant dynamics, the feedback controller would be redundant, and so would remain silent, as the plant would perfectly track the target trajectory. The goal of learning is to reduce the output of the feedback element to zero. This is achieved in the simplest way, by using the feedback output as an error term for the feedforward controller. It turns out that this algorithm has interesting convergence properties that make it a practical method for inverting the plant dynamics; the actual gradient that the learning follows is closely related to that used in Distal Learning.

1.4.3 Direct Control and Reinforcement Learning

In contrast to the model-based approaches outlined above, direct control techniques use the performance index itself as the training data. There are two main approaches in the reinforcement learning literature. *Temporal Difference* methods are based on the dynamic programming approaches to learning control in Markovian domains. Control in such situations is discrete, in that time proceeds in steps, and the controller has a finite number of actions available. Reinforcement learning in continuous domains is usually approached through the use of *stochastic automata*. Both these approaches have their benefits and drawbacks. TD methods are more suitable to finite problems with delayed rewards, such as game-playing, and stochastic automata more useful for control of dynamic systems.

THE REINFORCEMENT AGENT

We may define a reinforcement agent in the same way as a nonlinear controller, ie as a mapping from sensory input to control actions. The only information available for training the agent is the performance index itself. As there is no gradient information available, the controller has to experiment with different actions, and compare their effectiveness. This means that the controller has to have some source of variability in its output, ie it has to be able to perform different actions in identical situations. This is usually achieved by having a stochastic element in the controller. Reinforcement learning is thus seen as a process of *selection*, which operates on the variability in outputs. The reinforcement received is given by an element called the critic. This is some function of the current state of the plant, just as the index of performance described above, with the important qualification that reinforcement problems usually imply that the reinforcement function itself is a random function. If the reinforcement function is deterministic, the whole learning task becomes greatly simplified.

Setting the degree of variability in the controller output involves resolving a conflict between two opposing goals - exploration and exploitation. Exploration refers to the need for the controller to try out different actions to gain information about their utility. Exploitation refers to the use of existing knowledge to maximise performance. These goals conflict, because exploration naturally drives the controller into states about which it has little knowledge, while exploitation naturally moves the controller into states which are known to lead to high reinforcement. This usually results in choosing a level of exploration which is some kind of compromise, although some researchers have used controllers which switch between exploration and exploitation phases (Thrun and Moller, 1992). The exploration itelf is most easily implemented as choosing random actions. However, this is highly inefficient, and not practical for many control tasks in which random controls could damage the plant. This has lead some researchers to use more directed forms of exploration, that are based on the intuitively sensible idea of trying to maximise information gain (Thrun, 1992)

Consider a simple case in which the agent takes a linear combination of inputs, and uses that value to generate an output. The output is often a random variable, in contrast

to the controllers discussed above; this is the simplest way of producing variability. The actual action may be one of a discrete set of allowed actions, or it may be a continuous variable. The TD-based methods are derived from the field of *dynamic programming*, in which the plant and the agent are described as discrete random Markov processes (Barto et al., 1989; Barto et al., 1983; Ross, 1983). This formulation has consequences for the derivation of the TD learning rules, such as restricting the controller to producing one of a set of allowed actions, rather than allowing continuous actions. This also restricts the architecture of the controller to a look-up table, in which the action for one input may be changed independently of all the other inputs. Given these restrictions, convergence of learning can be proven (Dayan and Sejnowski, 1993). In practice, these restrictions are often relaxed; for example, controllers with distributed representations are commonly used.

One important class of reinforcement problems is that in which the reinforcement is delayed with respect to the corresponding actions. A classic case of this occurs in game playing, where the reinforcement could simply be an indication of whether the system won or lost. The problem is that this reinforcement is only available at the end of the game, but in a game like chess, the moves actually responsible for the outcome may have occured much earlier in the game. The can be seen as another version of the credit assignment problem, in this case a temporal version. The TD class of algorithms attempt to solve this problem by making predictions about the future values of reinforcement that will be received. In the chess example, the *utility* of a particular board position is defined to be the expected value of all the future reinforcements that will be received as a result of starting in this position, and using the current control policy. This sum is usually made convergent by an exponential weighting of the future reinforcements. The measure of 'utility' incorporates future reinforcements, so its effect is to smooth the reinforcement function into neighbouring states.

TD learning is a particularly elegant method for learning this type of utility functions, by making use of the fact that the utility of the state at t is related to the utility at $t-1$. TD methods are examined in detail in (Barto et al., 1989), using an example problem of a mobile robot trying to find a goal state. This is a good example of learning a delayed

reward task, as the robot only receives reinforcement when it finds the goal. The utility of all the other states simply reflects their distance from the goal. TD learning involves two components - learning the utility of each state, and using that utility to produce actions. In a slightly different formulation, Q-learning (Watkins, 1989) combines the operation of these two components into one system, by considering the transition between states rather than the states themselves.

STOCHASTIC AUTOMATA

One major limitation of the TD methods is that the controller output has to be drawn from a finite set; this is clearly more suitable for game-playing than control of a dynamic system. Stochastic Learning Automata were developed from a different theoretical perspective, which is less concerned with delayed reward problems. Such an agent produces a real-valued output, and essentially measures correlations between changes in its output with changes in the reinforcement received. This information is used to change the parameters of the distribution that generates the control action. In one of the earliest applications of these ideas, a reinforcement agent was used to balance an inverted pendulum (Barto et al., 1983) - a task that has become a benchmark for reinforcement algorithms. A more recent algorithm is that of the SRV unit (Gullapalli, 1992). The actions are drawn from a Gaussian distribution, with the mean and variance under the control of learning. In (Gullapalli et al., 1992) this system was used to solve a peg-in-hole insertion task for an industrial robot.

In all these cases, it is possible to have more than one agent. For a plant which takes multidimensional inputs, the control system could be a network of agents, in which each agent provides the controls for one of the plant's inputs. There are several ways in which the network may be trained. Each agent may be trained in isolation, so the reinforcement is broadcast to each of the agents, and each agent tries to independently optimise the performance. Alternatively, the agent may form the output layer of a neural net, with backpropagation used to train the hidden layers. This enables the agents to use shared representations, rather than acting independently of each other.

Athough reinforcement was originally conceived as a model-free approach, models of one form or another have found widespread utility. One kind of model might be to

approximate the reinforcement function itself; this treats the reinforcement task as a distal learning problem. Once this model is sufficiently accurate, we may apply targets that correspond to the maximum possible reinforcement, and backpropagate to optimise the control actions. Another kind of model would be to approximate the behaviour of the plant. One reason for doing this would be to enable *planning* methods to be used, in which the utility of whole action sequences is evaluated by trying them out on the model. In (Thrun et al., 1991), a differentiable plant model was used to simulate the effects of a sequence of N actions, and the fact that the model is differentiable was used to backpropagate and optimise the action sequence.

1.5 Training Neural Networks for Control Tasks

In the methods discussed above, the controller may be implemented using a variety of methods, from look-up tables to rule-based systems. However, neural networks have found widespread application in the control and robotics fields, mostly due to their ability to approximate arbitrary nonlinear functions, and to generalise well from sparse data (White and Sofge, 1992; Miller et al., 1990, give an excellent review). These factors make neural nets an attractive method for building nonlinear motor control systems.

Learning control is not curve fitting. This can be obscured when neural net methods are used, as it seems that we are always performing gradient descent, or some other optimisation procedure, just as we would in nonlinear regression. The difference in control tasks is that a dynamic system is involved: the plant. Consider the situation of a feedback controller implemented as a neural net. At each time step, the net receives some coding of the task, together with the plant state, and emits a control action. The control action changes the state of the plant, possibly in a way that will not be detected until much later. This means that the future inputs are partly a function of the current controller output. To perform true gradient descent of an error function, this dependency has to be taken account of, usually with some form of *recurrent backpropagation* (Narendra and Parthasarathy, 1991). Recurrent backpropagation is an extension of the standard algorithm to deal with nets that have feedback connections between units, so that the current state of the net forms part of the future inputs. The algorithms themselves are discussed in more detail

in chapter 4. The point to be made is that neurocontrol generally involves changing the parameters of a nonlinear dynamic system, which implies that problems of stability and convergence are a major concern, and calculation of gradient information is more involved than in the static case.

1.6 Conclusions

We have outlined the most common ways in which nonlinear motor control tasks are typically approached, drawing the important distinction between indirect, or model-based approaches, and direct approaches. These two methods represent two different ways of solving the credit assignment problem. The credit assignment problem is a key part of learning control: solving this problem allows the search for optimal actions to become more directed.

The framework of Distal Learning is of particular relevance in this thesis. The basic idea is to model the plant as a differentiable system, and to account for the plant behaviour by differentiating the model. This is a general method for accounting for the presence of *any* dynamic system in the control problem. This will become relevant when the structured nature of biological systems has been discussed. If the control system consists of a set of modules, each performing its own optimisation, the training of each module has to take into account the presence of all other modules. Distal Learning is a straightforward way for each of the modules to solve their own credit assignment problem.

Reinforcement learning attempts to learn control without a plant model, and will be expected to proceed more slowly than model-based control for many cases. One way to enhance the power of a reinforcement controller is to provide it with a repertoire of more complex actions. This will generally mean a decomposition of the control problem into more simple subproblems, and taking an incremental approach to the complete control task. The details of such an approach are left until chapter 6. This kind of hierarchical system is clearly related to the structures seen in biological control systems, the topic of the next chapter.

2 | Biological Control Systems

2.1 Introduction

The concept of an animal is partly defined by movement; what separates animals from plants is primarily the ability to move around their environments. The nervous systems of animals have to solve difficult control problems to produce adaptive behaviour. The complexities of the control problem are partly due to design of animal's bodies, as well as the dynamics of their environments. These complexities become apparent when attempts are made to build robots to solve the same problems. A good example of this is the control of walking systems, a problem which has eluded solution except in simple cases. As with many other biological control problems, walking involves the co-ordination of many moving parts, each with their own nonlinear dynamics. The degree of nonlinearity, and the number of degrees of freedom both serve to increase the complexity of the problem. Despite this, insects are able to walk in a variety of different conditions, including uneven surfaces with with widely varying characteristics, and able to cope with damage, such as the loss of a leg.

One of the puzzling aspects of biological control systems is that they are often composed of rather few neurons - circuits containing tens of neurons can generate entire, complex behaviours (e.g. the stomatogastric ganglia in lobster (Marder, 1988)). Although biological neurons are far more complex than the units of most connectionist models, it does seem that evolution has found very simple solutions to very difficult problems. The main goal of this chapter is to describe the key architectures used throughout biological systems; in other words, to discuss what is common across species.

Most biological control tasks correspond to oscillatory motion: flying, swimming, and

walking all consist of repetitive, periodic actions. The environments in which animals have to survive are often unpredictable and noisy, and adaptive behaviour often means producing a variety of different motor outputs in different circumstances.

2.2 Forming the basic pattern: Central Pattern Generators

Given that most biological motor tasks consist of producing oscillatory control signals, a great deal of research effort has focused on finding the source of these oscillations. Much of this work concentrated on the debate between central and peripheral sources (Delcomyn, 1980). Ultimately, a compromise has been reached, where central mechanisms are seen to be responsible for the basic pattern, but external influences (such as reflexes and sensory inputs) have a large influence on the final behaviour. One powerful concept to arise from this research is that of the *Central Pattern Generator.* Central Pattern Generators (CPGs) are typically small neural circuits that are capable of producing temporally structured outputs (usually oscillations). It is now well established that, in many species, these circuits produce well-coordinated outputs even when completely isolated from external inputs (Getting and Dekin, 1985; Grillner and Wallen, 1984; Selverston, 1985). For example, the spinal cord of the lamprey can be isolated and dissected from the animal, and the basic oscillatory behaviour evoked (so-called *fictive swimming*) by stimulating the appropriate neurons (in this case stimulating the trigeminal nerve is sufficient to produce fictive swimming).

One well-studied example is the sea slug *Tritonia*, whose CPG is responsible for producing swimming movements as an escape behaviour. Initiation of CPG activity is often caused by the activation of *command neurons* (in the case of *Tritonia* these are called the Dorsal Swim Interneurons (DSI cells)); these cells are interesting because their activation results in a complex, temporally structured behaviour, but the input to DSI cells is of a simple, non-structured nature. This general design clearly has implications for a decision-making module located at a higher level, as complex behaviours can now be treated as individual units, in much the same manner as the subsumptive architectures of Brooks (Brooks, 1991)

Other examples have also been extensively studied, in particular the swimmeret in

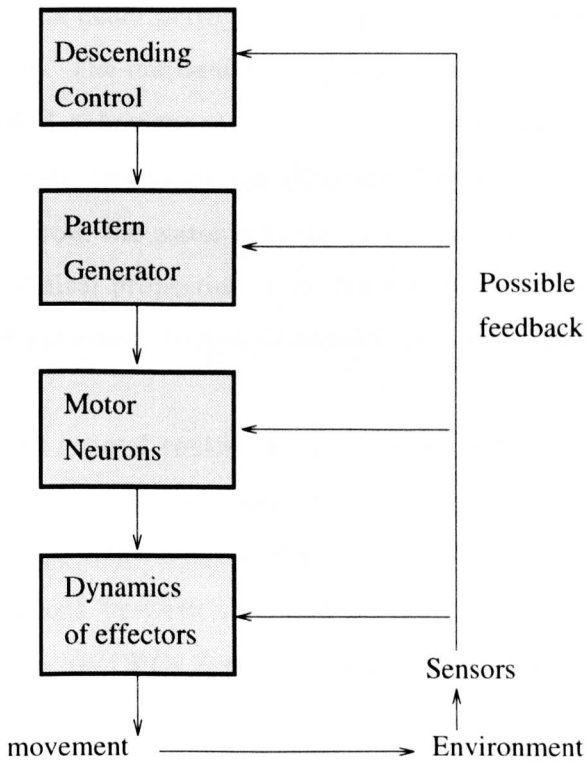

FIGURE 2.1. Hierarchical nature of biological control structurs

lamprey, the gastric mill in lobster, and the circuit controlling locomotion in the locust (Grillner and Wallen, 1984; Tsung et al., 1990; Heinzel and Selverston, 1988; Williams and Sigvardt, 1992). The lamprey swimmeret consists of a chain of coupled oscillators running down the animals body, with co-ordination between the oscillators achieved by a set of couplings. The neural activity produces a wave of curvature passing down the body that pushes the animal forwards. The frequency of this wave is such that there is always roughly one complete period between the head and the tail. This remains constant despite changes in swimming speed, so the delay between neighbouring segments is adapted to maintain a constant relative phase. The mechanical properties of the muscles, and the water itself play a part in the evoked behaviour, and some researchers have produced sophisticated simulations of the entire fish/water system (Ekeberg, 1993). One result from this work is to show that the forces from the water help the fish maintain a constant frequency. In a similar study, the mechanical properties of the lobster gastric mill were simulated, along with the known neural dynamics, to give reasonably plausible chewing behaviours (Doya et al., 1993).

In all these cases, the general control scheme can be pictured as in 2.1. This scheme should not be taken too literally, as individual neurons may participate in more than one motor circuit, and command neurons may be part of the pattern generating circuit. There may also be feedback between the other parts of the system. The main point is that all these systems contain CPGs that can independently produce complex, temporally structured outputs.

Basic Physiology of the circuits

CPGs reside in the segmental ganglia of invertebrates, and in the spinal cord of vertebrates. They range in complexity from single neurons with intrinsically oscillatory dynamics, to coupled oscillators comprising many neurons. Reciprocal inhibition is a common mechanism for producing oscillations. CPGs in invertebrates are particularly amenable to analysis; this is due to the fact that the ganglia usually contain relatively few neurons, and the neurons themselves are often very large, making recording of their activity particularly simple.

In contrast, motor circuits in vertebrates usually contain very large numbers of rather

small neurons, making it much harder to understand how the circuits work (Cohen et al., 1988). The basic action of spinal circuits seems to be very similar to those of invertebrates: basic oscillatory motion can be produced entirely by spinal circuits, without any input from the brain or the senses. (Grillner, 1985, shows this for spinal cats). The patterns produced by these circuits are rather complex: as discussed below, incoming sensory information to the spinal circuits has a very large effect on the movement.

2.3 Modifying the behaviour to the demands of the task

Central mechanisms for producing structured outputs are clearly useful: their presence enables higher-level structures to plan at the level of complete behaviours, they ensure that the animal is not completely pushed around by its environment (which would be the case if sensory inputs were solely responsible for the behaviour), and they provide a degree of robustness in the face of external disturbances (due to the limit cycle properties of the oscillations). However, a system that relies completely on central control would be insensitive to environmental changes, and a control system that can only produce one pattern places limits on the behavioural repertoire of the animal. The biology seems to have adopted the strategy of building CPG circuits that can be modified in a variety of ways: by sensory inputs, by descending information from the brain, and by circulating neurotransmitters.

SENSORY INPUTS:

Animals are faced with the problem of controlling their actions in the face of uncertainties about the environment. In the previous chapter, the use of explicit planning and reactive strategies for learning control in an environment was examined. Parallels may be drawn between the use of CPGs, and the planning methods described earlier. On the other hand, reactive strategies correspond to the use of sensory information to modify the behaviour. For example, flight in the locust requires alternating activation of elevator and depressor muscles, which is basically produced by central mechanisms. However, the animal is subject to unpredictable disturbances such as gusts of wind, and so some form of feedback is necessary to maintain the appropriate behaviour (i.e. keep the animal heading in the same direction). The locust is equipped with wind-sensitive hairs on its head,

and stretch receptors that measure the position of the wings. Both these sensory inputs are capable of influencing the phase of the CPG oscillations responsible for flight, which is a simple mechanism for coping with disturbances in the environment. The *optomotor response* uses optic flow to maintain a constant heading.

The CPG for locomotion in the cat receives several sources of sensory information, from proprioceptors in the muscles, angle sensors in the joints, and stretch receptors in tendons. Again, it has been shown that coordinated locomotion is possible in a spinal animal. The usual experimental set-up is to place the spinal cat on a treadmill that is run at a variety of speeds. If the dorsal roots carrying sensory feedback information are left intact, the locomotion is seen to be matched to the treadmill speed. As the treadmill is made to go faster and faster, the gait changes from walking to trotting to galloping. However, if these dorsal roots are cut, the speed of locomotion no longer matches that of the treadmill (Grillner, 1985); so this sensory feedback is clearly necessary to adjust the behaviour to the demands of the task. In the intact animal, information from these sensors, and from the CPG itself is fed back to higher centres, although the exact role of this feedback is much less well understood. (Oku et al., 1993; Pearson, 1993).

DESCENDING SIGNALS FROM BRAIN:

Descending inputs from the brain to motor circuits often represent some kind of behavioural choice. For example, in the dogfish, stimulation of the left mesencephalon produces left turns, and stimulation of the right mesencephalon produces right turns: these areas have a clear behavioural interpretation. In the case of the dogfish, the situation is particularly simple, as steering can be achieved by superimposing a bending motion on the basic oscillations. Obviously, this is an advantage for the control system, as the activity of the CPG remains unchanged. However, this is not true for many other animals, and the most common situation is that timing relations between the CPG neurons have to be changed to modify the behaviour.

NEUROTRANSMITTERS:

Finally, the biology makes widespread use of various neuromodulatory substances to change the activity of the motor circuits. In some cases, the effects are a gradual modification of the basic pattern; in other cases the neurotransmitters can dramatically reconfigure

the circuit. A well-studied example of this is the lobster gastric mill. The pyloric part of the lobster is controlled by a small circuit of 14 neurons: this circuit controls the action of a set of teeth that are used to chew food particles to aid digestion. The circuit is capable of producing two distinct modes of chewing: squeeze, and cut-and-grind. Both these patterns involve the teeth describing complex three-dimensional trajectories. Levels of proctolin appears to be used to switch between these two modes of behaviour. Other circulating compounds have effects on the frequency and amplitude of the pacemaker cell, but the effects of these compounds on other neurons in the circuit are dependent on the mode the circuit is in. In this way, proctolin may be thought of as reconfiguring the circuit (Marder, 1988).

UNDERSTANDING THE FUNCTIONING OF MOTOR CIRCUITS

These points have implications for understanding the functioning of motor circuits. The CPG in *Tritonia* has the same number of neurons in all animals of the species; moreover, the neurons can be individually identified and classified. Despite this, it has proved difficult to assign computational roles to each neuron, mainly because the whole circuit is capable of organising itself into several different configurations, depending on the behavioural context. We may call such a system *strongly coupled*, in that the activity of one neuron is strongly dependent on the activity of all the others. *Weakly coupled* systems are those in which the contribution of the other neurons may be 'averaged out' in some sense. For example, the mean field theories used to prove convergence for Hopfield-type nets (Hertz et al., 1991) relies on replacing the contribution of all the other neurons by their mean activity. Analysis of strongly coupled systems is clearly more difficult, and models quickly become swamped by large numbers of parameters and variables.

2.4 Pattern Generating Systems and Motor Programs

The concept of a motor program, in one form or another has been central to our understanding of human motor behaviour for a long time. Motor programs are usually conceived as an abstract representation of a behaviour, which is used to generate a sequence of motor commands in an open-loop fashion. Originally, these were thought of as a literal analogy to computer programs - an entire sequence of actions that would be selected, and run.

There are storage problems with such a system, as it seems that a new motor program would be needed for every new movement. There have been subsequent attempts to generalise the notion of a motor program to a scheme-like entity, that uses a set of parameters to generate a whole family of movements. Motor programs are also now seen as including feedback terms.

There is a close relation between motor programs and CPGs, and the two terms are used interchangeably in the experimental part of this thesis. CPGs take incomplete specification of a motor act, and generate the actual motor commands. The evoked behaviour is a function of the abstract representation of the task (the motor program), the interaction between the CPGs, and the dynamics of the plant and environment. A particular pattern of activity over the CPGs may be thought of as implementing a particular motor program, and so a set of CPGs supports a class of motor programs. It is a matter of controversial debate whether the motor programs should be thought of as distinct entities from the CPGs themselves.

2.5 Self-organising systems and models of psychophysical data

By comparison to invertebrate motor systems, the circuits in vertebrates are much more difficult to analyse. There is a body of work that attempts to take a much more abstract, and mathematical approach to understanding motor behaviour in these systems, and one that borrows very heavily from the language of dynamic systems: SYNERGETICS (Haken, 1983).

It has been well understood for a number of years that nonlinear dynamic systems are capable of very complex behaviours, which can be dependent on the settings of their parameters (Tabor, 1989). In addition to the literature on chaos, there is a body of work that considers the phenomena of *pattern formation* in open, non-equilibrium dynamic systems. A classic example of pattern formation is given by a fluid heated from below. Most of the time, this will result in convective vortices forming in the fluid, but in an apparently random manner, reflecting the large number of degrees of freedom in the system. However, particular values of the heat input result in the spontaneous formation of hexagonal patterns in the convection. When this happens, it is found that relatively few variables

are needed to describe the behaviour of the system (these are called the *order parameters* of the system), and so it is said that the system has self-organised onto a low-dimensional attractor. This phenomenon is of particular interest in motor control, because biological motor systems usually have to control high-dimensional plants (Bernstein's 'degrees of freedom problem' (Bernstein, 1967)). Self-organising systems appear to give a way of reducing the effective number of degrees of freedom of a system, and in a way that is extremely sensitive to the particular state of the system, unlike other methods, such as introducing regularisers. The other feature of attractor systems is that they are usually capable of a high degree of behavioural complexity, as parameter changes can produce phase transitions, and so switch easily between different behaviours. This approach is in contrast with the idea of representing motor programs as schemas, or any other static representation. This implies that the CNS not only represents individual patterns (the attractors of the dynamic system), but the entire dynamic environment of those patterns, in particular, the stability of the patterns as control parameters are varied.

Perhaps the most interesting point is that these analytic tools have had a great deal of success in modelling biological systems. For example, locomotion in quadrupeds (Schoner et al., 1990) is characterised by a set of distinct behaviours (gaits), where each behaviour consists of coordinating the motion of many degrees of freedom. Switching between gaits is caused by a change in one parameter, the speed of locomotion. The two characteristics of attractor systems mentioned above - reduction of effective number of degrees of freedom, and pattern change - seem to correspond very well with the observed behaviour.

MODELLING PSYCHOPHYSICAL DATA WITH SYNERGETIC SYSTEMS

In a classic series of experiments, Scott Kelso and his colleagues used concepts from synergetic theory to model biological movement data, ranging from gaits in quadrupeds, to timing measurements in human tapping tasks (Kelso et al., 1988; Schoner and Kelso, 1988a; Schoner and Kelso, 1988b; Kelso and Schoner, 1988; G and Kelso, 1990; Schoner and Kelso, 1992). The modelling was performed at the level of organisational principles of behaviour, in contrast to the work on invertebrates. Part of the reason for this is that the systems involved are typically large and complex, and it is far from clear that a useful understanding of function could be made from such a low-level approach. The biological

systems studied are capable of producing highly patterned outputs, but in a way that is very dependent on the environmental demands (i.e. the same mechanisms are responsible for a whole variety of patterns). The patterns do not appear to be *hard-wired*; rather, they seem to self-organise to suit each behavioural situation. This means that a particular neural structure underdetermines the behaviour that it produces. On the other hand, the organisational structures underdetermine the biological implementation, as very similar organisational principles can be implemented in a variety of ways. For these reasons, it would seem that an analysis at the level of behavioural organisation might be profitable.

Most of the experiments involved subjects performing a tapping task with the forefinger of each hand, the frequency and relative phase between the fingers being specified externally.

To model this kind of data using concepts from synergetics, a link has to be made from experiment to theory:

- The order parameters have to be identified. This may not always be straightforward, but in the case of the tapping experiments, the relative phase between the two hands is an obvious choice; in the case of quadrupedal gaits the relative phase between the limbs is chosen.

- Stable patterns observed in experiment need to correspond to attractors of the order parameters: in other words, the equations describing the behaviour of the order parameters have to have solutions at points that correspond to the stable patterns observed.

- Control parameters need to be identified; the values of these parameters determine which patterns will be observed (in the gait example above, the speed of locomotion is an obvious choice, as changing the speed moves the system through the possible gaits. In the tapping example, the frequency turns out to be the appropriate choice).

- The stability of the observed patterns needs to be determined, and the way in which this depends on the values of the control parameters. It is by studying the manner in which this stability varies with the control parameters that the form of the underlying

equations is discovered (i.e. the equations describing the behaviour of the order parameters).

- Identifying the order parameters for a system corresponds to identifying one level of description: a key goal is to understand the relationship between this level, and the implementation, in terms of neural hardware. In most of the work by Kelso *et al*, the neural implementation is modelled as a collection of coupled nonlinear oscillators.

One key finding is that the stability of the observed patterns can depend in a critical manner on the value of the control parameter, in the same way that dynamic systems can go through bifurcations as their parameters are changed. In the tapping experiments, subjects can only reliably produce in-phase and anti-phase oscillations; moreover, the anti-phase condition loses stability above a critical frequency. Bifurcation points can be identified by critical loss of stability, which corresponds to *critical slowing down*, i.e. a system which is perturbed away from the pattern takes longer and longer to return to the pattern. As the system is modelled as operating in a constant level of noise, the variance of order parameters gives another measure of their stability as the control parameters are varied. Variances are easily measured from the raw movement data; critical slowing down (an increase in *relaxation time*) is measured by adding small torques to the subject's fingers. Loss of stability is suggested as a mechanism for changing patterns.

This model corresponds to a set of equations describing the behaviour of the order parameter, which, in this case, is relative phase between the fingers. In the tapping case, the model is a differential equation describing the dynamics of the relative phase between the fingers. This equation has two terms, corresponding to intrinsic and extrinsic dynamics. Intrinsic dynamics describe the behaviour of the system in the absence of any sensory inputs, and extrinsic dynamic describe the way in which any sensory inputs modify the behaviour. The equations are able to explain many details of the experimental data, such as the number of stable patterns, and the way in which the stability of the patterns changes with the control parameter. The fact that the equations have so few terms is an indication that the biological system is indeed well-described by a low-dimensional system, and suggests that we have a minimal model of the phenomenon.

Finally, *learning* of coordination patterns has been analysed within the same framework

(Schoner and Kelso, 1992). As above, the coordination dynamics are modelled as a sum of the intrinsic dynamics, and those imposed by the demands of the task. In addition, a third term represents currently memorised information, where the strength of contribution of this term is given by a memory coefficient. This coefficient controls how much the dynamics are dominated by memory, and the value of the coefficient is defined to be some simple function of the degree of match between the memorised and required pattern, so that the memorised information becomes more dominant as the task is learned.

The other component of learning is to have the memorised pattern become matched to the task constraints, ie have the memorised relative phase become matched to the required relative phase. In absence of any other information about the dynamics of learning, the simplest model would have the memorised patterns gradually approach the required pattern, with the motion being some simple function of the degree of mismatch between the two. In (Schoner and Kelso, 1992), the motion was modelled with a linear differential equation. This causes the memorised pattern to exponentially approach that required by the task constraints (learning rates for skill acquisition usually closely follow an exponential time course) , and the strength of contribution of memory is simply dependent on the degree of match between memorised information and task requirements. All of this can easily be expressed in another dynamic system, so the whole of the subject's performance, *including the learning process* is now captured in one dynamic system. What is so interesting about this work is that the model is able to account extremely well for the experimental data, despite the apparent simplicity of the model. This strongly suggests that biological control systems are amenable to analysis at an abstract level, despite being composed of many complex parts.

2.6 The role of the Environment in Control

One factor that is often ignored in control theory and robotics is the role of the environment; in contrast, the dynamics of the environment appear to be very important factor in biological control. Some researchers have taken an extreme position about the possible benefit of embedding agents in complex environments, claiming that this coupling effectively does away with the need for internal representations at all (Brooks, 1991; Husbands

et al., 1993; Parisi et al., 1990) (Clark, 1994, argues in favour of moderating this view). In any case, it seems more likely that a complex, dynamic environment might be partly responsible for complex behaviour, and this will change our view of internal representations, as internal states no longer enjoy a simple mapping to behaviour.

In the context of locomotion, the environment may well be partly responsible for the stability of movement. For example, in lamprey swimming, mechanical interactions with the water produce a phase-locking effect; the fact that the coupled fish-water system is phase-locked provides a basic stability against perturbations. In the case of flight in the locust, the isolated CPG seems to be rather poor at producing stable oscillations with a reliable frequency; sensory inputs have the effect of entraining the flight with the environment. Although the view that oscillatory motion is peripheral in origin has been shown to be incorrect, the dynamics of the environment, as mediated through the senses, and through purely mechanical interactions, are now known to be partly responsible for the stability of the motion.

2.7 Discrete Voluntary Movements in Primates

Humans and other primates make great use of their hands. One important part of manipulation is reaching behaviour: to use a representation of the location of an object (say, from vision), a representation of the initial position of the hand (say, from proprioception), and produce the motor commands that result in the hand being positioned close enough to the object for grasping. This is a task that is frequently addressed in robotics, and so it provides an interesting case for comparing engineering efforts and biological strategies. A typical engineering approach to this problem would follow very closely the methods outlined in the previous chapter.

Robotics Approaches

A classic robotics approach to reaching would involve a representation of the workspace of the robot, which is just a map, in the configuration space of the robot, which locates all the obstacles, and the initial and goal state. Trajectories are usually formed by a combination of planning and reactive techniques. A planned trajectory can be considered to be a collection of setpoints, and the role of the controller is to drive the system through those

points. It is very common in robotics to have explicit forward and inverse kinematics and dynamics models; most robot manipulators are designed with a minimal number of degrees of freedom, so that redundancies are reduced to a minimum, and explicit regularisers that are based on controllability measures are often used to remove remaining excess degrees of freedom.

BIOLOGICAL APPROACHES

Because most feedback loops in biological reaching systems are relatively slow and inaccurate, it seems that reaching is largely pre-planned, and operates in open-loop mode, which implies the same centrality that was seen in CPG circuits. In **which species**, these central representations have been extensively studied by single cell recordings. In rhesus monkeys, the representation involves a set of motor maps, in which the direction of movement is coded by activity over a population of neurons (Georgopoulos et al., 1986). The individual cells are broadly tuned to a particular movement direction, with activity falling off roughly as $cos(\theta)$, where θ is the difference between the unit's preferred direction and the actual movement direction. The vector sum of all the units (their preferred direction weighted by their activation) is highly correlated with the direction of movement, with the length of this population vector representing the initial movement velocity.

The population vector has some interesting properties, particularly concerning what happens if the representation of the movement changes. In an especially illuminating set of experiments (Georgopoulos et al., 1993), monkey subjects were given a delayed reaching task, in which a target position was displayed, then removed, and a trigger signal then given which prompted the movement. The key in this experiment was that the trigger signal itself was a vector, and specified that the movement direction be at an angle from the original stimulus vector. The direction of this rotation was not known to the subject in advance. The subjects reaction time increased with the size of the rotation, which leads to the hypothesis that the population vector is smoothly moved to the new movement direction. In the second half of (Georgopoulos et al., 1993), this was experimentally observed: the population vector is seen to rotate smoothly to the new direction. In some related computational work, it was shown that a motor map with

simple local interactions could explain these results, through the use of simple mechanisms for changing the population vector (Massone, 1994)

USE OF INTERNAL MODELS

It has been widely hypothesised that skilled reaching requires the use of a plant model, and the considerations of the previous chapter lead us to see such a model as a natural candidate for a central mechanism for reaching (i.e. an *inverse* model is a natural feedforward controller). The presence of a *forwards* model enables several computations to be performed. In a system in which feedback is delayed, the model may be used to generate artificial feedback data before the real data arrives. The forwards model may also be used to give training data for a feedforward controller. Finally, a model is needed to combine information from motor commands and sensory feedback to give an estimation of the state of the plant. A related version of this state estimation problem is that of 'efference copy' (Gallistel, 1980), in which knowledge of the current control actions are used to cancel their sensory effects. This is a key ingredient for a system which uses optic flow to estimate motion: the effects of eye movements within the head cause their own optic flow, which has to be 'subtracted' to leave the component which is due to movement of the body. Despite all this, the internal model has been an elusive system.

One study providing strong support for a forward model involved human subjects making reaching movements in the dark, and trying to estimate the final position of their hands (Wolpert et al., 1995). In this case, there are two possible sources of information about the plant state: the motor commands (dead reckoning), and sensory inputs from proprioception. In this situation, the position of the hand is considered to describe the state of the plant, and the motor commands are the control inputs to the plant. The information from proprioception is the plant output, but note that the plant outputs are not immediately informative about the plant state. As discussed in Chapter 1, a plant model is required to transform one into the other.

The estimation process was modelled as a Kalman filter, which is a linear dynamic system, designed to form optimal estimates of the internal state of a system given the control inputs and observations of the output. This is a 'hidden variable' problem, because the plant state cannot be directly measured, so the estimate of the plant state at time t is

used to make predictions of the sensory inputs at time $t+$. The discrepancy between the predicted and actual sensory inputs is used to update the estimated plant state.

This system was 'tuneable' in the sense that the relative contributions of motor commands and sensory inputs could be changed. The system was able to account for the data only if it received both motor commands and sensory inputs; it was the trade-off between the accumulating errors in simulating the arm dynamics, and the feedback from sensory inputs, that produced the pattern of the data. This trade-off is not observed in a system which relies just on dead reckoning, or sensory feedback. As discussed earlier, the use of sensory information in such a state estimation task requires a plant model in some form, so this work provides good evidence for the existence of an internal plant model.

In a related paper (Shadmehr et al., 1995; Shadmehr and Mussa-Ivaldi, 1994), the characteristics of the hypothesised model were investigated, by presenting subjects with a reaching task in which the dynamics of the environment were changed. This involved subjects making reaching movements while holding the end of a planar robot arm. The robot can produce torques at the end effector, and so novel dynamic environments may be simulated. Subjects typically make straight reaching movements towards a target; the novel environment was a force field that disrupted the subjects performance. After a few hundred movements, the subjects had learned to remove the effect of the force field. This learning was persistent, in the sense that they were much better at the task after a gap of 24 hours than naive subjects. They were also much *worse* on a task where the force field was the negation of the one used in training, and this interference is suggestive of an internal model of the environment that uses the same representation in both cases.

THE ROLE OF PLANT PROPERTIES

Unlike actuators used in robotics, which often control directly for joint torques, biological muscles show interesting dynamic behaviour of their own. Although this makes muscles unattractive from the control theory point of view, it seems that the dynamics of muscles make a significant contribution to biological control. A simple way of modelling these dynamics is the *virtual equilibrium hypothesis* (van Soest and Bobbert, 1993; Bizzi and Mussa-Invaldi, 1990; Bizzi et al., 1992; Massone and Myers, 1996). According to this view, muscles are arranged in pairs, one flexor and one extensor for each joint

(this is a slight simplification as primates have more muscles than this). Each muscle acts somewhat as damped spring, where the equilibrium position and spring constant are given by control inputs. An equilibrium position for a pair of muscles gives an equilibrium position for the limb; the spring constants give the stiffness. If the equilibrium position is varied in time, the limb will describe a trajectory (we now speak of a 'virtual equilibrium' trajectory, as the limb does not actually settle to equilibrium, except at the end of the movement). One important result of these springlike properties is that they provide a simple form of error feedback, and one that operates much more quickly than those that rely on sensory feedback. Muscle properties have been shown to be able to account for many features of reaching movements, implying that planning a reaching movement may consist of specifying a set of virtual equilibrium points.

PLANNING THE MOVEMENT

The simplest way to plan a path is simply to draw a straight line between the initial and goal states. This will produce very different path shapes, depending on whether the states are described in Cartesian co-ordinates, or in terms of joint angles. If planning is performed in Cartesian co-ordinates, then the equivalent joint angles have to be calculated for every point on the path. I.e. the *inverse kinematics* mapping has to be found. Alternatively, planning could occur in joint angles, and then the inverse kinematics would only have to be solved for the endpoints of the path. Unfortunately, this also causes a loss of control over the precise shape of the path. A straight line in joint space will correspond to a curved path in Cartesian co-ordinates, and the precise shape of the curve depends on the configuration of the arm. This can become important in cases where there are constraints on the motion, *e.g.* obstacles, because these constraints will usually be described in Cartesian co-ordinates. Considerable psychophysical evidence exists supporting the claim that primates plan in terms of joint angles (Bizzi and Mussa-Invaldi, 1990), which implies that humans are capable of at least approximating a solution to inverse kinematics.

2.8 Modelling of Biological Systems

Most modelling of biological circuits occurs at a fairly low level (Selverston, 1994), with the models having dedicated variables for the properties of gap junctions and membrane

currents. This is in contrast to a lot of neural network modelling studies, in which the system is treated as a completely black box. However, there is an increasing emphasis on more abstract models. Part of the reason for this is that the enormous amount of detail can easily obscure the overall properties of even the smallest of networks. Additionally, it is very hard to decide which aspects of neural functioning are 'mere implementation details', and which are essential to the functioning of a particular network. It is useful to think of these issues in terms of Marr's levels of modelling, and types of theories (Marr, 1977). Marr distinguishes types of theories by their complexity: a type III theory is one in which the theory is of the same complexity as the data itself. This arises when the phenomena are very complex. Type I and type II theories are those in which it is possible to find general principles which apply to other data.

The detailed data at the physiological level corresponds to type III theory, but the relationship between this level and higher ones is not particularly straightforward. This is because data at the type III level is only a weak constraint on higher levels: detailed knowledge of the functioning of the individual parts of a system is not necessarily very informative about the functioning of the system as a whole, as has been learned from nonlinear dynamic systems theory (Verhulst, 1989). This seems to be particularly true in the case of neural circuits. On the other hand, knowledge of the task constraints (a type I theory) does not specify the implementation at all tightly, as a wide variety of schemes are seen in biology to solve (apparently) related problems.

To take a simpler example, imagine that we are faced with the task of explaining flight in animals: how are we to understand the mechanisms that underlie winged flight? A computational theory is concerned with the demands of the task itself, and so is not just attempting to model experimental data. Rather, the goal is to develop a *theory* of flight, in which the data is explained; the theory tells us why the data appears the way it does, and makes predictions about the way in which other systems would behave. Presumably, the theory would contain such concepts as aerodynamic lift, drag, and turbulence. However, there is an obvious problem - such a body of theory already exists, yet flight in animals is not well understood at all, mainly because the theory deals very well with static structures, but animals use flexible, flapping wings to generate lift. Obviously, there is no

evolutionary pressure to generate structures that are easily explained by existing theory. The implication is that a theory of the computational demands of a task is not completely synonymous with understanding the functioning of biological systems; this is reflected in the separate, but connected fields of robotics and biological motor control.

From the point of view of dynamic systems theory, all CPGs appear very similar, whether they occur in vertebrate or invertebrates. The basic form is that of a nonlinear oscillator; the oscillatory behaviour is realised as a limit cycle attractor of the system, thus giving a basic stability against perturbations. Any sensory inputs present have the effect of resetting the phase, and descending inputs effectively change the parameters of the system, often resulting in a bifurcation. The first mechanism provides a way of ensuring the system is sensitive to changes in the environment; the second facilitates control by a higher level structure.

2.9 Implications for Neural Network Approaches to Control

Some researchers have taken the design concepts in biological control systems, and attempted to apply them to solving difficult control problems. In some cases, the biology has been quite faithfully reproduced (ref for Ermentrout); in others, the similarities are more abstract (Beer et al., 1992; Berthier et al., 1993; Berthier et al., 1992). One goal of this kind of *Computational Neuroethology* is to understand exactly what computational complexity is necessary to perform certain tasks. A similar issue has been introduced above; i.e. how detailed do models of biological systems have to be? Biological systems have other constraints that partly determine their form, for example, previous evolutionary history, and the fact that some of the biological hardware is there to keep the animal alive. Studying artificial agents in simulated environments gives a way to perform a kind of *Comparative Biology*. As these constraints will not be operating for the simulated agents, their structure should be more informative about the demands of the *task*.

Insect locomotion is a particularly well-studied example. As has been discussed, a great deal is known about the physiology of animals such as the cockroach, as well as the behavioural patterns produced in locomotion. It has already been suggested, here and elsewhere, that there are problems in understanding the functioning of such motor circuits

from detailed, low-level data.

In the early work of Brooks *et al* it was shown that a collection of finite-state automata (FSA) was capable of reproducing the gaits and gait transitions observed in the cockroach. In subsequent studies, Beer *et al* showed that a wide range of architectures could produce the same behaviour, none of which demanding the precise timing imposed in FSA (Beer et al., 1992; Beer and Gallagher, 1992). These results have been generalised further, by showing that a wide range of connectivity patterns could generate similar data (Buchanan,), and that the form of the underlying oscillations can be varied without changing the global behaviour (Collins and Richmond, 1994)

In (Taga et al., 1991), a computational model of bipedal locomotion was built. The model included oscillators that showed the same basic function as those known to exist in human spinal cord, together with plausible reflex loops. The locomotion was controlled by one descending parameter which determined the speed of locomotion. Simple equations of motion were used for the limbs, which meant that they had their own oscillatory dynamics. Several interesting points came out of this study. Firstly, stable locomotion was realised by entrainment between the neural system, the plant dynamics, and the environment. The combined controller, plant and environment formed a global limit cycle, which meant the system was resistant to perturbations in the position of the limbs. The locomotion was also shown to be able to cope with changes in the parameters of the environment: small uphill gradients and smaller downhill ones were successfully negotiated. Finally, the control parameter smoothly varied the speed of locomotion, but the system went through a phase transition - a change to a running gait - at a critical value.

This work is in contrast to robotics approaches to legged locomotion. Firstly, no there is no explicit planning of the motion, and hence no desired states. Rather, the task is represented as a limit cycle of the entire system. Secondly, there is no explicit plant model. Despite this, there is a simple relationship between the control parameter, and the behaviour. These points are related, as can be seen in a simple minded use of standard control techniques. The task of walking can be represented as a desired trajectory in the configuration space of the limbs, although finding the trajectory that corresponds to the desired walking patterns may not be a straightforward problem in itself. In purely

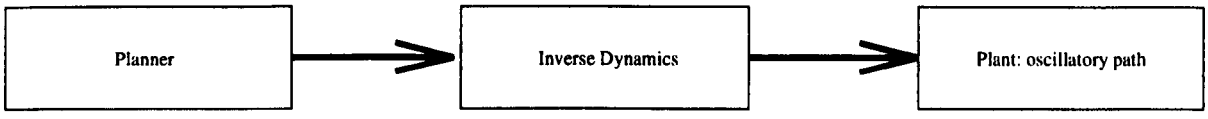

FIGURE 2.2.

feedforward control, the presence of an inverse dynamics model makes it possible to plan in terms of the state of the plant; i.e. it gives the control inputs to drive the plant through *any* sequence of states, assuming such controls exist. Obtaining an inverse model can be very difficult, particularly for high-dimensional, nonlinear plants. The desired trajectory would be an oscillation in configuration space. The scheme is shown in Fig 2.2.

As the plant has oscillatory dynamics anyway, it makes little sense to invert the plant dynamics, plan an oscillatory path, and find that a very large set of control inputs would produce the desired output trajectory. The rationale behind using a full inverse dynamics model is that the plant can be driven to any of its possible states, but tasks such as locomotion probably requires the plant to move through a very small fraction of its available state space.

2.10 Conclusions

The structure of biological control systems are the result of many interacting constraints, of which the control task is just one. Despite the variety seen in animals, and the evolutionary niches they occupy, there is a large degree of convergence in their basic design. Central Pattern Generators and motor programs are powerful concepts in understanding the function of these system. Modelling of such systems occurs at a variety of scales, depending on the particular animal, although it seems possible that an abstract computational theory may emerge.

Convergence in evolution is suggestive of strong design principles, as in the case of evolution independently producing flight from different routes. One way to investigate these hierarchical designs is to use them in learning control tasks. The main goal of the rest of this thesis is to implement Pattern Generators in a neural network framework,

and examine the effects on learning motor control tasks. The control tasks will be fairly simple, and generally involve the use of a simulated robot arm. A variety of methods for building CPGs is discussed, and the way in which they may be used in direct and indirect control.

3 Motivation for Experimental Work

3.1 Introduction

One of the striking features of biological control systems is their similarity across species. The control systems are hierarchical in design, with a careful balance between central and peripheral components used to produce adaptive behaviour. The central components include the powerful concept of the Central Pattern Generator. These have been described as dynamic systems, that take as input relatively simple signals, and produce complex, temporally structured outputs. For plants with many degrees of freedom, they are expansive systems that take low-dimensional inputs and co-ordinate the many individual actuators. Consequently, their presence reduces the dimensionality of the control task for a higher system, and may simplify the problem in other ways, such as reducing nonlinearity, or restricting the plant to tractable parts of its state space. It was also seen that much of the data on gaits and gait transitions may be succinctly explained by the interactions between sets of CPGs.

The obvious question is whether these ideas have any utility in robotics and motor control tasks. The main goal of this thesis is to investigate the use of such modular systems for neural net learning of control tasks. In the field of neural networks, there has recently been a good deal of interest in modular architectures, and incremental learning methods (Elman, 1993). For motor control tasks, it seems natural to draw a close analogy between these methods for improving learning, and the modular architectures seen in biology.

In adaptive control, the central and peripheral components usually correspond to inverse dynamics and feedback loops. In such a system, the controller outputs have direct

effects on the plant: for a robotic arm, the controller directly specifies the torques at each of the joints. Perhaps the most obvious way to implement modularity is at the level of the controller. In its simplest form, a pattern generator would be conceived as a system placed between the controller's outputs, and the plant itself. The pattern generator takes the outputs from the controller, performs some transformation, and gives the input to the plant. In this thesis, the controller and the pattern generator are implemented as fully recurrent nets. This makes it straightforward to consider different connection schemes between the two components, such as including feedback from the pattern generator to the controller's inputs. The use of recurrent networks also implies that the pattern generator will have its own dynamics, and so there can be a variable division of labour between the controller and the CPG.

3.2 Structure of Experimental Work

Chapter 4 is concerned with learning in recurrent nets, and the use of recurrent nets for abstract pattern generation tasks. Training recurrent nets involves changing the parameters of nonlinear dynamic systems, and this causes the instabilities that are seen during the course of learning. At least some of this instability arises from the existence of bifurcations in the learning process. In the neural network literature, incremental learning techniques have been used to improve the performance of recurrent network learning (Giles et al., 1992; Elman, 1993) These generally involve training a single network on a series of tasks, gradually increasing in complexity.

The goal of this chapter is to investigate a slightly different approach, in which parts of the learning system are trained on simpler tasks, and then fixed during later learning. The fixed parts are called Central Pattern Generators, by analogy to the pattern generation systems seen in biology. The remaining parts of the learning system are trained with Distal Learning, which accounts for the presence of the CPGs by backpropagating errors through them. In section 1.4.2, distal learning was seen as a way of accounting for the behaviour of a plant, assuming the existence of a sufficiently accurate plant model. Chapter 4 represents an extension of that work, and considers distal learning to be a mechanism for performing credit assignment in a system with a number of components, fixed or adaptable, and where

the components may differ in their type (e.g. recurrent or feedforward).

It is shown that the use of artificial CPG nets are a powerful way of representing prior knowledge for a task, and so constraining the behaviour of the system. This is reflected in an improvement in learning, both in terms of error and stability. The CPGs themselves are built with two methods: learning on a simpler task, and a Genetic Algorithm. Both are shown to be effective at building CPGs that help later learning. One interesting finding is that the optimal CPGs found by the GA are unstable dynamic systems, which implies that some bifurcations do not disrupt learning. Other schemes are also considered, in which part of the system is reduced in complexity to a feedforward net.

Chapter 5 applies these ideas to motor learning tasks, involving a simulated redundant robot arm. This chapter is concerned with supervised learning tasks in which a plant model is available, and distal learning is used to account for the presence of the CPGs. The learning tasks are simple reaching-type movements, in which the control system is to generate a family of movements parameterised by sensory inputs. The CPGs are used to represent a kind of task decomposition, and it is shown that it is relatively easy to build CPGs to represent elements of pattern generation, constraint satisfaction and local feedback. This work is related to other research on the use of motor programs in learning families of movements. The work of Lipitkas *et al* (Lipitkas et al., 1992; Lipitkas et al., 1993; Bock et al., 1993; Bock et al., 1996) implements motor programs as waveform generators, which take as input a set of parameters specifying the movement, and produce a time series of torques as output. The main difference in the work presented here is that the motor programs are also learned, rather than being hard-wired.

Finally, chapter 6 investigates the use of direct methods, including reinforcement learning. The reinforcement literature has concentrated mainly on the derivation of new algorithms, and analysing the convergence properties of these algorithms. As a results, many of the problems used have been little more than toy examples. As reinforcement techniques typically involve very long training times, learning difficult control tasks with reinforcement learning is often impractical. A popular approach to more demanding problems is the use of *shaping*, where the complexity of the problem is gradually increased (Gullapalli, 1992, chapter 4). The work presented in Chapter 6 takes a slightly different approach,

and considers an agent which has limited control over a complex motor program.

In direct control, the effects of control actions are unmodelled, and this fact may be used to build controllers whose outputs have complex consequences. In one set of experiments, the controller outputs are used to set the strengths of couplings between sets of oscillators, and it is shown that this is an effective way to generate different gaits. The final set of experiments consider the use of models that operate on simple measures of the plant's behaviour. The learning and use of such models is discussed in the context of learning phase relations between a set of coupled oscillators.

4 Recurrent Networks and Motor Pattern Generation

This chapter considers the use of recurrent neural networks for pattern generation tasks. Firstly, the issues involved in training recurrent networks are discussed, and the recurrent backpropagation algorithm is introduced. The first experiments are concerned with learning a single temporal pattern: an oscillation of a fixed frequency. Learning of oscillatory behaviour is shown to suffer from instability, and methods for controlling learning rates for individual weights are shown to improve performance.

The remaining experiments consider the learning of variable oscillations, either variations of frequency, or relative phase. In section 4.8, a modular architecture is discussed, in which one network (called a CPG) is trained on a simpler version of the task, and then used as a subnet in a larger network. It is shown that this improves learning performance, in terms on convergence rate, and stability of learning.

Section 4.8 investigates the use of Genetic Algorithms to build the CPGs, and it is shown that the GA can find CPGs which outperform those built by learning a simpler task. Finally, section 4.9 discussed the use of static networks in feedback loops with recurrent networks.

4.1 Introduction

This chapter is concerned with temporal pattern generation, and dynamic systems that generate temporal patterns. The motivation for studying dynamic systems of this type lies partly in the desire to understand biological systems, such as the neural mechanisms that generate the complex motor controls seen in locomotion. In invertebrate systems, measurements can be made at the level of individual neurons, and so much of the exper-

imental data is at an extremely fine scale. Nevertheless, it is now clear that there are general organising principles, and so more abstract studies may be useful. In any case, such fine-scale analysis is less useful in vertebrate systems. It is also desirable to understand how these pattern generation systems fit into a wider control scheme, which includes sensory feedback, the dynamics of the plant, and descending command signals.

However, there is also research interest in pattern generation as a subject in its own right, particularly in the context of robotics. Much recent work in the control of walking systems has concentrated on building controllers that are clearly inspired by biological pattern generators (Beer et al., 1992; Brooks, 1991) and show much the same overall structure as biological systems. This approach may turn out to be much more successful than a traditional engineering strategy; nevertheless, it is essential to understand *why* particular architectures are more successful than others, and that implies that pattern generation needs to be understood in itself.

As general purpose dynamic systems, recurrent neural nets are clearly of interest in pattern generation, and optimisation methods such as recurrent backpropagation are a powerful technique for synthesising dynamic systems. In the context of understanding biological systems, recurrent nets can provide a way of modelling experimental time series data, and in a way that can be tied closely to the known physiology. For example, the learning algorithm of Selverston (Rowat and Selverston, 1991) is capable of modelling nonlinear time series. More importantly, it has explicit terms to account for the parameters of invertebrate circuits. This provides a way to include detailed knowledge we may have about a particular circuit; it also provides a way to generate testable hypotheses about unknown parameters. These ideas have found particularly usefulness in attempting to understand the functioning of invertebrate motor circuits (Tsung et al., 1990; Selverston, 1994). More generally, recurrent networks can be considered as performing a mapping from a time series of inputs to a time series of outputs, and so they have found applications in tasks with a temporal component, such as constraint satisfaction problems in vision, speech processing and financial forecasting. (Grossberg, 1986; Elman and Zipser, 1988)

In context of robotics, recurrent nets are attractive as a way of implementing complex controllers, and recurrent nets represent very little in the way of designer bias. Some

researchers have found that general optimisation methods such as Genetic Algorithms are able to build recurrent controllers that can solve difficult control and navigation problems; other research has concentrated on the use of supervised learning (Eckmiller, 1989; Floreano and Mondada, 1995).

4.2 Training Recurrent Nets for Pattern Generation

Recurrent neural nets represent a very general class of dynamic system. The Stone-Weierstrass theorem was originally used in neural networks to show that a sufficiently large feedforward net is capable of approximating any smooth function to the desired degree of accuracy. In (Doya, 1993), the Stone-Weierstrass theorem was extended to recurrent nets, showing that a sufficiently large net could approximate any dynamic system to the desired degree of accuracy. However, as with feedforward nets, a more interesting question is whether a particular problem can be *learned*. There are several learning algorithms for these types of nets, each making slightly different assumptions.

LEARNING FIXED POINTS

Fixed points are those points in the state space where a system is stationary. *Learning* fixed point generally involves making the points stable as well as stationary; this makes the point an *attractor*. The general approach here is to assume that the net *does* show convergent dynamics. Under that condition, the equations for the error derivatives show the same convergent behaviour as those governing the dynamics, and they are relaxed until a fixed point is reached. The units have dynamics that will be used widely in this thesis:

$$\frac{dx_i}{dt} = -x_i + \sigma\left[\sum_j w_{ij}x_j\right] + I_i \tag{4.1}$$

or

$$\frac{dx_i}{dt} = -x_i + \sigma(y_i) + I_i$$

if y_i is used as shorthand for the weighted inputs $\sum_j w_{ij}x_j$. w_{ij} is the weight from unit j to unit i; I_i is an external input to unit x_i, and σ is the activation function of the unit. It

was discovered by Pineda (Pineda, 1987) that the backpropagation rule is a special case of a general error correction rule, one that applies to recurrent nets. In backpropagation, the equations governing the activity and error derivatives are given by

$$x_i = \sigma(y_i) + I_i \tag{4.2}$$

$$z_i = \sigma'(y_i) \sum_j w_{ij} z_j + e_i \tag{4.3}$$

$$\frac{\partial E}{\partial w_{ij}} = y_i z_j \tag{4.4}$$

z_i is the partial derivative of E with respect to x_i, ordered in time; E is the error defined on $x(t_\infty)$ (i.e. the final state of the unit), and e_i is the simple derivative of E with respect to the unit's final state (i.e. $e_i = x(t_\infty) - x^*(t_\infty)$). Of course, a feedforward net has no dynamics, so a fixed point is reached immediately. The interesting point is that these equations still hold when there are recurrent connections, as long as a fixed point is reached. This can be seen by considering what happens when the net is stationary: the RHS of equation 4.1 becomes zero. In that case, equations 4.2, 4.3 and 4.4 are satisfied. Fixed points for equation 4.1 are found by relaxing to a solution; the same may be done for the z values, using the equation:

$$\frac{dz_i}{dt} = -z_i + \sigma'(y_i) \sum_j w_{ij} z_j + e_i$$

This scheme assumes that the net *will* relax to a fixed point, and not, for example, produce oscillations. There are many ways to guarantee such convergent behaviour, such as using zero-diagonal symmetrical weights (so that $w_{ij} = w_{ji}; w_{ii} = 0$). One peculiar property of fixed point algorithms is that these restrictions don't seem to be necessary; applying fixed point algorithms to unrestricted weight matrices seems to produce stable networks.

Learning fixed point behaviour often correponds to building a pattern completion device, in which initial patterns corresponding to a noisy or incomplete pattern are transformed into some prototypical pattern. The most obvious use of such a system is as a content addressable memory (Hopfield and Tank, 1985): the attractors of the system correspond to the memorised patterns.

Learning non-fixed point behaviour

Fixed point dynamics define a class of behaviours for the system, which correspond to the system always converging to a stationary state. Recurrent networks are capable of producing a larger set of behaviours, which includes oscillatory and chaotic dynamics. Learning such non-fixed point behaviours requires the use of more complex optimisation techniques.

Some researchers have used networks with reduced forms of recurrency, in which there are feedback connections between some parts of the network and the input layer, but these connections are fixed. The fact that the recurrent connections are fixed means that standard backpropagation may be used (Elman, 1990).

In a fully recurrent network, there may be modifiable connections between any pair of units. There are two main approaches here: the first is to unroll a feedforward net for every point in time. This is discrete-time algorithm, in which the dynamics of the network are considered to be an *iterated function system.* An iterated function system is defined by a static mapping. This mapping takes the state at time t, and maps it onto $t+1$. The new state is then fed in as the new input. This suggests that we may consider the state of the system at $t+n$ to be the result of n functions, cascaded in sequence. When the system is a neural network, training for n time steps results in training one large static network, which is basically n copies of the one network, with output of the network i forming the input to network $i+1$. The drawback to this method is that the storage requirements grow very quickly; the advantage is that standard backprogation may be used.

Williams and Zipser (Williams and Zipser, 1989) showed how to perform backpropagation without storing n separate networks. This procedure still makes use of the discrete time dynamics, but now differentiates the dynamics with respect to the weights at each time step, to get a relation with the derivative at the previous time step.

An alternative approach considers the network to be a continuous-time system, and a learning rule may be derived by the calculus of variations, or by making a finite-time approximation. One example of this is Time-Dependent Recurrent Backpropagation (Pearlmutter, 1990); this algorithm is discussed in more detail below.

4.2.1 Issues in Training Recurrent Nets

Training recurrent nets is considerably more difficult than feedforward nets, as this involves adapting the parameters of a nonlinear dynamical system. The early stages of learning are particularly difficult, when the dynamics of the net are likely to be very different from the target dynamics. Learning algorithms for recurrent nets implicitly assume that the net is on, or near the desired trajectory; the error derivatives are not informative when the net is far away from the target. One way around this is to use *teacher forcing*, where the outputs of the net are literally placed on the target trajectory after the error derivatives have been calculated, thus ensuring that the net never wanders too far away from the target. Teacher forcing was found to be necessary to learn limit cycles with the Williams and Zipser rule, and can drastically speed up learning of other tasks. Teacher forcing is usually used with discrete time networks, so that the outputs are forced to the target values at each time step, after the error derivatives have been calculated. It is also possible to perform teacher forcing with continuous time nets; the teacher is implemented as a forcing term added to the net's dynamics, where the magnitude of this term is reduced as learning converges.

The main problem with teacher forcing is that there is no guarantee that the net will perform as required once the teacher forcing has been removed. Also, forcing tells us how to set the value of the output units, but not the values of the hidden units. Nevertheless, teacher forcing is widely used in the learning of difficult temporal tasks, and can turn completely unlearnable problems into tractable ones.

A rather elegant alternative to teacher forcing is phase-space learning (Tsung and Cottrell, 1995). This only really applies to modelling autonomous systems (i.e. those that take no inputs). The plant is modelled as lying on an attractor. The attractor not only specifies the long term behaviour of the plant (the trajectory of the attractor itself), but also the vector field surrounding the attractor. The vector field specifies the velocity of the system at all point in the state space: it is basically the value of the RHS of the equations of motion for the system. If the equations of the plant were actually known, then the vector field would be known for all points in the state space. This is rather unlikely, but some simple assumptions can enable us to reconstruct the vector field for an unknown system. Basically, an attractor is so called because trajectories nearby will be attracted towards it,

usually in a simple (i.e. exponential) fashion. Given this assumption, the vector field can be generated away from the target trajectory. So, as the net wanders away from the desired trajectory, new training data is generated that will still be informative. Additionally, the stability of the attractor has been explicitly represented during learning, which is not the case if the net is merely trained on one trajectory. Finally, knowing (or approximating) the vector field means that feedforward structures may be used for the approximation, in which case learning trials can be randomised. When learning is complete, the feedforward net is converted into a dynamic system by considering it to be an iterated map. The only real problem with this is that there is no clear relation between the magnitude of errors in learning the vector field, and errors of the dynamic system, simply because iterating an inaccurate mapping can mean that the errors will become amplified.

INCREMENTAL METHODS

As with other difficult learning problems, incremental methods have been widely used in training recurrent nets (Giles et al., 1992). On the whole, this means that a net is initially trained on a much simpler version of the task. Once the performance is judged to be sufficiently good, the complexity of the task is increased. This is essentially a *temporal* method, as the same net is used throughout. An alternative strategy, used widely in this thesis, is to take a *structural* approach. This means that a network will be trained on a simpler task, and then used as a component in a larger system. This is clearly related to the 'chunking' methods seen in AI (Iba, 1989), and may also give insights into the hierarchical structures seen in biological control systems.

4.3 Introduction to the Experiments

Most biological motion is characterised by oscillatory motor activity; this includes such diverse behaviours as legged locomotion, swimming, flying and chewing. Underlying all of these behaviours are neural circuits called Central Pattern Generators. The main components of this behaviour are the initiation and termination of oscillations, and the modulation of the basic pattern to suit the demands of the task at hand. In the experiments reported here, recurrent nets were trained on abstract oscillatory motor tasks, using recurrent backpropagation. The actual patterns to be learned are fairly arbitrary,

and consist of sine waves at various frequencies and relative phases. The use of recurrent backpropagation for these simple tasks is a case of computational overkill, as simpler methods exist for co-ordinating oscillators (Doya and Yoshizawa, 1990). However, the goal of this chapter is to investigate the performance of more general methods, which make no assumptions about the nature of the patterns.

Architecture:

The nets considered are fully connected, recurrent, real-time nets. The dynamics of each unit evolve according to :

$$T_i \dot{x}_i = -x_i + \sigma\left[\sum_j w_{ij} x_j\right] + I_i(t); \quad w_{ij} \neq w_{ji}$$

where $I_i(t)$ is an external input, T_i is a time constant for unit x_i. The activation function σ is taken to be the usual sigmoid function:

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

Each unit is connected to every unit, including itself. Unlike Hopfield nets, the weight matrix is not constrained to be symmetrical, as this restricts the dynamics to fixed point behaviour. Networks with continuous dynamics were chosen, instead of discrete time nets. There are several advantages to using continuous time net. Firstly, a variety of methods may be used to integrate the equations, not necessarily the simple Euler method that is implicit in discrete time systems. It is not yet clear how much of the instability found in training recurrent nets is due to the choice of integration method. Even if such a simple method is used, the size of the time step may be varied, for example, if the dynamics are known to be particularly nonlinear. Also, if recurrent nets are used in conjunction with other dynamic systems, say in the control of a simulated plant, then the equations of the combined system may be integrated together; it is common to use higher order methods when simulating physical systems. Finally, the use of continuous dynamics automatically gives a degree of temporal smoothness to the network's behaviour, smoothness that may otherwise require the use a regulariser. In these experiments, a simple Euler method was used, with a fixed step size of $0.01s$.

Learning Algorithm

Time-Dependent Recurrent backpropagation is a supervised method for adapting the parameters of the net. This assumes the existence of target values for some set of units $i \in \mathcal{T}$ to be considered the net's outputs, although these targets need not appear at every time step. Gradient descent is used to minimise a cost functional:

$$E_i = \int_{t_0}^{t_1} (x_i - x_i^*)^2 \; dt; \quad i \in \mathcal{T}$$

where x^* is the target value for unit i. $\mathcal{T}$ is just the set of units for which there is a target value; this may change during the time period, with targets appearing for some units at some points in time, and others for other points. In this sense, recurrent nets do not have such a fixed notion of input and output units.

There are several algorithms for performing gradient descent on these kinds of cost functionals; Real Time Recurrent Backpropagation was used for all the experiments in this chapter. This estimates error derivatives by solving another set of differential equations. If we denote by $z_i(t)$ the time-varying error derivative $\frac{\partial E}{\partial x_i}(t)$ for unit i, then the dynamics of $z_i(t)$ are given by the following equation:

$$\frac{dz_i}{dt} = \frac{1}{T_i} z_i - e_i - \sum_j \frac{1}{T_j} \sigma'(y_j) z_j$$

where y_j has been used as shorthand for the weighted inputs $\sum_j w_{ij} x_j$

The boundary condition $z(t_1) = 0$ is used, so these equations are integrated backwards in time. The error derivatives for the weights are given by

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{T_j} \int_{t_0}^{t_1} x_i \sigma'(y_j) z_j$$

and for the time constants

$$\frac{\partial E}{\partial T_i} = -\frac{1}{T_i} \int_{t_0}^{t_1} z_i \frac{dx_i}{dt} dt$$

This gives error derivatives for the weights and the time constants, and simple gradient descent was then used to minimise the error. The term 'parameters' is used to refer to the modifiable parameters of the network, i.e. the weights and time constants. All the code for the network dynamics and learning was written in 'C', and the simulations ran on an HP 700 series Unix workstation.

As with forward modelling, error derivatives $\frac{\partial E}{\partial x_i}(t)$ for the unit's activations are also calculated; these may be used to backpropagate through a recurrent net, in order to optimise the inputs to the net.

4.4 Learning stable oscillations

It is easy to show that a two unit net is capable of producing oscillations. To investigate *learning* of oscillations, small networks, consisting of 6 units, were trained to approximate a sine wave $y = 0.5 + 0.45sin(\omega t)$. The target trajectory was applied to one unit, and the others were left unconstrained. The weights were randomised from a normal distribution with mean zero and standard deviation of 1.0; the time constants chosen from normal distribution with mean 1.0 and s.d. 0.1. During learning, the weights were unconstrained, but the time constants were not allowed to become lower than 0.1, as this usually leads to instability. The learning rate for the weights was 0.2; that for the time constants was 0.1.

20 networks were initialised with random weights and time constants, and trained on the target trajectory. Each training epoch consists of three steps. Firstly, the network dynamics are integrated across a fixed size time window. The time window was 3 *secs* long, corresponding to 3000 samples in time. The equations for the error derivatives were then integrated *backwards* in time. Finally, the derivatives for the weights and time constants were found by a third integration, forwards in time. This whole process gives one error derivative for each modifiable parameter of the network, and these parameters were then adapted with standard gradient descent.

Each network was trained for 10 000 epochs. The performance of the networks is just given by the RMS error for the unit at which the targets appear.

4.4.1 Results

The learning curves for this experiment are shown in Fig 4.1. The form of the curves suggests that learning suffers from instability, as there are many uphill jumps in error. This is also reflected in the long training times.

The actual fit to the target trajectory is not perfect, despite convergence of learning. The output of a typical net is shown in Fig4.2; the residual error is mainly at the extremal

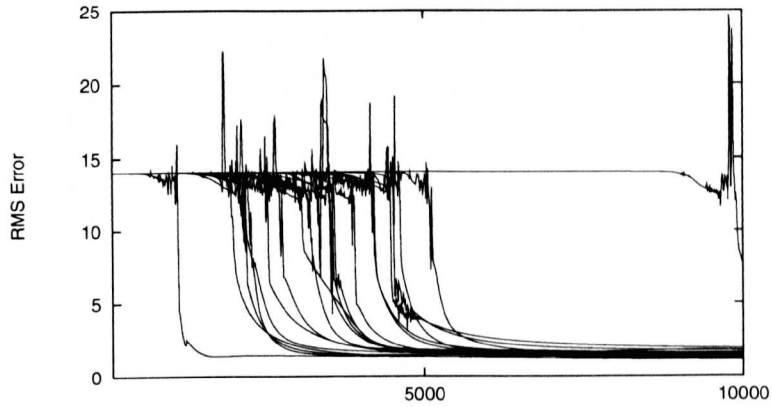

FIGURE 4.1. Learning curves for 20 networks on the single oscillation task. Learning is rather unstable, as can be seen by long periods when error is not decreased, and spikes in the learning curves

points in the oscillations. This is really a weakness of RMS error as a distance metric for trajectories: a low value of RMS error can be obtained without fitting the 'tricky' parts of the target trajectory, and so once the basic oscillation is established, convergence becomes very slow. Other methods have been investigated elsewhere (Tsung and Cottrell, 1995), which are more sensitive to the actual direction of motion of the outputs, but the main concern here is the qualitative nature of the dynamics, rather than the detail.

4.4.2 Using The Delta-Bar-Delta Rule to improve Learning

Instability in learning is often due to the solution between continuously overshot; this can be caused by steep surfaces in weight space. It is quite reasonable to expect steep error surfaces with recurrent nets, as learning involves changing the parameters of a nonlinear dynamic system, so in some regions of paramter space small weight changes may lead to large changes in behaviour (Doya, 1992). In other parts, the surface may be relatively flat, causing slow convergence. The problem is that instability seriously limits what can be done with the learning procedure, and also limits the range of applications for recurrent networks.

One way to approach this problem is to use adaptive learning rates for each parameter, and this is the basis of methods such as the *delta bar delta* rule (Fang and Sejnowski, 1990). The basic idea is to try to detect if learning has overshot the solution, for a particular

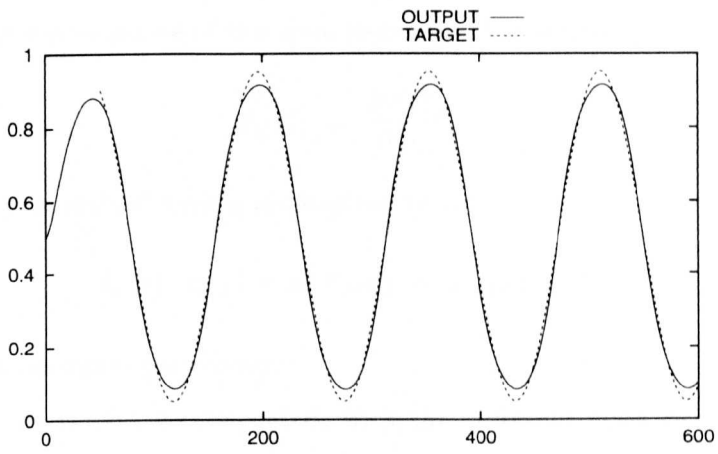

FIGURE 4.2. Performance of a typical trained network on the single oscillation task. The dotted line shows the target trajectory; the actual output of the network is shown with a solid line

parameter. This is done in a very simple way, by comparing the signs of successive parameter changes. If the sign stays constant with every step, then the learning is heading in the right direction (for this parameter), and so the learning rate can be gently increased (usually by a constant amount scaled by the current RMS error). If the error derivative ever changes sign, then this indicates that the solution has been overshot, and the learning rate is greatly reduced, usually by a fractional amount (e.g. 90 percent). The cost of using this method is that three additional parameters have been introduced (the two above, plus a momentum term for the current error derivative), and have to be set by hand. So the update rule for the learning rate η_{ij} is

$$\Delta\eta_{ij} = \kappa \quad if \delta_{ij}(t-1)\delta_{ij}(t) > 0 \tag{4.5}$$

$$-\phi\eta_{ij} \quad if \delta_{ij}(t-1)\delta_{ij}(t) < 0 \tag{4.6}$$

$$0 \quad otherwise \tag{4.7}$$

where κ is an additive increment to the learning rate, and $-\phi\eta_{ij}$ is a multiplicative decrement. κ usually related to the current RMS error, so that fine control becomes possible as learning approaches the solution:

$$\kappa = \lambda E(t)$$

$\delta_{ij}(t)$ is the current estimate of the error derivative for weight ij:

$$\delta_{ij}(t) = \frac{\partial E(t)}{\partial w_{ij}}$$

This estimate is stabilised with a momentum term:

$$\delta_{ij}(t) = (1-\alpha)\,\delta_{ij}(t) + \alpha\,\delta_{ij}(t-1)$$

where α is a momentum parameter.

The previous experiment was repeated, with the delta-bar-delta rule used to set the learning rates. This procedure introduces 3 additional parameters that have to be set by hand, and some experimentation was necessary to find combinations of values that worked. This method is really only applicable to learning one pattern, as learning input-dependent patterns will involve the error derivatives constantly changing sign even if learning is proceeding in the correct direction. 20 networks were trained on the single oscillation task, with parameter values of$\lambda = 0.01$; $\alpha = 0.1$; $\phi = 0.8$.

4.4.3 Results

The learning curves for the delta-bar-delta rule are shown in Fig4.3. The time to convergence is much shorter, although the large spike in one of the runs suggests that it is still possible for learning to lose stability.

4.5 Learning variations of the basic patterns.

In this section, networks were trained to produce variable oscillations. Two variations were considered: varying the frequency of the oscillations(**Task 1**), or varying the relative phase (**Task 2**). In both conditions, there were two output units, each receiving target trajectories. Larger networks were used for this experiment, consisting of 9 units. 20 networks were trained for each of the two tasks, with training times limited to 20 000 epochs. The weights and time constants were initialised from the same distributions as before, and the networks simulated for the same size time window. A learning rate of 0.2 was used for the weights, 0.1 for the time constants.

Task 1:Variable Frequency

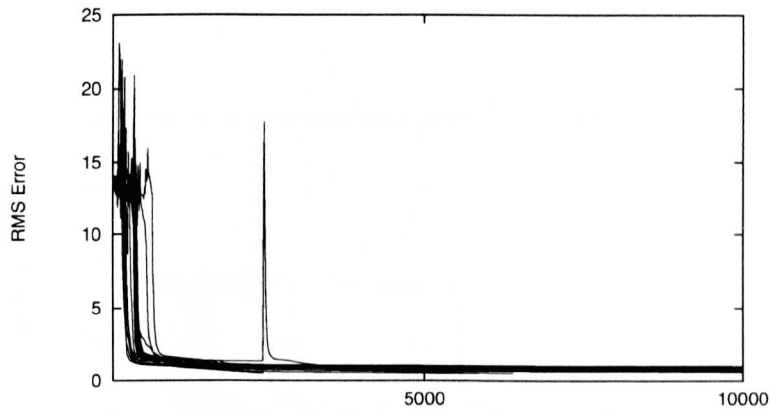

FIGURE 4.3. Learning curves for 20 networks on the single oscillation task, using the Delta-Bar-Delta rule to control the learning rates. Learning is seen to converge much more rapidly and stably, although the large spike suggests that some instability remains

There were two target trajectories for this condition: $y = 0.5 + 0.45sin((1 + I)\ \omega t)$ for one unit and $y = 0.5 + 0.45cos((1 + I)\ \omega t)$ for another. The two target units were required to oscillate at a frequency given by the external input, and to maintain a phase difference of $\frac{\pi}{2}$. The variable I controls the frequency of the desired oscillation, and this value was given as input to one unit. I was varied over the range 0 to 1, and was either chosen from the continuous interval $[0 : 1]$, or from the discrete set $\{0, 1\}$. This was simply to see whether different kinds of generalisation would be produced. Given the range of I, the highest frequency was twice the lowest.

Task 2:Variable phase

In this condition, the target trajectories were $y = 0.5 + 0.45sin(\omega t)$ for one unit and $y = 0.5 + 0.45sin(\pi I + \omega t)$ for the other. Again, I was varied over the range 0 to 1, either continuously or discretely. This meant that the required relative phase between the two units varied from 0 to π.

4.5.1 Results

The learning curves for the variable frequency task are shown in Fig 4.4; those for the variable phase are shown in Fig4.5. In both the phase and the frequency condition, learning proceeds more slowly than learning a single trajectory.

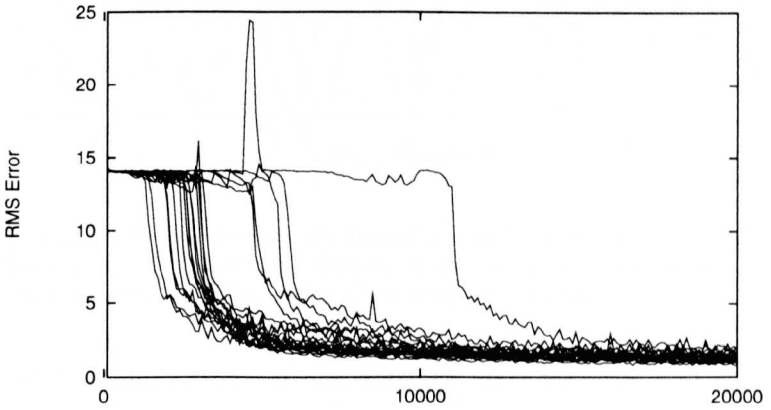

FIGURE 4.4. Learning curves for 20 networks trained on the variable frequency task

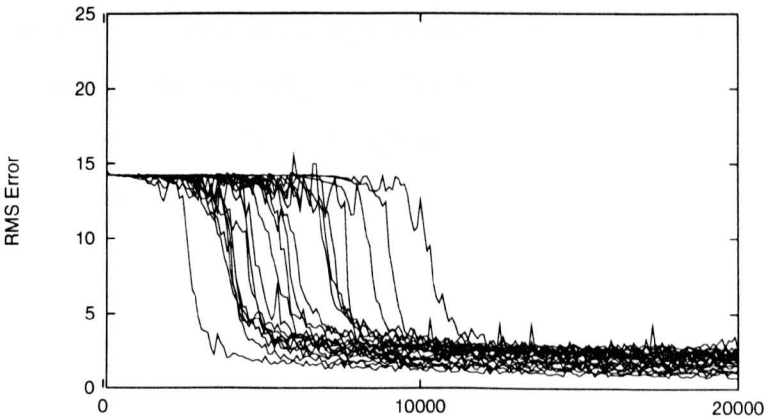

FIGURE 4.5. Learning curves for 20 networks trained on the variable phase task

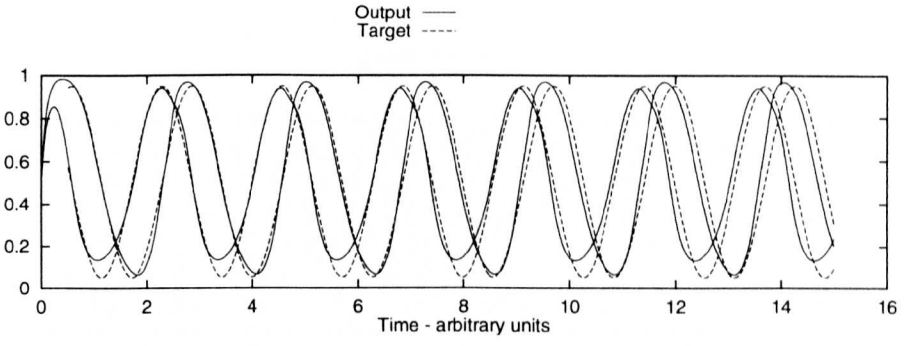

FIGURE 4.6. Performance of a typical trained network on the variable frequency task. The target appears as the dotted line, the output as the solid line. This plot shows a **low frequency** pattern

Figs 4.6 - 4.8 show the output of one network after learning had converged; the net is tested on three different target frequencies. This test output was obtained by running the network dynamics for a much larger time window than that which was used during learning. It can be seen that the output drifts slowly away from the target trajectory, and the fit to the shape of the target is similar to the single trajectory case.

Figs 4.9 - 4.11 shows the corresponding curves for the variable phase condition. The phase condition seems to be more difficult than the frequency task, as reflected by the time to convergence of the learning. Again, the output drifts away from the target, and the fit to the target is slightly inaccurate.

In both **Task 1** and **Task2**, training with inputs from a discrete set $0, 1$ produced networks which showed poor generalisation: testing with inputs other than those used in training produced mixtures of the two learned patterns. Consequently, all subsequent experiments used continuous training inputs.

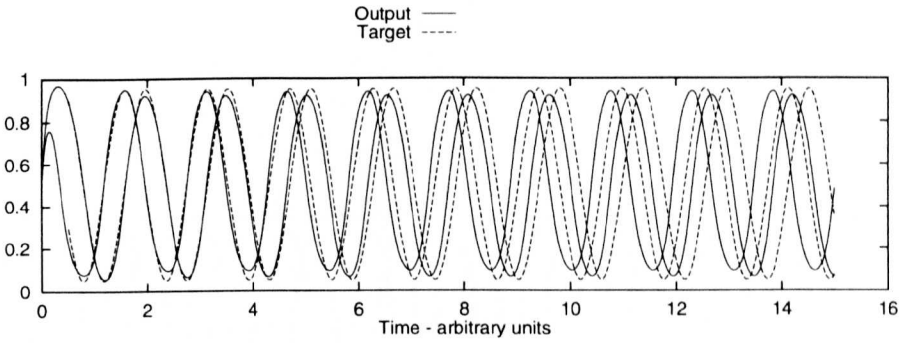

FIGURE 4.7. Performance of a typical trained network on the variable frequency task. The target appears as the dotted line, the output as the solid line. This plot shows a **medium frequency** pattern

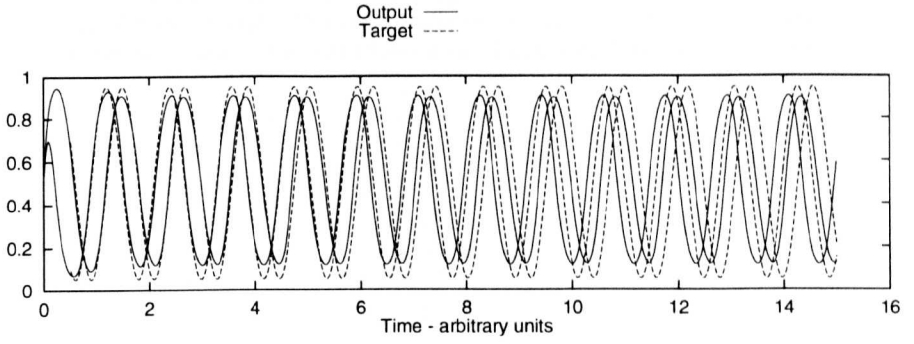

FIGURE 4.8. Performance of a typical trained network on the variable frequency task. The target appears as the dotted line, the output as the solid line. This plot shows a **high frequency** pattern

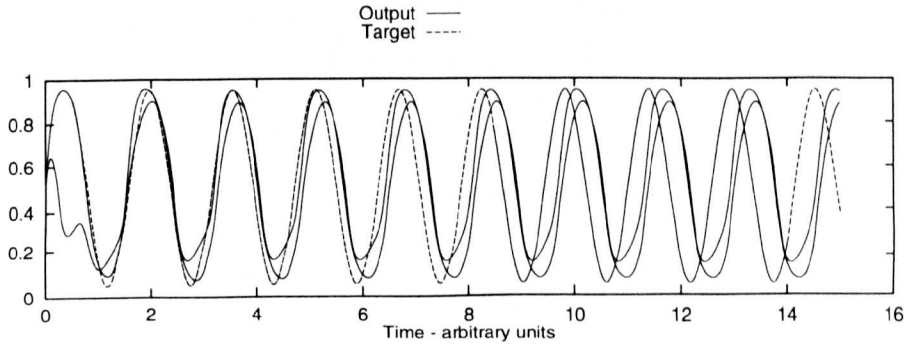

FIGURE 4.9. Performance of a typical trained network on the variable phase task. The target appears as the dotted line, the output as the solid line. This plot shows a **small relative phase** pattern

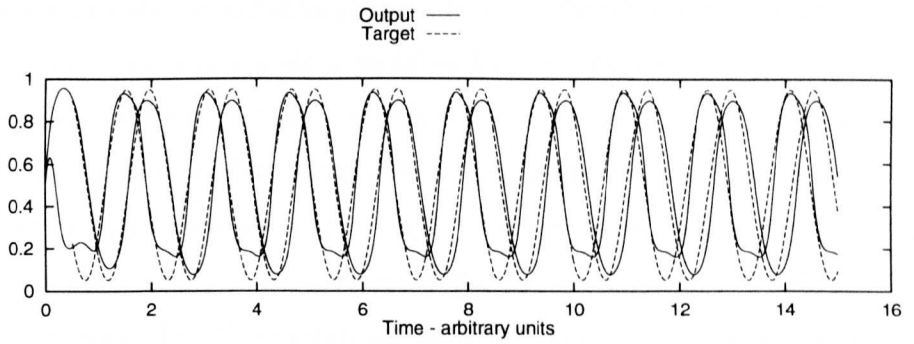

FIGURE 4.10. Performance of a typical trained network on the variable frequency task. The target appears as the dotted line, the output as the solid line. This plot shows a **medium relative phase** pattern

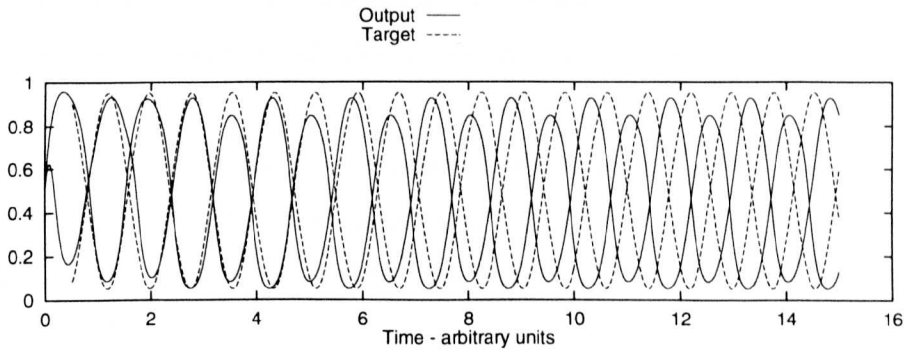

FIGURE 4.11. Performance of a typical trained network on the variable frequency task. The target appears as the dotted line, the output as the solid line. This plot shows a **high relative phase** pattern

4.6 Issues in Training Recurrent Nets: Motivation for using CPGs

It has been shown elsewhere (Tsung and Cottrell, 1995) that bifurcations can occur in training recurrent nets - qualitative changes in the network dynamics that produce large changes in error. This is obviously a problem for a gradient descent procedure; gradient descent assumes the error surface is smooth and continuous. It is precisely because recurrent nets are such powerful systems that these problems arise. Even relatively small nets can show complex behaviours, and there are often quite small regions of weight space that contain a wide variety of behaviours. If the dynamics could be constrained to being qualitatively correct, then these bifurcations should disappear, with an increase in learning performance. For the current learning task, this would mean constraining the network dynamics to being oscillatory - a system with non-oscillatory dynamics *has* to go through a bifurcation in order to oscillate.

One way to constrain the network dynamics is suggested by the basic architecture of biological control systems, particularly the existence of CPGs. CPGs were described as complex dynamic systems capable of producing temporally structured outputs in the absence of external inputs, but any inputs present served to modify the outputs to suit the task at hand. The descending inputs only have a limited amount of control over the CPG outputs, usually limited to modifying the frequency or phase of oscillations. This means that they are unlikely to be able to destroy the oscillations, or make other large changes to the behaviour.

As this general architecture is so common in biological control systems, it might be suspected that this represents a strong design principle, one that may be of use in control and robotics. There are several reasons why introducing CPGs might be beneficial in pattern generation:

- As in biological systems, CPGs can be used to introduce constraints between the various degrees of freedom of the plant. This can be achieved by other methods, such as using regularisers; the constraints implicit in a CPG will be state-dependent, and so sensitive to the behavioural context. This gives a very powerful framework for introducing dynamic constraints, one that may turn out to be more relevant for

motor control task than the use of regularisers.

- An important issue in adaptive control is *exploration*; in learning a plant model, for example, the plant should be exposed to a sufficiently rich set of inputs, so that the learner observes all the modes of behaviour of the plant. On the other hand, the plants inputs during learning should be representative of the sort of inputs that the plant will see during operation. These issues are particularly important during the early phases of learning. Using a CPG may be an effective way of addressing this issue, as this would effectively mean that the controller has a repertoire of complex actions.

- Stability is a major concern in nonlinear adaptive control. It is shown here that the presence of CPGs can improve the stability of procedures such as recurrent backpropagation. This is mainly because the CPG ensures that the output of the control system is qualitatively correct; adaptation is then a process of fine-tuning those controls.

4.6.1 Introducing CPGs into the Control System

In section 1.4.2, it was seen that a plant model may be used to transform error in the space of the plant output into the output of the controller. It was also seen the transforming system need not be a plant model, and that adding other mappings provides a mechanism for introducing constraints. These ideas can be extended further: the transforming system can be an arbitrary dynamic system, and backpropagating through this system will have very different effects, depending on the dynamics of the system. This provides an easy way to introduce CPGs. The basic set up is shown in Fig4.12.

Both the CPG and the controller are recurrent networks. The main difference with the previous experiment is that the CPG is a fixed system: its weights are not changing during the learning task. The learning task for the controller is identical to that for the networks in the previous task, except that it is separated from the outputs by the CPG.

The simplest way to connect the two nets is to have full connectivity between the controller and the CPG, and vice versa. This means that every unit in the controller

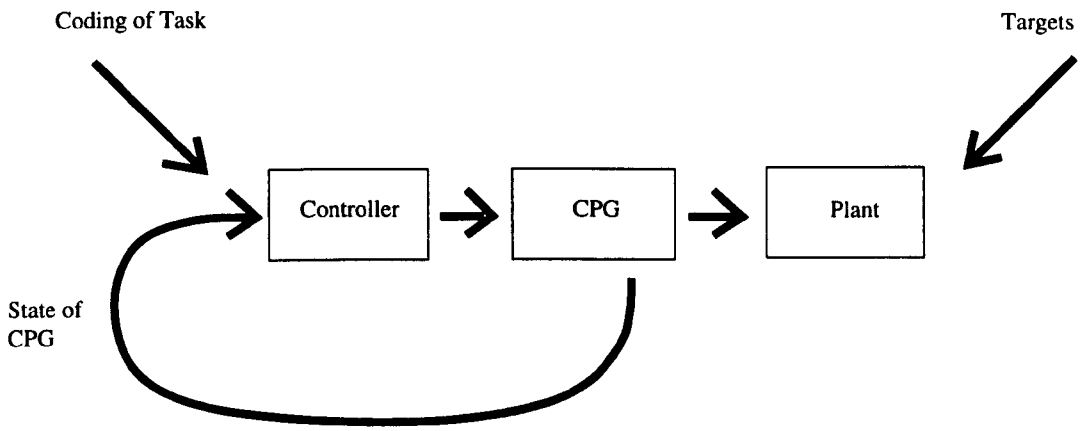

FIGURE 4.12. Adding a CPG to the control scheme. The controller is again a recurrent network, trained with recurrent backpropagation. The CPG is a dynamic system placed between the outputs of the controller, and the inputs to the plant. The CPG is implemented as another recurrent network, so error derivatives may be backpropagated through the CPG into the controller. At each time step, the controller receives as input the current state of the CPG, and sends outputs to every unit in the CPG. In effect, this is one large recurrent network, in which some of the weights are fixed

will have modifiable connections with every unit in the CPG, and vice versa. In this case, there is effectively one large recurrent net, in which a subset of the weights are held fixed. Otherwise, the learning will proceed as normal, so no changes are made to the backpropagation procedure.

It is also quite easy to implement other connection schemes, and other types of controller. For example, the system labelled the controller does not have to be of the same type as the CPG; it could simply be a feedforward net. This is just because the backpropagation procedure calculates the derivative of the error with respect to the activation of all the units in the CPG, including the inputs, without making any assumptions about the source of those inputs. The error derivative with respect to the inputs is used to give training information for controller, regardless of the form the controller takes. However, recurrent backpropagation is still required in such a setup, despite the fact that the controller is a static, rather than a dynamic system. This is because the controller is in a feedback loop with a dynamic system, so future inputs are a function of the current controller outputs, and this dynamic interaction has to be taken account of.

Once these error derivatives have been calculated, it is straightforward to get the derivatives for the controller itself. For some problems, the weights from the CPG back to the controller may be unnecessary, and in other cases, it may make sense to have the controller just set the initial conditions for the CPG, and let the CPG's dynamics generate the rest of the behaviour. Which scheme is appropriate really depends on the task, and on how much of the behaviour is going to be centralised in the CPG, and how much is left to the controller. In extreme cases, almost all the behaviour comes from the CPG, and the role of the controller is reduced to acting in similar way to the 'command neurons' discussed in 2.2. All these cases fit quite easily into a backpropagation framework.

4.7 Using CPGs from simpler task

The main problem is designing the CPG, as it is not immediately obvious what characteristics would be desirable. One possible method for building CPGs is *'shaping'*; this refers to decomposing a difficult task into subproblems, and combining the individual solutions to give a solution to the original task. There are generally two main approaches to using shaping. The first approach, which could be called *incremental learning*, corresponds to learning a series of tasks, beginning with a simple problem, and increasing in complexity, until the final task is the original problem. This usually requires monitoring the performance of the controller, so that simpler tasks are learned to a sufficient degree before more complex versions are introduced.

The other main approach could be called *structural shaping*. This involves decomposing the control task into a set of simpler problems, and training low-level controllers on those individual subproblems (e.g. navigation could be described in terms of simpler behaviours, such as turn left, turn right). These elemental behaviours are then available to a high-level controller as primitives for more complex problems. This is very much in the spirit of the use of CPGs in pattern generation and control, as the presence of the CPG means that simple outputs from the controller may produce complex behaviours. One reason why structural shaping may be more effective than incremental learning for recurrent nets lies in the fact that the weights of the low-level controllers are kept fixed

as the higher level controller is adapted. This makes it harder for the combined system to unlearn the elemental behaviours as the more complex task is learned. These issues of unlearning are particularly important in training recurrent nets, where learning consists of setting the attractors of a multidimensional system, and unlearning can force the system to go through bifurcations to re-learn the task.

4.7.1 Learning the Variable Oscillation Tasks with CPGs

The approach taken here is to train a small network (6 units) on a simple fixed oscillation task, and use the optimised net as a CPG for the more complex task. The controller (a 3 unit network) and CPG for the complex task are arranged as in Fig4.12; the outputs of the controller are fully connected to the units of the CPG, and vice versa. The combined system has the same number of units and weights as the networks in section 4.5; the only difference is that some of the weights are held constant. The path back from the CPG to the controller is not strictly necessary; however, if the controller outputs need to be oscillatory, then it does not make sense for the controller to learn oscillations from scratch, when these can be given as inputs from the CPG.

The performance of the controller depends on how well the CPG has learned the fixed oscillation task. However, it is also likely to depend on the *way* in which the simple task has been solved. This is because different CPGs are likely to have learned different hidden unit representations, which implies that they will respond to external inputs in a different way. In some cases, this will help the learning of the controller; in other cases, it will make it more difficult to learn the required variations. Consequently, each controller trained on the full problem used a 'fresh' CPG trained on the reduced problem; this was an attempt to average out the effect of different hidden unit representations.

The tasks used are exactly the same as the variable frequency (**Task1**) and variable phase (**Task2**) problems from the previous section. For all these cases, the CPG was built by training a network on a simpler version of the task: $y = 0.5 + 0.45sin(1.5\ \omega t)$ for one output, and $y = 0.5 + 0.45cos(\frac{\pi}{2} + 1.5\ \omega t)$ for the other. This pattern is the 'mean' of both the variable frequency and the variable phase patterns. The weights of the CPG were then frozen. The remaining weights and time constants for the controller network

were chosen from the same distributions as before (N(0,1) and N(1.0,0.1) respectively), and the learning rates for the weights and time constants set to 0.2 and 0.1 respectively.

The combined net was trained on the same patterns as before. In the frequency case this is $y = 0.5 + 0.45sin((1 + I)\ \omega t)$ for one unit and $y = 0.5 + 0.45cos((1 + I)\ \omega t)$ for the other. For the phase condition, the patterns are $y = 0.5 + 0.45sin(\omega t)$ for one unit and $y = 0.5 + 0.45sin(\pi I + \omega t)$ for the other. 20 networks were trained on the variable frequency task, and 20 on the phase task.

4.7.2 Results

The learning curves for the frequency condition are shown in Fig4.13; those for the phase condition are in Fig4.14. The results show a dramatic improvement in learning compared to training a single net (4.5.1); the learning is seen to proceed much faster, with convergence occurring in 1500 epochs. This is one effect that was to be expected, and it was also expected that the stability of learning would improve. Stability of high-dimensional systems is hard to measure, but a rough idea of stability improvement can be gained by using different learning rates. This experiment initially used a fixed learning rate of 0.2; however, the learning rate may be increased to around 0.7, with convergence still obtained. Using such a high learning rate for the single net architecture used above generally prevented learning.

STATISTICAL ANALYSIS:

To compare learning curves, the curves have to be parameterised in some way. The problem with the learning curves presented here is that there is no obvious function that describes their shape (as would be the case in some learning situations, where learning curves are well-described by exponential decay). There are two obvious ways of giving each curve a measure - the final error, averaged over some number of trials, or the time taken to reach an error threshold. In this case, the learning curves all asymptote to roughly the same value, so the final error is not particularly informative. Instead, the performance was measured by the time taken to reach an error threshold, in this case 5.0. As the learning curves are quite noisy, they were first smoothed over 300 epochs.

To compare the CPG method with training a single net, the cost of training the

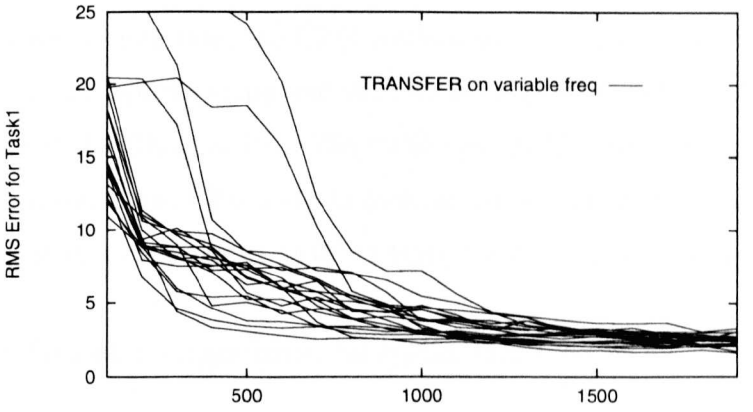

FIGURE 4.13. Learning curves for 20 controllers on the variable frequency condition, with a CPG from 'shaping'. Note the values on the x-axis; learning converges within 2000 epochs

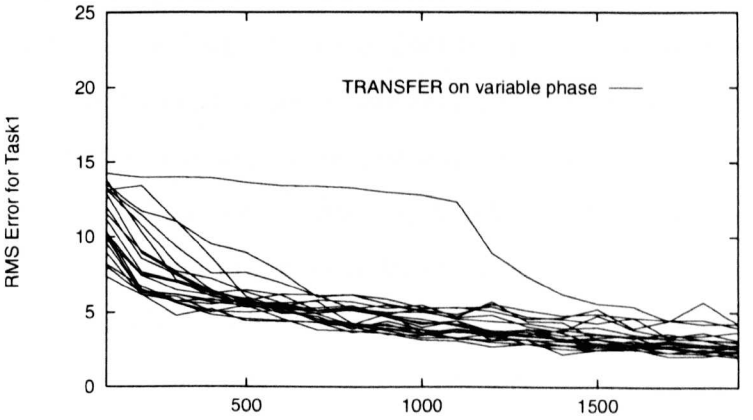

FIGURE 4.14. Learning curves for 20 controllers on the variable phase condition, with a CPG from 'shaping'. As with the frequency condition, learning is now convergent within 2000 epochs

CPG has to be taken into account. This was done by using the same 'time-convergence' measure. Each run in the transfer experiments required the training of a new CPG net, so the time to convergence for that CPG was added to that on the variable oscillation task. In the frequency condition, the CPG method was significantly better: average time to convergence of 4328 epochs compared with an average of 6265 for a single network. (t test: $t = 4.761; p < 0.0001; df = 19$). The CPG system for variable phase showed a less significant improvement: the CPG method took an average of 4283 epochs to convergence; a single network took an average of 4440. (t test: $t = 0.253; p > 0.25; df = 19$).

4.8 Using a Genetic Algorithm to build the CPGs

Although it seems obvious to use shaping to build the CPGs, this may not provide systems that have the most benefit on learning. To examine this, a Genetic Algorithm (GA) was used to construct CPGs. The CPGs were used in exactly the same way as in the transfer tasks; the CPG was placed in series with the controller, and the controller and CPG were fully connected with each other. There were two GA experiments, one for the variable frequency, and one for the phase. Each experiment consisted of 5 separate GA runs.

The GA built an initial population of binary strings, which coded for a 6-unit recurrent net. Each member of the population was evaluated, and a new population produced by selecting individuals on the basis of their evaluation, and using the genetic operators of mutation and crossover to build a new generation. The weights and time constants were gray-encoded, with 10 bits allocated to each parameter, and the ranges for the parameters were fairly large - [-16, 16] for the weights, and [0.5,5.5] for the time constants. There is a tradeoff involved in selecting the range of these parameters, as recurrent backpropagation tends to work best with small initial weights, but the weights at the solution of an oscillatory task are often quite large. A 6-unit network has 42 parameters (the weights plus the time constants), so the string contained 420 bits. Each population consisted of 100 networks.

The evaluation consisted of the following operations. The string was de-coded to produce a CPG net. This was then connected to a controller, in exactly the same way as the shaping experiments above. The controllers weights, as well as the interconnections

between controller and CPG, were randomised from the same distribution as above. The combined system was then trained for 800 epochs on the same variable oscillation tasks as before. The MSE on the task, averaged over the final 50 epochs, was taken to be the cost evaluation for the CPG (this is a *cost function*; the GA performs hill-descending rather than hill-climbing). The evaluation function reflects the end of a learning process rather than the whole learning curve; this was to avoid punishing CPGs where the controllers initial weights are poor. 800 epochs is not a very long time to learn the task, but longer learning trials would have made the experiment prohibitively slow. It was hoped that this would be long enough to separate the better CPGs from the others.

The learning rates were all set to be rather high (0.7 for all the weights; 0.2 for the time constants). The reasoning here is to bias the GA into finding structures that produce stable learning, rather than those that only work with small learning rates. The evaluation function for any one CPG will be rather noisy, as it is dependent on the initial weights of the controller. However, GAs have been shown to be effective at optimising noisy, nonlinear functions, so the performance is expected to be reasonable. Also, this is a rather *small* GA for the string length, in the sense that a larger population and more generations would be better. However, the fact that there is learning within the evaluation cycle should help matters; it has been shown that such learning within the evaluation can greatly improve the performance of GAs (Hinton and Nowlan, 1987)

More seriously, as the string codes directly for the values of the weights, a genetic algorithm would be expected to have problems in using crossover effectively. This is because crossover is only an effective operator when the closeness of parameters on the string is related to their degree of coupling, with nearby string elements coding for highly coupled parameters. In the present case, the problem is that the 'goodness' of the value of a particular weight is dependent on the values of the other weights, so the degree of coupling does not fall off with separation on the genetic string. There are other problems too, such as the fact that all the units in a net can be re-labelled, and the behaviour of the net will be unchanged, but the genetic representation will be radically different. This is called the 'permutation problem'. Despite all the theoretical problems, crossover has been found to be an effective operator in these problems (Beer and Gallagher, 1992), and

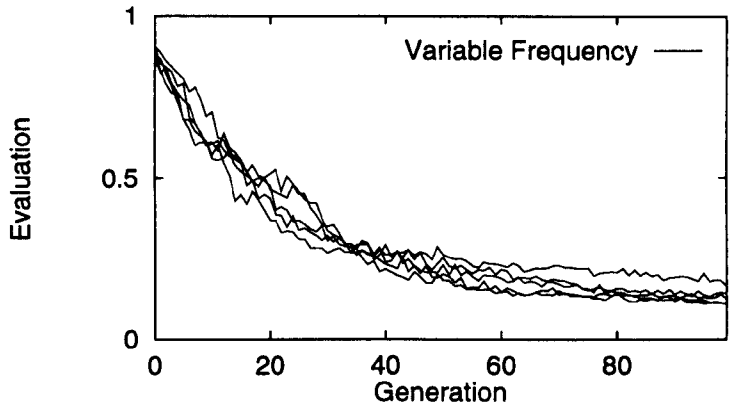

FIGURE 4.15. Mean evaluation of the whole population on the variable frequency task. Plot shows 5 curves, one for each of the separate GA runs

a reasonable crossover rate of 0.6 per string was used in these experiments.

4.8.1 Results

Figure 4.16 shows the average evaluation of the whole population on the variable frequency task, for each of the five runs. Figure 4.15 shows the same performance measure for the variable phase task. In both the variable frequency and the variable phase experiments, the GA converged within 100 generations, suggesting that the reasoning about population sizes was correct.

The goal of the GA is to build CPGs which are optimal, in the sense of helping the learning for a randomly initialised controller. The CPGs are dynamic systems in their own right, which may be investigated by observing their behaviour without any external connections from the controller. The dynamics of the evolved CPGs are broadly divided between oscillators, and damped oscillators. A damped oscillator is a system which is a 'near-oscillator'; it shows oscillations which decay over time, and the system eventually converges to a fixed point. The dynamics of one such CPG are shown in figure 4.17.

The CPGs which are stable oscillators are evolved rather quickly - usually within 30-40 generations. Evolution of oscillators typically takes 100-200 generations for nets of this size, so the presence of learning with the evaluation cycle is speeding evolution. This is an example of the Baldwin effect (Baldwin, 1896); learning in the evaluation informs the GA

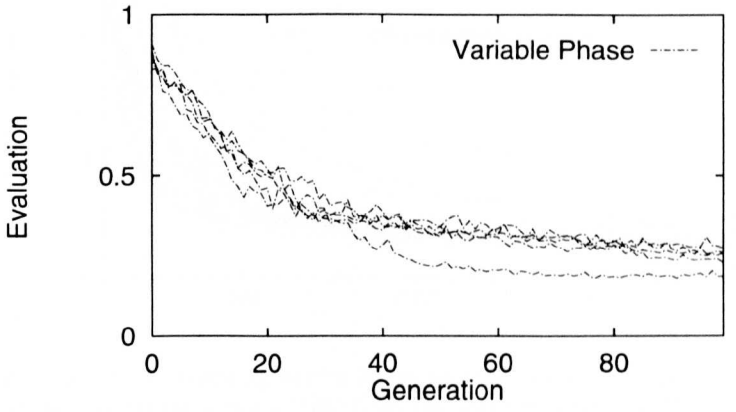

FIGURE 4.16. Mean evaluation of the whole population on the variable phase task. Plot shows 5 curves, one for each of the separate GA runs

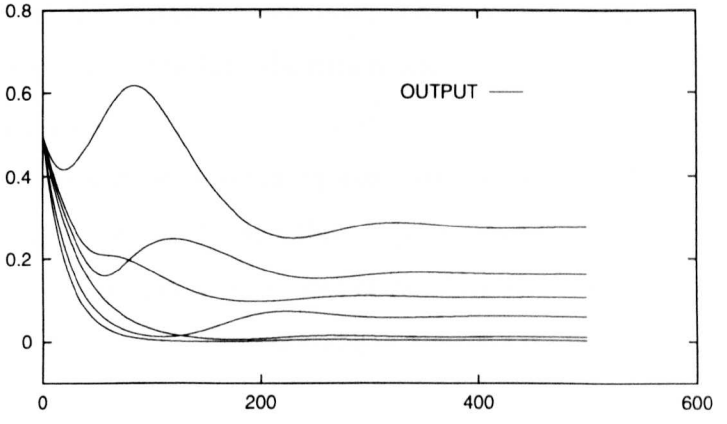

FIGURE 4.17. Output of one evolved CPG. This behaviour is generated by the CPG in isolation, i.e. all the connections between the CPG and the controller have been removed. The y-axis shows the activity of each of the CPGs units; the x-axis is time. The dynamics of the CPG are that of a damped oscillator; the activity spirals in to a fixed point

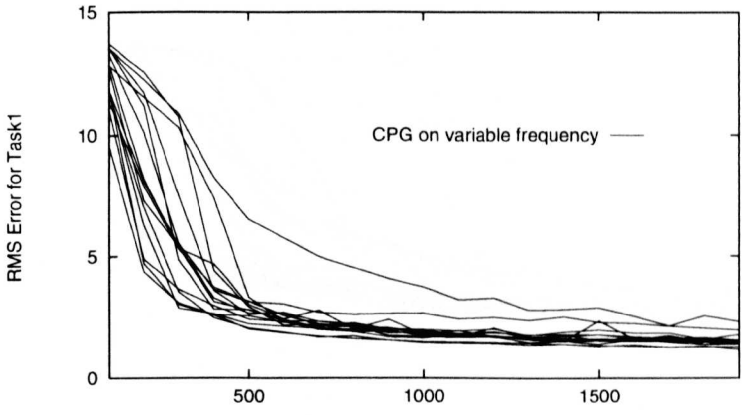

FIGURE 4.18. Learning curves for 20 controllers on the variable frequency condition, using a CPG from the variable frequency GA. The GA has found CPGs which perform better than those from shaping

about a *region* in weight space, rather than just the point that corresponds to the initial weights.

The learning curves in figure 4.18 were obtained by selecting 20 random CPGs from the variable frequency GA, and training a controller with those CPGs on the variable frequency task. Figure 4.19 shows the same procedure for 20 CPGs from the variable phase GA, trained on the variable phase task. The learning rates and distributions for the initial weights were as in the transfer experiments.

Statistical Analysis

The same time-to-convergence measure was used to compare these CPGs with those from the transfer experiments. Unlike the transfer experiments, there is no easy way to take into account the computational cost of building the CPGs with the GA, so this comparison is purely concerned with the training time for the controllers, and not for the effort required to build the CPGs.

The CPGs from the GA were significantly better than those from the transfer experiment in both cases. In the frequency condition, the CPGs from the GA took an average of 440 epochs to converge; those from shaping took an average of 795. (t test: $t = 8.557; p < 0.0001; df = 19$). In the phase condition, the evolved CPGs took an average of 693 epochs; those from shaping took an average of 775 epochs. (t test:

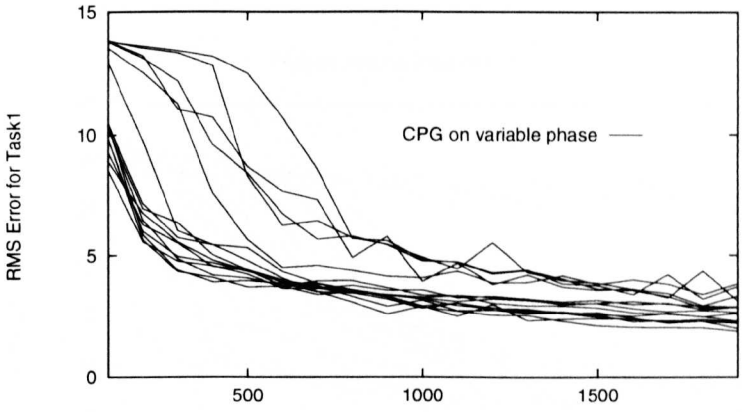

FIGURE 4.19. Learning curves for 20 controllers on the variable frequency condition, using a CPG from the variable phase GA. The GA has found CPGs which perform better than those from shaping

$t = 2.461; p < 0.05; df = 19$). The entire set of evolved CPGs, for both the frequency and the phase GA, was then divided into stable and unstable oscillators. A comparison between these two types of dynamics was performed by selecting 20 stable oscillators (10 from the frequency GA, 10 from the phase), and 20 unstable oscillators. Each CPG was evaluated on its corresponding task, and the difference between the two groups (unstable vs. stable) measured. The results are that the unstable CPGs showed a significantly better performance than the stable oscillators, when averaged over the two tasks. (mean of 494.5 vs 591.8: t test: $t = 2.214; p < 0.05; df = 19$).

One consequence of using such a high learning rate in the GA is that CPGs which give stable learning are selected over others. This improved stability is demonstrated by training controllers on the same tasks, but using a longer time window. The variable frequency and variable phase task were used as above, but the time window for the simulations was increased from 3 s to 6 s.

Figure 4.20 shows learning curves for 20 controllers on the variable frequency task, using a CPG from the GA, and using the increased time window. Figure 4.21 shows the same for the phase task.

Increasing the time window has had a detrimental effect on learning, with two controllers in both tasks failing to learn at all. However, the performance of controllers

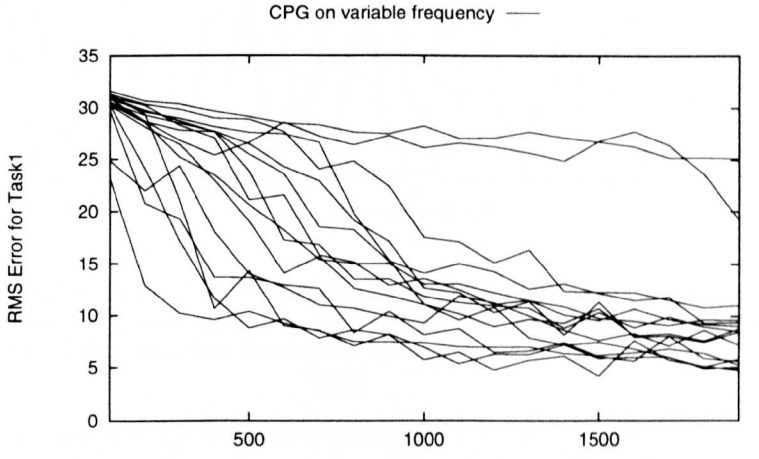

FIGURE 4.20. Learning curves for 20 controllers on the variable frequency condition, using a CPG from the variable frequency GA. Learning is performed on a longer time window of 6 secs, which is seen to cause learning performance to deteriorate.

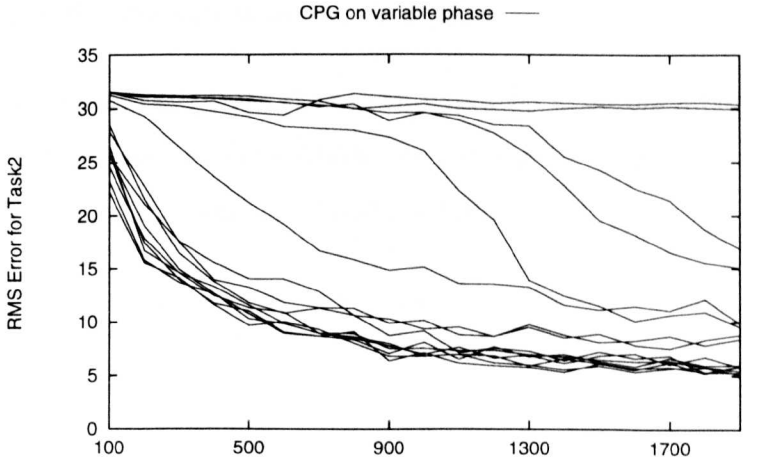

FIGURE 4.21. Learning curves for 20 controllers on the variable phase condition, using a CPG from the variable phase GA. Learning is performed on a longer time window of 6 secs. As with the frequency condition, learning performance has deteriorated by using a larger time window, but the performance of the learned controllers is improved

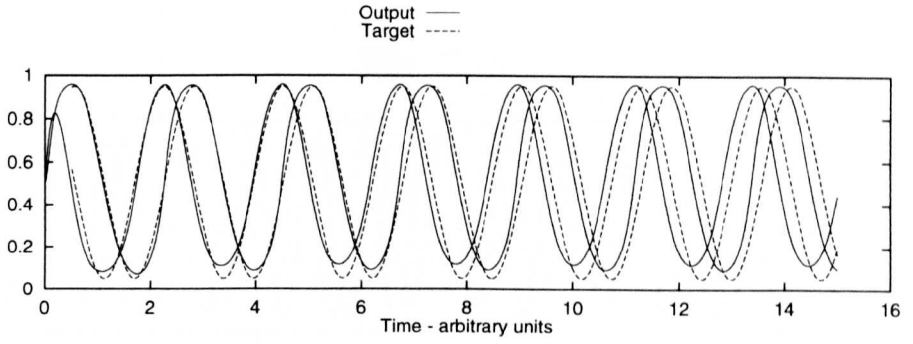

FIGURE 4.22. Performance of a controller trained with a larger time window. Target trajectory is the dotted line; actual output the solid line. Plot shows **low frequency** output

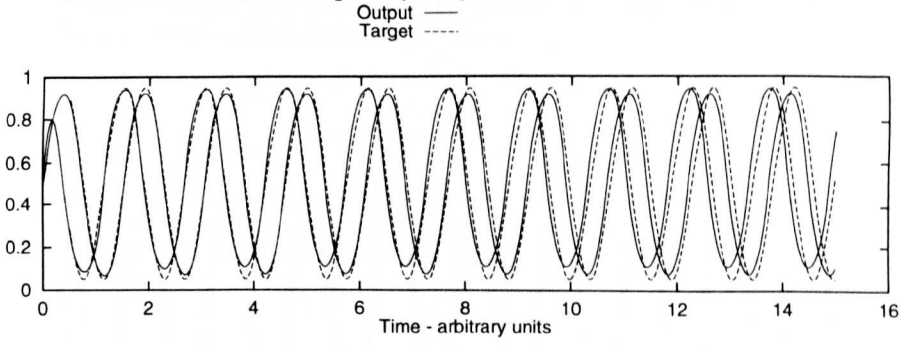

FIGURE 4.23. Performance of a controller trained with a larger time window. Target trajectory is the dotted line; actual output the solid line. Plot shows **medium frequency** output

which successfully learned the tasks showed an improved fit to the targets. Figures 4.22 - 4.24 show output from one controller which was successful in learning the frequency task. Figures 4.25 - 4.27 show the same for the phase task.

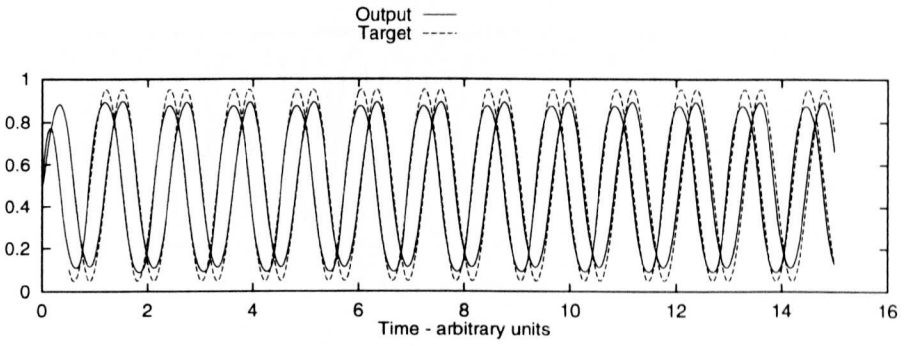

FIGURE 4.24. Performance of a controller trained with a larger time window. Target trajectory is the dotted line; actual output the solid line. Plot shows **high frequency** output

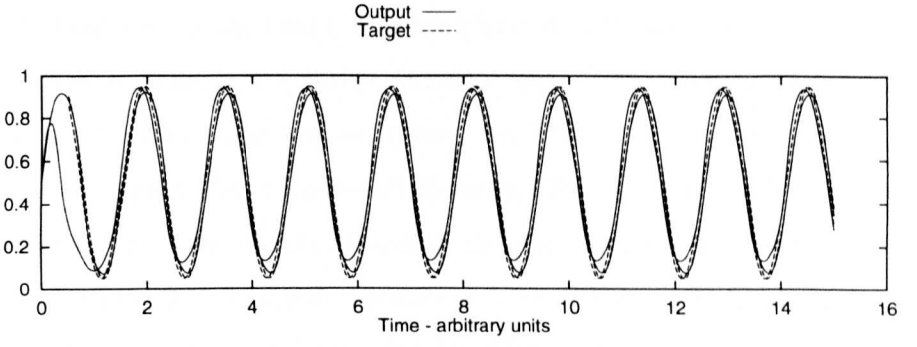

FIGURE 4.25. Performance of a controller trained with a larger time window. Target trajectory is the dotted line; actual output the solid line. Plot shows **small relative phase output**

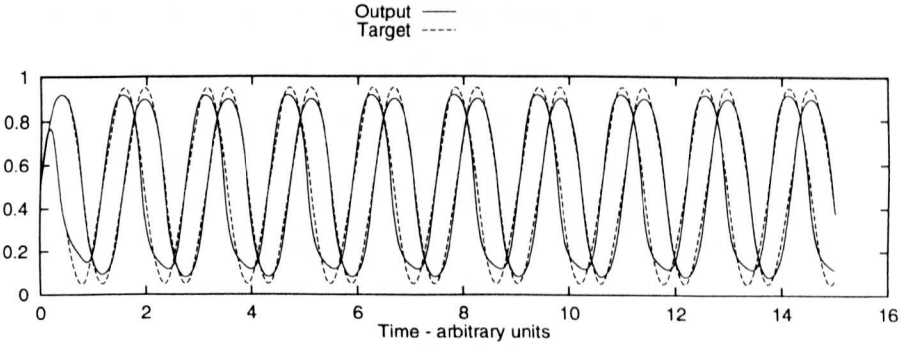

FIGURE 4.26. Performance of a controller trained with a larger time window. Target trajectory is the dotted line; actual output the solid line. Plot shows a **medium relative phase output**

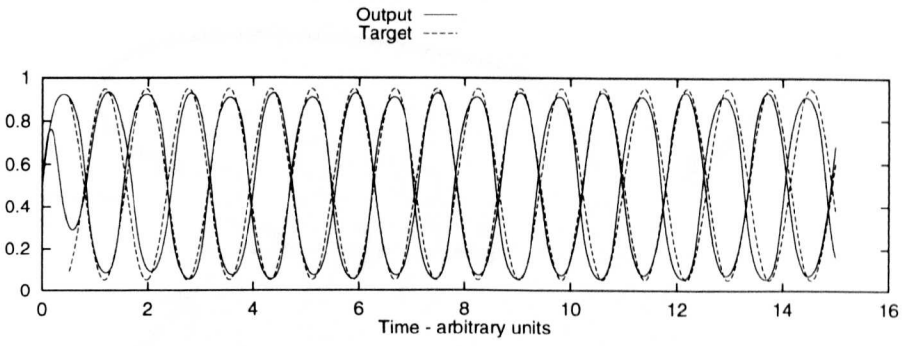

FIGURE 4.27. Performance of a controller trained with a larger time window. Target trajectory is the dotted line; actual output the solid line. Plot shows a **large relative phase output**

4.8.2 Discussion

It has already been mentioned that CPGs with unstable dynamics outperform those that are stable oscillators. This fact is rather puzzling: part of the motivation for using *stable* oscillators in the transfer tasks was to remove the likelihood of bifurcations in learning, as bifurcations have been shown to inhibit learning. The optimal CPGs from the GA are unstable systems, which means that learning must go through a bifurcation to produce the target oscillations. This is simply because a system that moves from a point attractor to limit cycle behaviour has to go through a Hopf bifurcation - this refers to an attracting point changing into a repeller, surrounded by a limit cycle attractor. Somehow, these bifurcations must be such that they do not have a detrimental effect on learning - the learning curves clearly show that there are no sharp jumps in error.

There are at least two possible reasons why damped oscillators are optimal for this task. The simplest reason is that the GA is trying to optimise a complex function of the CPGs' weights - the weights not only define the output behaviour of the CPG, they also define what happens to error derivatives when they are backpropagated through the CPG. It is conceivable that the CPGs are optimal in the sense that error derivatives, when backpropagated through the CPG, are most informative for the training of the controller, and that other CPGs would distort these derivatives too much.

The other possible reason has to do with the work of Kelso et al discussed in chapter 2; in that work, pattern generation was seen as stability of dynamics, but pattern change as a loss of stability. As the higher controller has to alter the frequency or phase of the

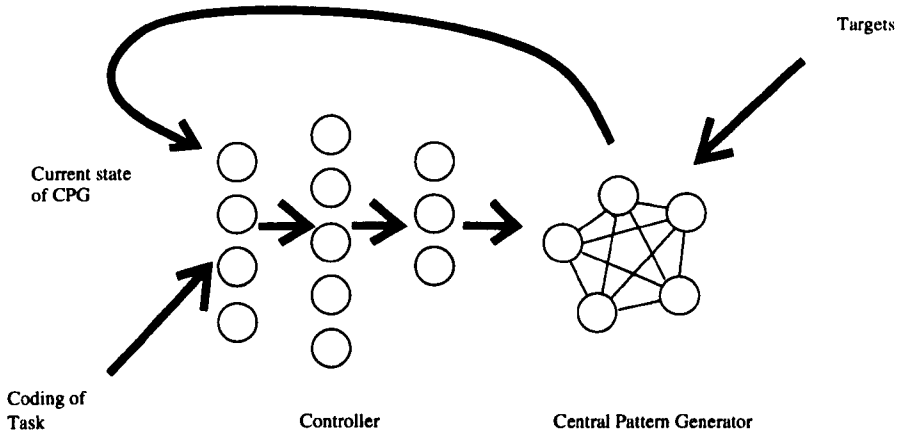

FIGURE 4.28. Using a feedforward net as the controller. As before, the controller receives the entire CPG state as input at each time step, and gives outputs to every unit in the CPG. The controller now has no dynamics of its own; it merely implements a static mapping

CPGs, instability in the CPG's dynamics might make the controller's task easier than a stably oscillating system.

4.9 Variations of the Basic Scheme

The connection scheme used above represents a particular decomposition of the problem - the CPG deals with so much of the problem, the rest is left to the controller. Other decompositions are possible, which make different demands on the two components, and can be easily dealt with within this framework. Consider the case of a simpler controller, implemented as a feedforward net (figure 4.28). At each time step, the feedforward net would take as input the current state of the CPG, together with a coding of the task (say, the required frequency). The output of the controller goes to the CPG.

To see how this affects the calculation of error derivatives, consider the simplest case of a feedforward net consisting of just one unit (Fig4.29). Unit i makes a weighted connection to unit j; unit i is a part of the recurrent net, and has dynamics governed by the recurrent equation, unit j is in the feedforward net, and outputs a static function of its input. The output of unit j makes a weighted contribution to the input to recurrent unit k in just the same way as all the other recurrent units - note that unit k makes no assumptions about the dynamics of the units that contribute to its input. Unit j has no dynamics of its own;

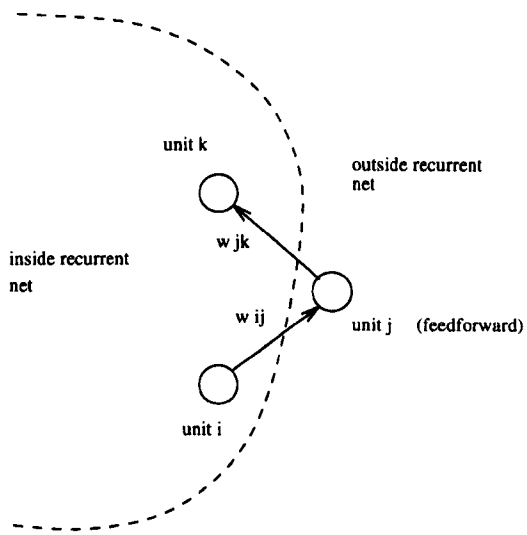

FIGURE 4.29. Connecting a feedforward unit to the CPG. Units i and k are within the CPG (i.e. they are both recurrent units). Unit j takes as input the activation of unit i, and the output of j goes to unit$k$. A form of recurrent backpropagation is used to learn the weights w_{ij} and w_{jk}

its value at each time step is merely a static function of unit i's: $x_j(t) = \sigma(w_{ij}x_i(t))$

As far as unit i is concerned, the feedforward unit provides an additional way in which its activity will influence unit x_k, which means that the equations for the error derivatives have to be augmented:

$$\begin{aligned} \dot{z}_i &= \frac{1}{T_i} z_i - e_i - \sum_l w_{li}\, \sigma'(y_l)\, z_l \\ &\quad - \frac{1}{T_k}\, w_{ij}\, \sigma'(y_j)\, w_{jk}\, \sigma'(y_k)\, z_k \end{aligned} \tag{4.8}$$

The final term takes into account the pathway through the feedforward net. This term can be decomposed, so that error derivatives are given for unit j itself; this is necessary when unit j is replaced with a net with hidden units. The error derivative for unit j is given by:

$$z_j(t) = \frac{1}{T_k}\, w_{jk}\, \sigma'(y_k)\, z_k$$

which means that equation 4.8 can be re-written as

$$\dot{z}_i = \frac{1}{T_i} z_i - e_i - \sum_l w_{li}\, \sigma'(y_l)\, z_l - w_{ij}\, \sigma'(x_j)\, z_j \qquad (4.9)$$

Given that we now have the error derivatives for all the unit's activations, the derivatives for the weights are calculated in exactly the same way as before. The extension to a feedforward net with hidden layers is straightforward. Finally, note that this situation is identical to the use of a feedback controller with a nonlinear plant, where the controller is implemented as a nonlinear, static function of its inputs. The only difference is that the recurrent net is not a model of the plant; rather it *is* the system to be controlled.

4.9.1 Learning with a Static Controller

In this experiment, a simple feedforward net was used as the controller for the variable frequency task. The CPG was the same CPG as that used in the transfer experiments with the recurrent controllers; it is a recurrent network trained on a single oscillation. The net's input at time t consisted of the entire state of the CPG, and the output went to all the units in the CPG. One additional input was given, specifying the desired behaviour. Logistic units with outputs in [0,1] were used, except in the output layer, in which a [-1,+1] sigmoid was used. This was so that the output units could drive the CPG activity down as well as up. Before each learning trial, the feedforward weights were randomised from a normal distribution with mean 0.0 and s.d. of 1.0. A learning rate of 0.1 and momentum of 0.9 was used throughout.

4.9.2 Results

The learning curves for 20 controllers are shown in Fig4.30. The performance on this task is relatively poor, with some networks not converging at all, and the convergent ones having a high residual error. Examination of the networks that failed altogether shows that they were initialised with sufficiently high weights that their outputs destroyed the oscillations of the CPG. In these cases, the learning seems to be unable to reduce the magnitude of the weights in order to restore the oscillations.

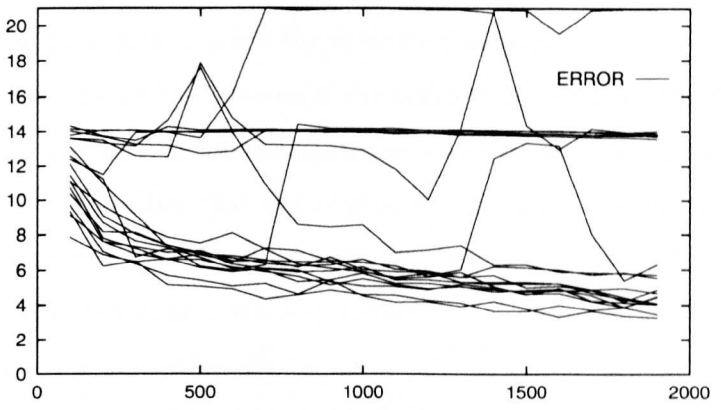

FIGURE 4.30. Learning curves for 20 feedforward nets on the variable frequency task. The learning performance is relatively poor, with many controllers failing to learn at all. Examination of controllers which completely failed to learn reveals that they gave sufficiently large outputs to destroy the oscillations of the CPG altogether, and learning is unable to reduce the controllers outputs so as to restore the oscillations

To examine the effects of disrupting the CPGss behaviour, a 'toy' problem was considered, in which the controller received just one input coding for the task, and produced one output. For the CPG, we now used the combined CPG-controller system from the previous experiment. This combined system is a network which has successfully learned the variable frequency task. The CPG now has one input, which codes for the desired frequency, and two outputs, which oscillate at the target frequency. As this is a system which has *learned* the variable frequency task, setting the value of the one input unit controls the frequency of oscillations at the outputs, so learning to control this system is simply the problem of setting the value of the input. The weights of this variable frequency system were frozen, and the entire system used as the CPG for the current experiment.

The controller for this experiment is very simple; there is one input corresponding to the desired frequency, and one output that goes directly to the CPG's input unit. The task for the controller is extremely simple; it merely has to copy its input to its output. The only complication is that the outputs are fed through the CPG, and the error backpropagated through the CPG, which may distort their values. The controller was initialised with weights from a Gaussian distribution with zero mean and unit s.d.

Again, some of the networks failed to learn the task, despite the fact that the controller's task in this toy problem is just to learn the identity function.

The reason for the failures can be seen if the training information for the controller is examined. The error derivatives for the controller's output unit are given directly from the CPG's input unit (remember that the z value for the controller output is identical at each time step to that for the CPG's input unit, save for a constant scaling factor given by the recurrent unit's time constant). We may reconstruct a target value for the controller's output unit (as the error derivative $\frac{\partial E}{\partial y_i} = x - x^*$; so $x^* = \frac{\partial E}{\partial x_i} - x$). Figure 4.31 shows the targets for a controller which failed to learn the task. There are three plots here, corresponding to a low, medium and high frequency target. The 'actual' targets - the values which cause the CPG to produce the correct oscillation - are shown as dotted lines at three constant values. The target values are not expected to lie precisely on their correct values, as they are the result of a integrating a complex set of differential equations. Nevertheless, the targets have to be informative, in the sense of causing learning to proceed in the correct direction.

As can be seen, the targets are not only distant from their optimal values; they are also going to lead learning in the wrong direction. Even the ordering among the three conditions has been reversed. Recall that these targets are the result of solving a set of differential equations *backwards* in time, starting at the end of the time window. It seems as though the early values, around $t = 20$ are quite reasonable (they roughly preserve the ordering between the conditions), but they become more and more distorted with time, and are completely uninformative for about half of the time window. The cause of this behaviour is that the output of the controller with randomly initialised weights was sufficiently large to destroy the oscillations. In this situation, the error derivatives are no longer informative, and the oscillations cannot be recovered, despite the fact that this only requires the controller output to be made smaller.

By contrast, consider the reconstructed target values when the controller is initialised with smaller weights (Fig4.32). These controllers gave small inputs to the CPG, and so the oscillatory activity of the CPG was left intact. The target values are roughly aligned with the 'correct' input value - the input value that produces the correct oscillations.

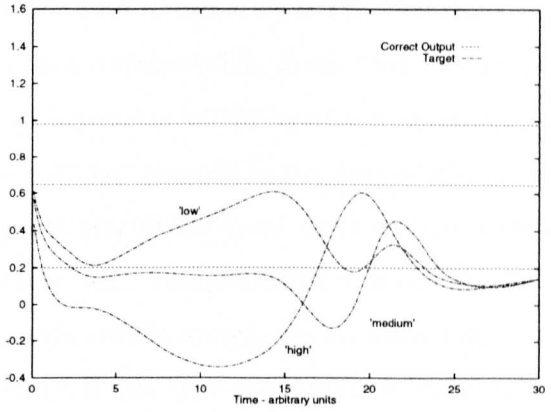

FIGURE 4.31. Reconstructed target values for a controller which failed to learn the variable frequency task. The y-axis shows the values of the targets; the x-axis is time. The target values are given by the error derivatives, which are in turn given by the solution of a set of differential equations, *backwards* in time. There are three plots, corresponding to three different patterns - **low frequency**, **medium frequency** and **high frequency**. At around t= 20 the targets are seen to be informative - the target for the **high frequency** pattern is the highest, followed by that for the **medium frequency**, and then the target for the **low frequency** pattern. These targets are informative because the ordering among the patterns is correct. However, the targets become more and more distorted as they evolve in time, and the final values are completely uninformative

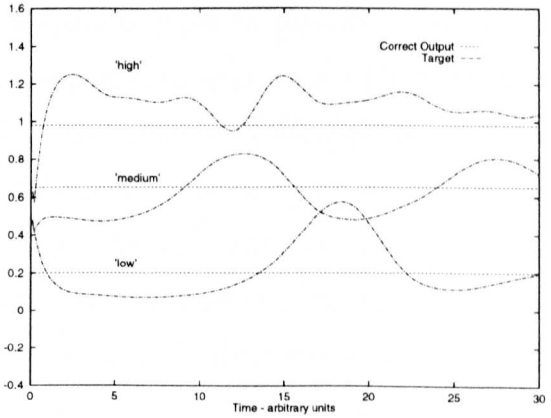

FIGURE 4.32. Reconstructed target values controller initialised with small weights. The targets values are seen to remain roughly aligned with their correct values - the values which produce the correct output from the CPG

The conclusion to be drawn here is that the error derivatives given by recurrent backpropagation are only likely to be informative if the behaviour of the net is close to the target behaviour. This is not unreasonable, given that backpropagation essentially considers the first term in an expansion of the system's equations. If the higher terms are significant, then learning may not proceed in the desired direction.

This highlights one particular difficulty of using backpropagation to estimate derivatives with complex recurrent nets: backpropagation is not an effective approach to credit assignment when the dynamics of the system are far away from the target dynamics. This has two implications. Firstly, if this type of supervised learning is to be used with CPGs, the dynamics of the CPGs may have to be constrained in some way to ensure that error derivatives, as they are backpropagated through the CPG, still point in the correct direction. This seems like a hard task, although we would expect weakly coupled systems to perform better than those that are strongly coupled. CPGs in which the activity is more localised, such as the motor maps seen in cortex, would be expected to perform better than the systems of coupled oscillators seen in spinal circuits. Note that even in the case of oscillatory spinal circuits, it is still not clear whether there are sets of essentially independent oscillatory systems, or whether the whole system is strongly coupled; this could make a large difference to the learning of complex motor acts that recruit these circuits.

Secondly, as this learning situation is identical in structure to learning with a distal teacher, we might expect distal learning to perform poorly when the plant (and so the plant model) shows very complex behaviour, even if the plant model is perfectly accurate. Finally, learning control in the presence of complex dynamic systems might sometimes be better approached with direct methods, which do not rely on gradient information. A good example of this would be the systems of coupled oscillators that are found in invertebrate motor circuits. We would not expect gradient descent methods to work well with such multistable systems, and yet there are often simple relations between external inputs and, say, frequency of oscillation. This is investigated in chapter 6.

Bearing this in mind, a feedforward net was used as a controller for a the original variable frequency task. The initial weights were initialised from a distribution with a smaller s.d of 0.2, and the learning is now seen to converge reliably to a good solution.

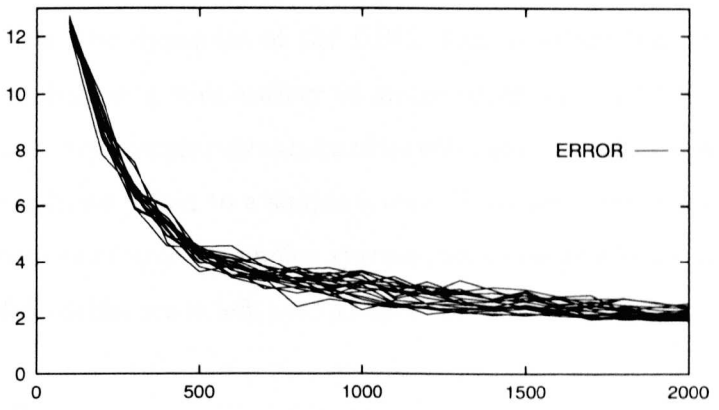

FIGURE 4.33. Learning curves for 20 feedforward networks on the variable frequency task. The initial weights of the controller were drawn from a distribution with s.d 0.2; these initial values were sufficiently small to leave the CPG oscillations intact

4.10 Using Biological Pattern Generators

One of the advantages of using CPGs instead of say, incremental learning, is that the CPG may be built with a variety of methods, not necessarily the current learning task. This includes any knowledge we may have about the structure of biological control systems; CPGs provide a way of including that structure in a control scheme. A good example of this arises in the control of legged robots - controllers for such systems have been widely based on pattern generating systems in animals. Such artificial CPGs might be the result of modelling of fictive locomotion in a specific animal; alternatively, it might represent quite general knowledge about the dynamics of biological systems. In any case, the CPG is treated as a functional unit that may be added to a control scheme.

There is widespread support for the idea that the gaits, and gait transitions seen in quadrupeds are produced by a set of coupled oscillators (Schoner and Kelso, 1988b; Yuasa and Ito, 1990; Collins and Richmond, 1994). There is much work yet to be done in understanding these systems: for example, it is still not known whether the couplings between oscillators change in switching between gaits, or whether the gait changes arise just from the network dynamics. Nevertheless, the broad structure is well-accepted. Such a system might be used as part of a controller, say for a legged robot. So far in this

chapter, distal learning has been used to account for the effects of a CPG, without making any assumptions about the dynamics of the CPG. This is attractive, as it provides an integrated way to implement a wide variety of motor programs. A CPG that consists of coupled oscillators has very complex dynamics of its own, and the role of descending signals from a controller is reduced, often to a simple scalar. However, using backpropagation to account for the dynamics of such a complex system can cause problems, and the problem of controlling coupled oscillators is left until chapter 6.

4.11 Discussion

The goal of this chapter was to investigate the use of recurrent networks for pattern generation tasks. A set of simple oscillatory tasks was used as a testbed, where the target outputs were defined by the frequency or relative phases of two sinusoids. All the networks studied were trained with supervised learning, using a variant of recurrent backpropagation.

In section 4.4, networks were trained to produce a single oscillation. Learning of oscillatory behaviour shows some degree of instability, and it was shown that the use of the Delta-Bar-Delta rule improves learning, by adaptively setting learning rates for each weight and time constant in the network.

Section 4.5 considered the training of networks on two separate tasks: producing variable frequency, or variable phase oscillations. The networks used were slightly larger (9 units), and the training times rather longer. Section 4.6 introduced the idea of a Central Pattern Generator in training recurrent networks, which corresponds to imposing a hierarchical structure on the learning system. In section 4.7, a structural shaping approach was used, in which CPGs were built by training smaller networks on simpler versions of the pattern generation task. It was shown that the subsequent training of the rest of the system was greatly improved, and that the stability of the learning process had been increased.

Section 4.8 took a different approach, and used a Genetic Algorithm to build CPGs. It was shown that the GA is able to find CPGs which outperform those from the shaping experiments, and that the optimal CPGs tend to show unstable dynamics. Finally, section

4.9 considered the use of different types of network for the controller and the CPG, and it was shown that it is straightforward to train feedforward nets as controllers, with the CPG still implemented as a recurrent network. It was also shown that there are potential problems in backpropagating through CPGs with complex dynamics, in that error derivatives may become uninformative when the behaviour is far away from the target trajectory.

The optimal CPGs from the GA experiments showed unstable dynamics, implying that learning variable oscillations still required the combined controller-CPG system to go through a bifurcation. The fact that these bifurcations do not disrupt learning is surprising, and deserves further attention. A more detailed study of the changing system dynamics as learning occurs would seem to be an obvious direction for further research.

The controllers used in section 4.9 were implemented as static mappings, which is a very straightforward variation of the recurrent backpropagation procedure. However, this does show how it is possible to build systems in which there is a variable division of labour between the fixed components and the learning controller.

The idea of using CPGs in training networks was inspired by the hierarchical designs seen in biological control systems. The main conclusions from this chapter are that such hierarchies may be seen as a kind of task decomposition, and that the presence of a CPG with the appropriate dynamics can greatly ease the task of a higher level controller. This kind of modular architecture is most obviously related to another approach in learning control: distal learning (Jordan and Rumelhart, 1992; Jordan, 1988). In distal learning, the plant model is a fixed (or slowly changing) component in the control system. The role of the plant model is to transform error derivatives into the space of the controllers output, but the model does not change the way in which the control actions affect the plant. Distal learning may be seen as a way of accounting for the presence of the plant, as long as the model is a sufficiently accurate approximation to the plant. Jordan (1988) also shows how distal learning may be used to solve ill-posed control tasks, by finding the control inputs which maximise satisfy some constraint function.

This chapter has taken a different approach, and considered distal learning to be a tool for accounting for the presence of other components in the control system, in particular, components which are added in order to solve part of the control task. This provides a

natural way to impose structure on a controller, breaking it up into subparts, and providing each subpart with meaningful training information.

This is also related to incremental methods in training recurrent networks (Giles et al., 1992; Elman, 1993), where networks are also trained on a series of tasks, increasing in complexity. The main difference with that work is that the CPGs used here are fixed during later learning. The effect of having a fixed CPG rather than a single network is hard to assess, as it would be difficult to ensure that both methods had the same number of variable parameters. However, the CPG method does allow the experimenter to represent different types of prior knowledge - for example, it could represent a model of an invertebrate motor circuit.

In the next chapter, these ideas will be investigated in a control setting, involving the use of a simulated robotic arm.

5 Learning Control with Motor Programs

Recurrent neural networks offer a powerful way of constructing complex controllers for motor control and robotics tasks. This chapter considers the use of recurrent backpropagation for training controllers for a series of motor control tasks, involving the use of a simulated redundant robot arm.

Firstly, the use of motor programs in robotics problems is discussed, and recurrent networks are introduced as a very general method for implementing motor programs. Section 5.4 introduces a set of simple motor control experiments, involving the robot arm making a series of reaching movements. CPGs are added to the control scheme; the CPGs are built by incremental learning, and from a Genetic Algorithm. The presence of the CPGs is shown to improve learning.

In section 5.9, a robot with two arms is introduced, and it is shown that modular control architectures give a way of de-coupling parts of the control problem. Finally, section 5.12 considers the use of a feedback controller for a robot arm with dynamics.

5.1 Introduction

This chapter is concerned with the use of recurrent neural networks for robot control and skill learning tasks. There are several reasons why neural networks may be attractive for these types of problems. Neural networks are powerful learning systems, capable of approximating arbitrary functions, and so capable of implementing arbitrary dynamic systems. In contrast to linear control, nonlinear control is really still in its infancy, and much of the rigorous framework of linear systems theory has to be abandoned in the full nonlinear case. It seems likely that neural networks will find widespread use in the

developing field of nonlinear control. The main attractive features of neural networks are their ability to generalise from sparse data, and to capture complex relations between variables. In the case of recurrent nets, they offer a powerful way of synthesising complex control systems. Recurrent nets have been investigated as a way of implementing complex controllers by a number of researchers (Meeden, 1984; Kumaresan and Sharkey, 1993).

Another, quite different source of interest in neural networks lies in their apparent relation to biological neural systems. In robotics, as in vision, biological systems are always of interest because they constitute the only existence proof that the really hard problems can be solved at all. This interaction works in both directions - researchers in biological motor control have much to learn from robotics, and it seems likely that many of the concepts in robotics will find a place in a computational theory of motor control tasks. Neural networks offer a way of tightening the parallels between these two fields, sometimes at the level of modelling the fine detail of motor circuits, and sometimes by implementing more abstract models.

This chapter will be concerned with the use of motor programs - a concept that has found use both in robotics and the experimental literature. As in the previous chapter, these motor programs are implemented as recurrent nets, with the machinery of distal learning used to perform credit assignment. This approach gives the experimenter a great deal of freedom in the design of the control system, as the motor programs are complex dynamic systems in their own right, and have the potential to solve difficult subparts of the control problem. Different motor programs may serve different purposes, and this chapter considers the use of motor programs to represent three different subparts of a control task - pattern generation, redundancy reduction, and feedback control.

These ideas are developed in the context of controlling a simulated robotic arm. All of the architectures considered here require the use of a plant model, so this is essentially supervised learning. In the following chapter, the ideas will be developed in a direct control setting, using both reinforcement and supervised learning methods.

5.2 Robotic Control

As an archetypal robotics task, consider the pick and place problem. This usually involves a robot arm, and an object, and the goal is to cause the arm to reach towards the object, and pick it up. A huge amount of research effort has been spent on this problem, mostly due to its ubiquity in industrial applications. A significant part of this problem is the reaching task - to take some coding of the target position for the robot, and its initial position, and generate a series of control actions to move the arm to the target. The task is usually ill-defined, as the target information only applies to the endpoint of the arm (the gripper), and there are usually nonlinearities that arise from the couplings between the dynamics of the individual links, and fictitious torques arising from the rotating coordinate systems. These factors make it difficult to design a controller using traditional methods from linear systems theory.

The other main source of difficulty is planning a path from the initial to the goal state. This is usually dealt with by a separate path planning module, which uses geometric reasoning about the kinematics of the plant, and the configuration of any objects, to generate a path that is optimal is some sense (robots often have to work in quite cramped conditions, and it is usually desirable to minimise energy expenditure and wear on the robot). The demands of the task often underspecify the trajectory, so it is common to use regularisers, or some other method for introducing constraints. These constraints may have a utility of their own, such as trying to minimise energy spent, or they may be rather arbitrary, and just serve to reduce the effective number of degrees of freedom. It is common for robots to have excess degrees of freedom (e.g. to have 4 joints, when the endpoint is a point in 3 dimensional space); such systems are called *redundant* robots. The difficulty of path planning is dependent on two conflicting effects. On the one hand, the complexity of path planning scales very badly with the number of degrees of freedom in the plant, as well as the number of objects. This is simply because a search process is necessarily involved, and would lead us to favour plants that have a low level of redundancy. On the other hand, many difficult path planning problems, such as reaching in cluttered environments, are often made trivial by allowing the robot to move in more ways. These considerations might lead us to suggest the use of more flexible ways of reducing the degrees of freedom.

Again, this is the kind of behaviour that is seen in biological systems: animals are able to make flexible and intelligent use of their many degrees of freedom.

Having found a path to the goal, the problem remains of actually finding the controls to follow the path. The most common approach is to use a plant model, which, when sufficiently accurate will map the relation between controls and states. This model will have a component to deal with the kinematics of the robot (the mapping from joint angles to positions), and a component for the dynamics. A forward model maps from controls to states; an inverse model maps (desired) states onto the controls that achieve those states. In either case, the plant model is used to find the time series of controls that achieve the desired reaching to the goal.

The combination of these processes makes the whole procedure rather computationally demanding, even in simple cases. Implicit in this scheme is a kind of task decomposition, in which the problem of control (in the sense of driving a system to a setpoint) is considered as separate to the problem of planning. However, some recent work on the use of motor programs has questioned whether this is the best way to divide up the control task.

5.2.1 Motor Programs in Robotics and Motor Control

The concept of a motor program has changed radically since its inception in the 1950's. Originally conceived as something resembling a computer program, motor programs have taken on board much of the terminology and concepts of cognitive science over the years, and so reflect the changes that have occurred in the field. These changes have prompted an intense debate over the nature of representations involved, with some researchers taking the extreme position that the whole idea of a motor program is now so contentious that the concept should be abandoned (Alexander et al., 1992, and peer commentary). Nevertheless, there is a general consensus of opinion that motor programs, in one form or another, will find a place in a complete account of motor behaviour.

In robotics, the concept of a *motor program* is increasingly receiving attention. This is partly because motor programs seem to be such an important part of biological motor systems, and biological systems show exactly the sort of flexibility and intelligence that is desired in modern robots. A lot of recent research in robotics borrows heavily from

our understanding of biological motor systems, from the use of actuators that behave in a similar way to muscles, to the use of behaviour-based strategies.

Also, the use of motor programs corresponds to a type of task decomposition, and is related to incremental methods for training a controller. Such methods are likely to become essential as robotics tasks increase in complexity. Finally, the introduction of motor programs often means that the complexity of the adaptive part of the control system may be reduced - for example, a linear controller may produce quite complex trajectories if the motor programs show complex dynamics.

The work of Lipitkas *etal* (Lipitkas et al., 1992; Lipitkas et al., 1993; Bock et al., 1993; Bock et al., 1996) is concerned with the representation of motor programs for reaching movements with a real robot. In that work, the motor programs are represented as a waveform generator that produces a prototypical time function for the torques supplied to the joints. These time functions are modelled as a combination of sine waves of length 2π, i.e. consisting of one positive lobe and one negative lobe. This function has six parameters that smoothly change its shape. The controller takes as input the specification of the task (the initial configuration in joint space, and the target in Cartesian space), and supplies as output the parameters to the torque generators. The controller implements a static mapping from task specification to torque parameters, once for every reaching movement. The torque generator takes the controller's output and generates the entire reaching movement in an open-loop fashion. The attraction of this method is that a dynamic problem has been converted into a static one, which greatly simplifies the role of the controller. The price (and attraction) of using such a simple system is that the control system is restricted to producing a small set of possible behaviours, and it is not clear how to extend this to more complex tasks - for example, there is no way to include feedback in this scheme.

The waveform generators themselves have to be built, and this process is rather complicated, involving the use of an operator to generate smooth reaching movements, and an inverse dynamics model to approximate the real torque profiles. These torque profiles are then approximated by sine waves, so that they correspond to the output of the torque generator. Feeding *this* time series into the arm, and observing the final position, gives

a way to build a training set for the controller. This is now a version of *direct inverse* learning.

Although building the motor programs is rather involved, the advantage of this system is that the optimisation of the controller is very simple, both in the sense that this is a static learning problem, and also in the sense that the credit assignment problem for the controller is of a simple form. This is because the relation between the controller outputs and the final endpoint position is a simple, smooth mapping. Other unpublished work by the same authors has shown that reinforcement learning is also effective at training the controller.

The studies of Berthier *etal* (Berthier et al., 1992; Berthier et al., 1993; Houk et al., 1990) are also concerned with the generation of reaching movements, but the control systems studied are more inspired by known physiology than considerations from adaptive control. In that model, reaching movements are produced by the activity across a set of Pattern Generators, corresponding to known circuits in cerebellar cortex. All the pattern generators contribute to the movement to some degree, so the motor program is distributed across a set of pattern generating circuits. The main result of this work is to show how gradient-descent type learning may be implemented with some rather simple ideas about known connectivity. The motor programs are essentially feedforward, although proprioceptive feedback is used to decide when to extinguish the movement, and is also used in later learning.

Motor programs of a similar type were also used in (Lane et al., 1993). At the lowest level, control was achieved by a set of servoreflexes, which were also used to train an inverse dynamics model with Feedback-Error Learning (Kawato et al., 1987, see section 1.4.2). Given a set of these reflexes, movement is achieved by the distribution of activity over the reflexes. At first, the movement is produced by selecting a weighted sum of the reflexes, where the weights are constant throughout each movement. Reinforcement learning is used to train CPG networks that modulate the values of the weights throughout the course of the movement, so as to minimise various cost functions. The key point is that the presence of the inverse dynamics model transforms the reinforcement problem from a nonlinear to a linear optimisation problem.

One approach to modelling *fast* movements is that of (Gorinevsky, 1993), in which the control system consists of a feedforward controller, muscles and reflex loops. The muscles were an approximation to the dynamics of biological muscles, and there were physiologically plausible delays on the feedback signals, and so the feedback control that arises from the reflexes was too slow to generate movements of the required speed. The feedforward controller is a general function generator, trained by some simple ideas from multivariate approximation theory. The appeal of this scheme is that the controller does not need a plant model to learn the task: it is optimised directly from the observed result of the current movement. The feedforward controller is restricted to producing piecewise constant outputs, and this restriction make the use of search techniques a viable method.

The work of Brooks *et al* (Brooks, 1991) on subsumption architectures takes a slightly different approach. Subsumption is a behaviour-based approach to control. The basic idea is to build individual modules to handle each of the elemental behaviours in a task; the role of the control system is to co-ordinate the activity of the modules. For example, a navigation task could be broken down into simple movement behaviours - walk forward, turn left, turn right. In (Brooks, 1991), these ideas were used to control the locomotion of an artificial insect. The elemental behavioural modules were implemented as finite state machines, with co-ordination achieved by careful control of the timing relations between the modules. Although this is not very biologically plausible, this control system was able to generate tripod and metachronal gaits, depending on the speed of locomotion. The tripod is a very common gait produced by biological hexapods when walking at low speed; the metachronal gait is generally produced at higher speeds. The attraction of such an incremental approach is that more complex behaviours may be built up from simpler behaviours; the motor programs act like debugged systems that can be relied on to perform a simple task. Unfortunately, Brooks has taken a somewhat idiosyncratic approach, which bears little relation to other work in the field, and these ideas have not been tested in more complex problems. These factors make the work rather hard to assess.

However, a key contribution of Brooks' work is that subsumption architectures give a natural way to think about the hierarchical structures used in biological systems. The idea of a system of behaviour-based modules is practically a definition of a pattern generator,

and all the studies mentioned above may be thought of as subsumptive architectures.

5.2.2 Representing Motor Programs as Recurrent Nets

In this chapter, motor programs are represented as fully recurrent neural networks. There were several reasons behind this decision. Firstly, as recurrent nets are very general dynamic systems, this gives a powerful way of constructing motor systems that vary widely in their behaviours. Another way of saying this is that very little designer bias has been introduced - so far. A second reason has to do with the work of Kelso *et al*, where motor systems are seen as specifying the entire dynamic environment of the behaviour, rather than just the stable patterns. As dynamic systems, recurrent nets also give a way of specifying the whole dynamic environment. This means that the way that the system loses stability, and changes patterns, is defined by the motor system, and so is potentially under the control of any learning processes used. This leads into the next point, which is that the motor programs considered here are *learned*, rather than hand-designed.

Finally, this design choice allows the use of similar learning methods throughout the control system, and so this shows how motor programs can be implemented using just the backpropagation procedure. A similar approach was taken in (Jordan, 1988), where it was shown that indirect control may be implemented using nothing more than the existing machinery of backpropagation.

TYPES OF MOTOR PROGRAMS

It is useful to consider individual behaviours as being either temporally discrete or continuous. Continuous behaviours are those that correspond to action that is ongoing; a good example would be locomotion. Discrete behaviours are those that have a definite beginning and end; a good example would be reaching towards a target. Motor systems that produce these behaviours would be very different from each other. A motor program for locomotion would almost always be some kind of oscillator, with the control task framed as specifying the frequency, amplitude and relative timing of the individual oscillations. Motor programs for reaching movements will almost always be fixpoint systems, in the sense that the system will always converge to a stable state under fixed inputs. In this case, the control task is seen as specifying the location and stability of the fixed points,

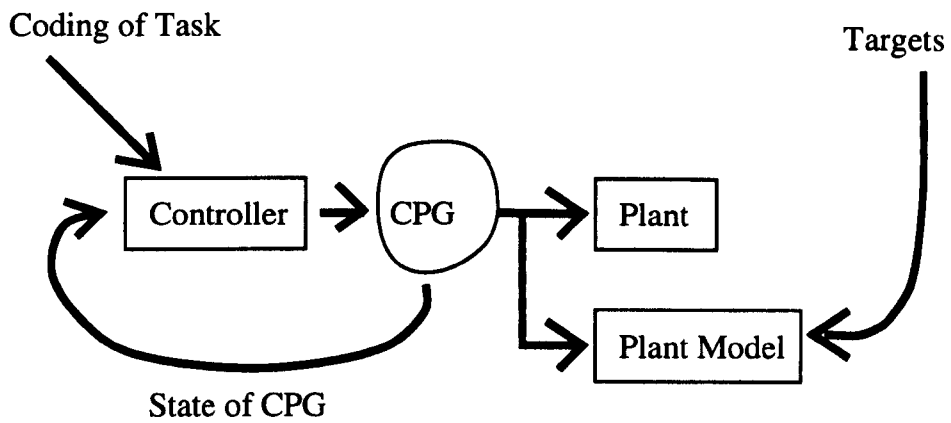

FIGURE 5.1. The full control connection scheme. This is basically the same as the implementation of CPGs from the previous chapter. The only difference is the presence of the plant, and plant model. Targets are now defined at the outputs of the plant, and the plant model is used to transform error derivatives back into the space of the CPGs outputs

and the transient behaviour of the system as it converges to stability.

POSSIBLE CONNECTION SCHEMES

This chapter will take a similar approach to chapter 4, and use a two-stage control system consisting of a motor program, and a higher controller (or equivalently, a CPG and a higher controller). Given this hierarchical design, there is some freedom in the way the components may be connected.

In the simplest case, both the motor program and the higher controller are fully recurrent nets, and there is full connectivity between the two. In effect, there is one single recurrent net, where some of the units are labelled as being in the motor program, and the others as being part of the controller. This is shown in Fig 5.1; we could call this the full-control case.

One obvious alternative is to use a simpler system to implement the controller, say a feedforward net; the backpropagation procedure for this case was discussed in section 4.9. In this case, the controller is still in a feedback loop with the motor program, so a form of recurrent backpropagation is required. A further simplification would be to remove the feedback connections from the motor program to the controller, so the output of the controller is given just by the task specification, and so remains constant throughout each

individual behaviour. The controller output could then be continuously presented to the motor program throughout the behaviour, in which case they act as a constant external force on the dynamics, or they may be briefly applied, in which case they are similar to setting the initial conditions for the motor program. An extreme case would be to have the controller output a non-specific signal that simply turned the motor program on; in that case, the whole behaviour is represented in the motor program, and the role of the controller is reduced to simply deciding when to use that behaviour.

Finally, the motor programs used here have their weights frozen once they have been built, so their place in a backpropagation scheme is purely to provide error derivatives for the controller. This has several implications. As the motor program remains unchanged while the controller is optimised, the behaviour of the control scheme is restricted by the behaviour of the motor program. This can be beneficial, as the learning represented by the motor program cannot be unlearned by the controller. However, this also means that if the motor program is not capable of generating the required range of behaviours, then the controller will not be able to compensate. This issues are very similar to those involved in learning a plant model, and using a fixed model in later learning (i.e. offline learning). In offline learning, the learning is stopped when the model reaches some accuracy criterion, but this accuracy has to be decided. A related issue concerns the range of inputs that the plant model 'sees' during learning; these should be as close as possible to the kinds of inputs the plant will receive in control (if they are sufficiently diverse to reveal all the modes of behaviour of the plant they are said to be 'persistently exciting'). In practice, it is not known what accuracy of plant model is required for a particular control task, and it is not straightforward to choose a training set that is persistently exciting.

These same issues apply to the learning of motor programs. The motor program may not show sufficiently rich behaviour, in which case the performance of the control system will be limited by this one fixed component. This could arise from picking too low a criterion for the training of the motor program, and again, there is no way of knowing in advance what criterion to use. Also, the task used to train the motor program may not be appropriate for the full control task. As with learning a plant model, *online* methods may be used to overcome these difficulties, but the consequence is often a loss of stability. In

this chapter, only offline methods are considered, but it should be borne in mind that the motor programs may also be allowed to adapt at the same time as the controller, although a smaller learning rate is probably best.

CO-ORDINATING THE BEHAVIOUR OF THE CONTROLLERS

For continuous movements, a hierarchical control system may be treated as if it were a single network. This assumes that the higher controllers continuously supply control inputs to the motor program. When the behaviours are discrete, this may no longer be true, in which case some effort has to be made to control the timing of the two levels. For example, consider a system that takes as input the initial and desired state of a robot arm, and produces a reaching movement towards the target. In the case that the controller supplies brief signals that the motor program converts into extended sequences, the controller is expected to be silent during the rest of the movement. If the higher controller receives a coding of a new task before the movement is complete, there needs to be some mechanism to deal with the controller's output. This might mean storing the output in a stack until the current movement is complete, or the controllers output might just be given to the motor program regardless of whether the current movement is finished, in which case there will be some interference between the two movements. These issues become more apparent as the representations in the output of the higher controller become more discretised, e.g. "perform action A, then B, wait until signal S and then do C".

This hierarchical control system is very similar to the use of *structural shaping*, mentioned in section 4.7. In structural shaping, controllers are built to solve subparts of the task, so that a higher controller is able to select the behaviour of these controllers as single actions, so the lower controllers function very much like the debugged behaviours in subsumption architectures. The current control scheme is a kind of subsumption system, but one important point is that the lower controller(s) can have a variety of sources - for example, they might be the results of modelling of the known physiology of invertebrate motor circuits. This is potentially a powerful source of information about the functioning of specific circuits, in that models of these circuits may be used in larger systems that are trained to actually produce a behaviour. This gives a way to take the data from 'fictive'

motor experiments, and make predictions for *in vivo* studies.

5.3 Motor learning for a multijoint arm: Feedforward Control

In contrast to most robots, biological systems are characterised by large numbers of degrees of freedom, strong nonlinearities, and delays in signal transmission. These factors would make it very difficult to use traditional feedback control methods for these sorts of systems. Consequently, if we are interested in building robots that are inspired by biological systems, it seems likely that we will need different approaches to their control. The control strategies seen in biology are likely to have been optimised for exactly these problems, so it would seem wise to study the use of CPGs in plants that share these characteristics. Consequently, a highly redundant simulated robot arm was chosen as the plant in this chapter.

5.4 Introduction to the Experiments

The arm operates in a plane, and has 6 degrees of freedom: two positional, and 4 joints. As the targets are defined as points in the plane, there are 4 excess degrees of freedom For some tasks, the arm is a purely kinematic system; the output of the control system directly specify the joint angles, and the position of the shoulder. In other experiments, dynamics are added to the system, and the outputs of the controller specify the torques that act on the arm. As all the experiments reported here correspond to supervised learning, a plant model is required, whether the plant is kinematic or dynamic. In both the kinematic and dynamic case, the plant model was learned to a reasonable degree of accuracy *before* the controller was adapted. This is *offline* learning; the alternative is to train the model and the controller simultaneously (*online* learning). Which method is best depends on the problem, but it should be noted that there are difficulties associated with both methods. With online learning, the main problem is that the controller is initially trained with an inaccurate plant model, which will serve to hinder learning. In offline learning, an arbitrary criteria for model accuracy has to be chosen to decide when the model may be used to train the controller. Unfortunately, there is no clear relation between the complexity of the

control task, and necessary model accuracy. This issue was not addressed in the present thesis.

These experiments use feedforward control, so the only external inputs to the controller are those coding the desired behaviour. The inputs themselves were given to the net for the duration of the movement. This tends to make the input units' activity saturate, particularly during the early stages of learning, but shorter durations generally prevented learning. The arm was left unconstrained for a short period before the targets were applied, to give the controller enough time to move the arm from its initial position to the first target.

5.4.1 Learning Sequential Reaching Tasks

In the first experiments, the controller is required to cause the endpoint of the arm to be at certain places at certain times. The targets correspond to a set of buttons in the plane, to be touched in the correct order. With a few exceptions, the tasks are variants of those used in Jordan (1988). [1]

The four buttons are shown in Fig 5.2. We may call this a zero-dimensional task, in that the controller is required to follow the same trajectory every time. A skill learning task generally involves producing a family of trajectories, parameterised by some input variables; the dimensionality of the task refers to the number of inputs, or the number of ways in which the patterns may vary. A 2-dimensional skill task was used where the position of the buttons were varied in the plane. Fig 5.3 shows the extreme positions of the buttons; the actual positions was varied continuously within this square. As the arm is purely kinematic, the outputs of the control system directly specify the angles of the joints, and the xy position. Each joint was allowed to vary over a range of $+/- \pi/2$, and the translational units could move the arm over a range of $[0 : 2]$ units. The links were of length 0.16 units. These ranges specify the *workspace* of the arm, i.e. the set of points which can be reached. One point worth making is that the workspace is large compared to the variation in the position of the targets, which means that the arm is never operating at the edge of the workspace. The reason for this is that control becomes difficult in those

[1] Some of this material appears in (Miller, 1997)

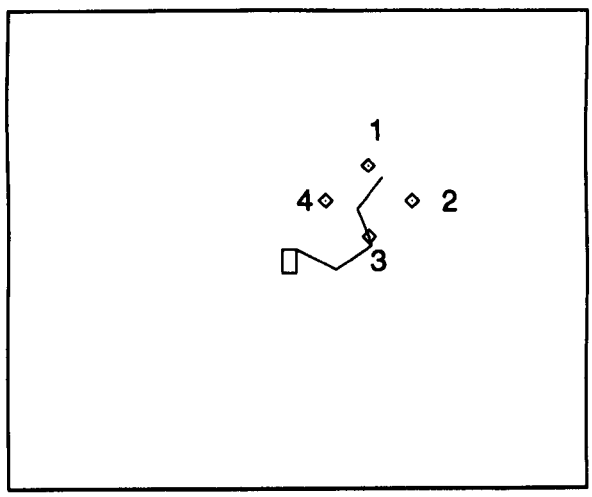

FIGURE 5.2. One set of targets. The figure shows four buttons, which are targets for the arm endpoints, together with the robot arm itself. The rectangle at one end of the robot arm represents the robot's shoulder, the other end is the endpoint (i.e. the hand). The numbers specify the order in which the buttons are to be touched

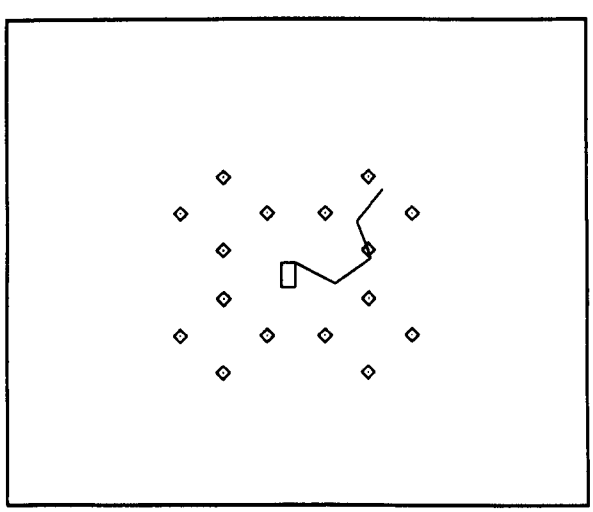

FIGURE 5.3. Varying the positions of the targets. The target's position was continuously varied in the plane; the figure shows the extremal positions

extreme regions, as the Jacobian of the kinematics mapping approaches singularity (i.e. some of the eigenvalues approach zero, which effectively means that the number of degrees of freedom goes down). Of course, real robots often have more limited workspaces (e.g. they are not usually able to translate), and so these issues have to be dealt with.

Given that the targets are only specified for the endpoint, the controller's task is ill-defined. There are two components to this redundancy: firstly, fixing the endpoint of the arm only partly specifies the state of the arm, secondly, targets are only specified at certain points in the trajectory, with the arm unconstrained at all other times. This redundancy usually requires some kind of regularisation to produce a well-defined problem. Jordan uses a cost function which penalises large movements from one time step to the next; the implicit smoothness of the recurrent net used here achieves the same effect. The reason for this is that the differential equations governing the evolution of the error derivatives are closely related to those governing the actual dynamics, and smoothness in the dynamics produces smoothness in the error derivatives. The result of this is to smear target information across time; the four spikes which correspond to the targets are converted to four temporally extended peaks in the error derivatives.

This is one advantage of using continuous-time recurrent nets with individual time constants, and the degree of temporal smoothness that the net displays is partly under the control of the experimenter, as it is always possible to define a prior distribution on the initial distribution of time constant values.

Task 1:

In the first task, the controller is required to press four buttons in the correct order, but at a specific speed chosen randomly from a continuous range, which is specified by one input to the controller.

Task 2:

In this task, the controller is required to press the same buttons as before, but the position of the buttons is continuously varied, and coded by two external inputs, one for each co-ordinate.

These tasks are essentially the same as the variable oscillations used in the previous chapter, with two differences. Firstly, there is a redundant nonlinear plant, and secondly,

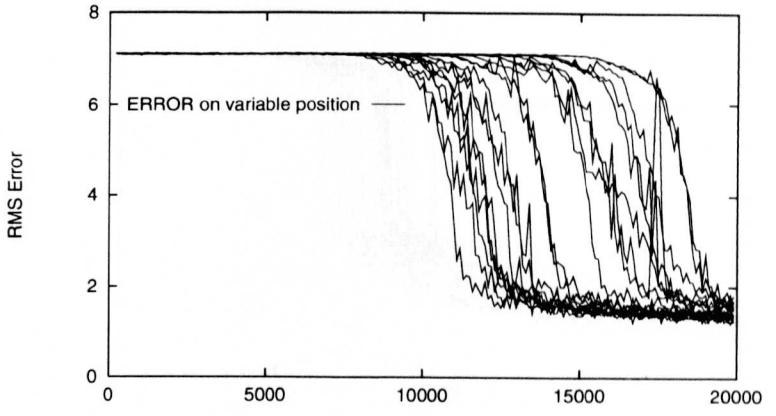

FIGURE 5.4. Learning curves for 20 controllers on the variable position task. Learning is seen to converge within 20 000 epochs

the targets only appear at 4 points in the sequence.

Recurrent networks with 12 units were initialised with random weights (N(0,1.0)) and random time constants (N(1.0,0.1)). The weights had a learning rate of 0.2; the time constants a learning rate of 0.1. As the targets are defined in the space of the arm's endpoint, the error derivatives are backpropagated through the plant model, to give error derivatives for the output of the controller. The plant model itself is just a static mapping, from the set of joint angles, and the position of the shoulder, to the position of the arm's endpoint. The model was implemented as a feedforward network, mapping from 6 degrees of freedom to 2. The network had 40 hidden units, and was trained on a set of random arm configurations until the error converged. The weights of the model were fixed throughout the rest of the experiments.

5.4.2 Results

The learning curves for 20 controllers on the variable position task are shown in figure 5.4; those for 20 controllers on the speed task are in figure 5.5.

The learning is seen to be unstable in just the same way as in the pattern generation tasks in the previous chapter. One interesting result of using such a strongly coupled system is that the oscillatory behaviour is spread across the whole of the network - even the input units show oscillations, an observation which holds for all the later systems trained

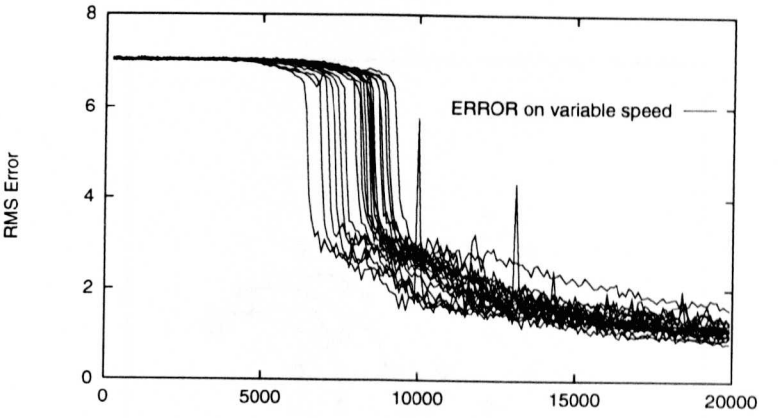

FIGURE 5.5. Learning curves for 20 controllers on the variable speed task. Learning is seen to converge more rapidly than the variable position task.

on this task. This fact may have implications for those researchers involved in trying to deduce computational function from the behaviour of physiological motor circuits. In invertebrates, these circuits are known to be strongly coupled systems, and the notion of a *multifunctional* circuit is becoming increasingly important. This implies that deducing the computational role of a neuron from its activity is likely to be very difficult, as this activity is also a strong function of all the other units in the network.

The performance of a typical trained controller is shown in figures 5.6 - 5.9. The controller is seen to be able to make flexible use of the degrees of freedom in the plant, and the movement seems to be well-coordinated.

The results of the previous chapter would suggest that much of the instability of learning corresponds to bifurcations in the controller's dynamics, and that constraining the behaviour of the controller would help learning. For a task that involves oscillatory behaviour, this would mean constraining the controller to be an oscillator. It is precisely because a recurrent net is such a general purpose system that these bifurcations arise: if the network was restricted to producing qualitatively correct behaviour, the learning should improve. This is the a possible role for the CPGs described earlier, for both pattern generation and feedforward control. In general, the CPGs may take on a more general role, such as implementing local feedback loops, or performing constraint satisfaction.

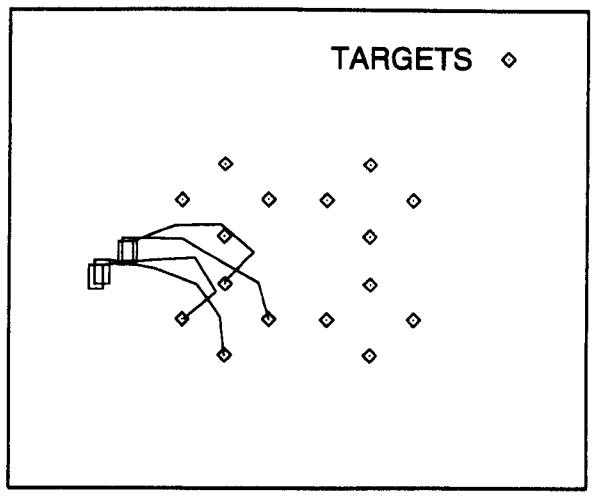

FIGURE 5.6. Performance of one controller on the variable position task. The plot shows the target buttons (in this case, those on the bottom-left), together with the robot. The robot is shown at four points in time; these correspond to the required times for the buttons to be touched.

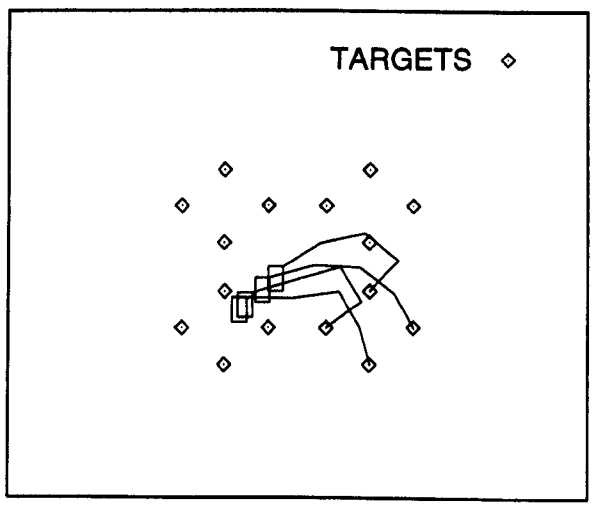

FIGURE 5.7. Performance of one controller on the variable position task. The target buttons are those on the bottom-right

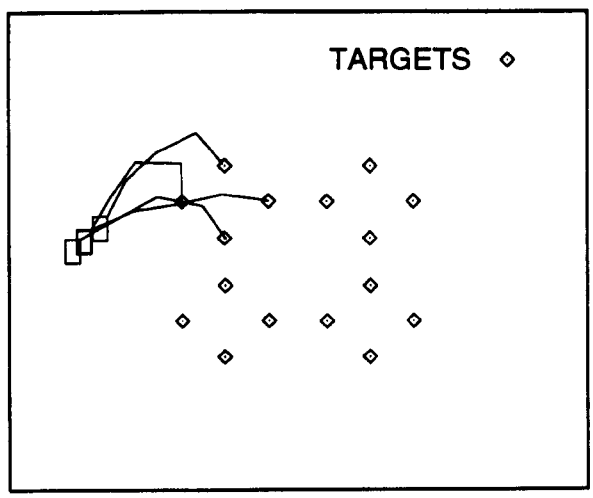

FIGURE 5.8. Performance of one controller on the variable position task. The target buttons are those on the top-left

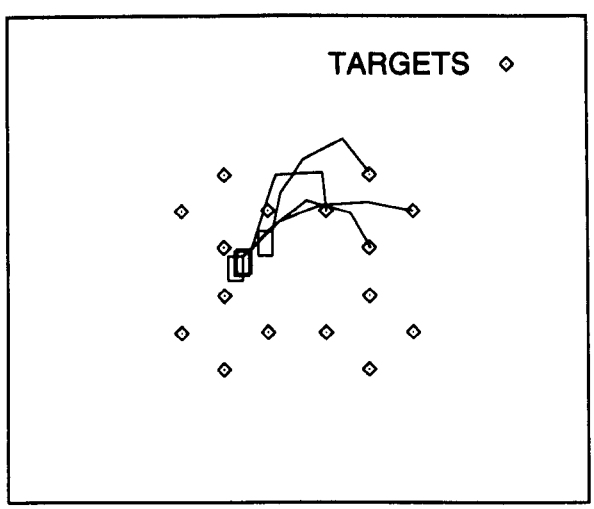

FIGURE 5.9. Performance of one controller on the variable position task. The target buttons are those on the top-right

5.5 Using CPGs in Control

Two methods are for building motor programs are considered; transfer from a simpler task, and a Genetic Algorithm.

5.6 Building CPGs from a simpler task

As in the previous chapter, a simpler version of the task was used to train a low-level controller, which would subsequently be used as a CPG. One obvious choice for a transfer task would be to touch the buttons at constant speed, and with constant position. Note that in general, deciding how to decompose a difficult learning task into subproblems is not at all straightforward, and the performance of learning may depend on getting this right. For the simple experiments considered here, this was not really an issue. CPGs were constructed by training networks on this task for 5000 epochs. The CPGs received no inputs during training; their task was to oscillate autonomously. For each task, the controller was a randomly initialised recurrent net coupled to the CPG as described, and the weights of the CPG were fixed during learning. The size of the controller and the CPG net were chosen so that the combined system had the same number of units (and weights) as the randomly initialised controllers used before (i.e. the controller had four units; the CPG eight). This architecture corresponds to *full control* design outlined above, where the controller continuously sends signals to the CPG, and where the entire state of the CPG is supplied as input.

5.6.1 Results

Learning curves for 20 networks on the fixed position task are shown in figure 5.10; as with the single oscillation tasks from chapter 4, the learning is slightly unstable, but converges within 5000 epochs.

The trained networks were then used as CPGs on the two tasks, variable position, and variable speed. 20 randomly initialised controllers were trained on **Task 1** and **Task2**, using CPGs from the fixed position task. The learning curves for these are shown in figures 5.11 and 5.12

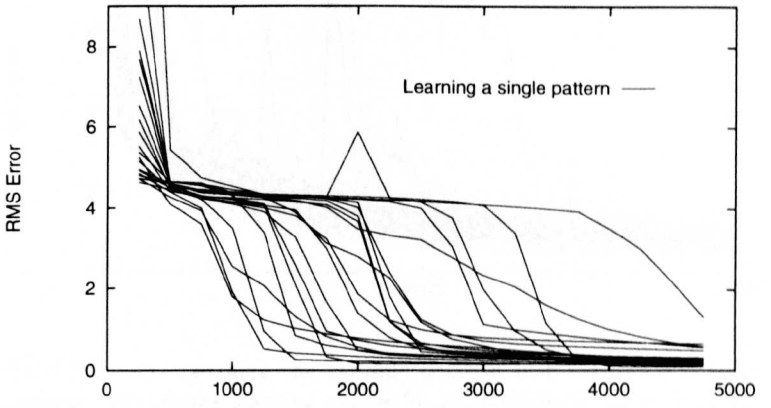

FIGURE 5.10. Learning curves for a network on the fixed position task. The resulting networks are then used as CPG on the full task

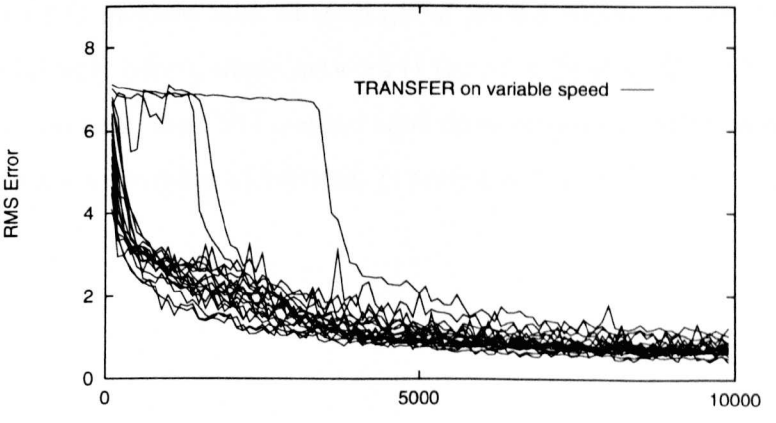

FIGURE 5.11. Learning the variable speed task with a CPG from shaping. Plot shows learning curves for 20 controllers. Note the values on the x-axis; learning trials were limited to 10 000 epochs

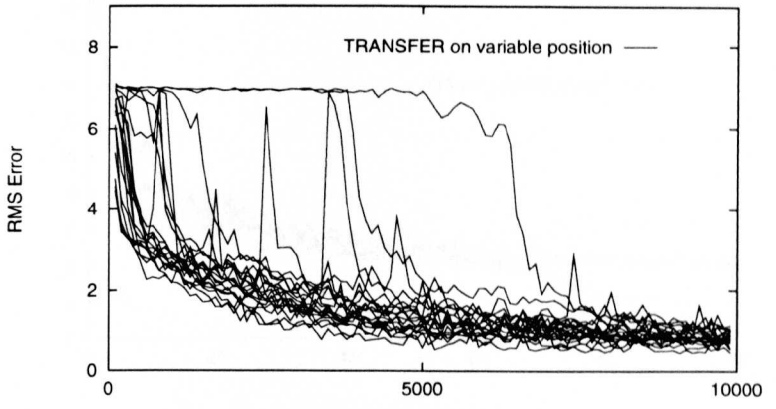

FIGURE 5.12. Learning the variable position task with a CPG from shaping. Plot shows learning curves for 20 controllers; again, learning trials were limited to 10 000 epochs

Statistical Analysis

The same method as in 4.7.2 was used to compare learning with and without CPGs. The error curves were measured by taking the time at which they first cross a threshold, in this case, the value 3.0 was chosen. The performance of learning with a CPG was assessed by adding the time-to-convergence for the CPG to that of the controller. All learning curves were smoothed with a window of 10 samples prior to analysis. In the variable speed task, the CPG method took an average of 2862.5 epochs to converge, compared with an average of 8695.0 for a single network (t test: $t = 21.408$; $df = 19$; $p < 0.001$). In the variable position task, the CPG method took an average of 4440.0 epochs to converge; single networks took an average of 14670.0. (t test: $t = 7.350$; $df = 19$; $p < 0.001$).

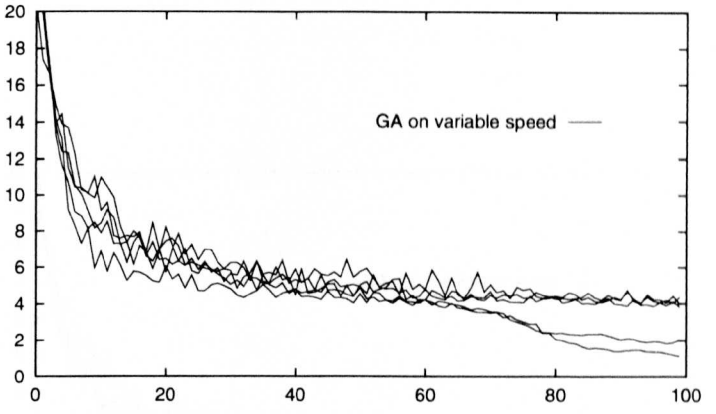

FIGURE 5.13. Performance of the GA on the variable speed task. Plot shows the mean evaluation of the whole population, for each of the 5 runs

5.7 Using a Genetic Algorithm to build the CPGs

As before, a Genetic Algorithm was used to build the CPGs. There were two GA experiments, one for **Task1** and one for **Task2**, with each experiment consisting 5 separate runs. The experiments were basically the same as those in section 4.8, with the difference that the genetic strings were slightly longer, as the CPGs are larger in this case. With 10 bits per parameter, the CPGs (8 units) were encoded with a string of 720 bits. The evaluation consisted of training the controller on the appropriate task for 800 epochs, and returning the mean error over the final 50 epochs. The learning rates for the controller were again set to high values: 0.7 for all the weights; 0.2 for the time constants.

5.7.1 Results

Figures 5.13 and 5.14 show the progression of the mean population evaluation of the two GA experiments. The evolved CPGs are again divided between stable oscillators, and unstable (damped) oscillators. The behaviour of two isolated CPGs from the GA are shown in Figs 5.15 and 5.16; one shows stable oscillations, the other has the form of a damped oscillator.

The evolved CPGs were assessed in a the same way as those in section 4.8.1. Figures 5.18 and 5.17 show learning curves for the two tasks, each using 20 controllers coupled to a randomly selectedCPG from the GA.

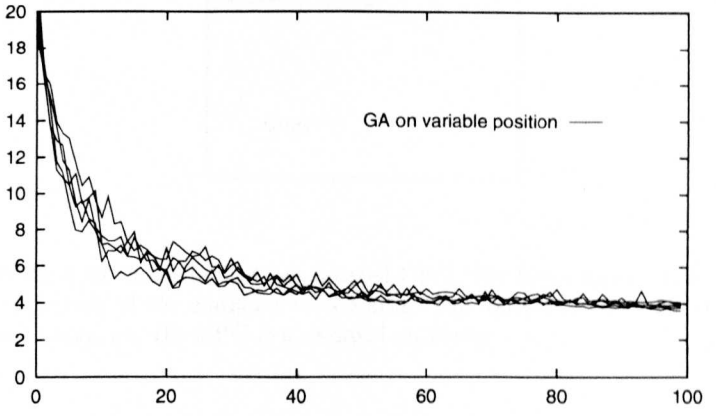

FIGURE 5.14. Performance of the GA on the variable position task. Plot shows the mean evaluation of the whole population, for each of the 5 runs

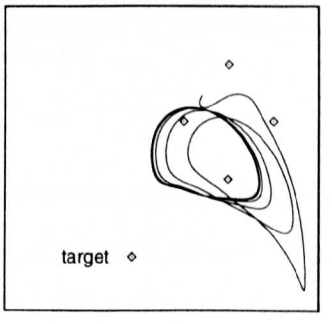

FIGURE 5.15. Behaviour of one isolated CPG. The figure shows the trajectory of the arm's endpoint, with just the CPG (i.e. there was no controller connected). The dynamics are that of a limit cycle

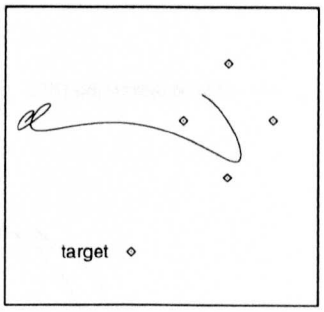

FIGURE 5.16. Behaviour of one isolated CPG. The figure again shows the trajectory of the endpoint of the arm. The dynamics spiral in to a fixed point, i.e. the CPG is a damped oscillator

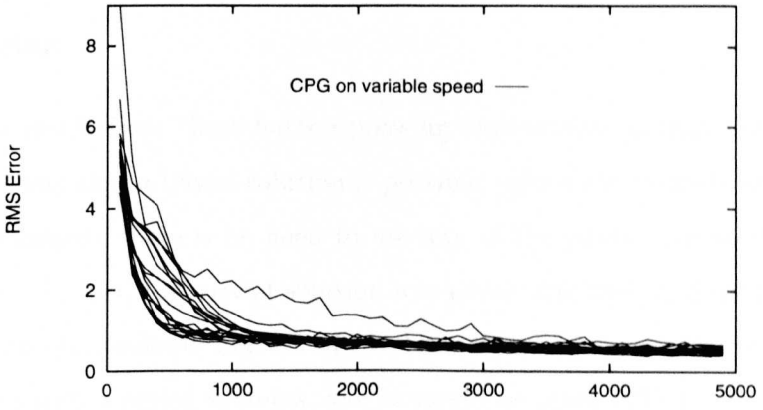

FIGURE 5.17. Learning curves for 20 controllers on the variable speed task, with a CPG from the GA

Statistical Analysis

The evolved CPGs were compared with those from the shaping experiment using the same time-to-convergence threshold of 3.0, and averaging the learning curves over 10 samples. In the variable position task, the CPGs from the GA took an average of 435.5 epochs, compared with an average of 1100.0 for those from shaping. (t test: $t = 2.599$; $df = 19$; $p < 0.02$). In the variable speed task, the evolved CPGs took an average of 380.5 epochs; those from shaping took an average of 815.0 (t test: $t = 3.065$; $df = 19$; $p < 0.01$)

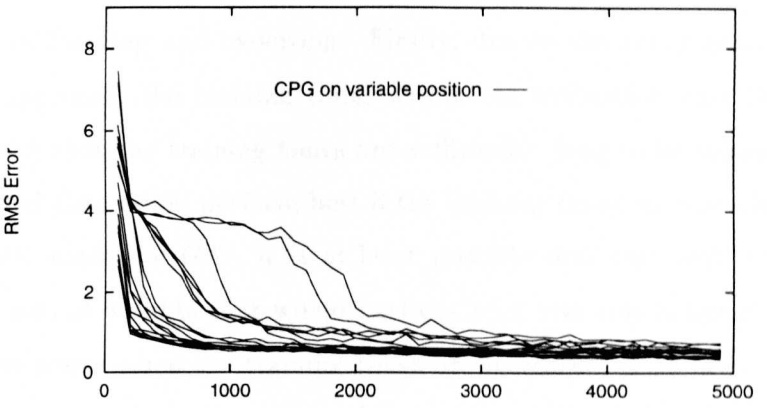

FIGURE 5.18. Learning curves for 20 controllers on the variable position task, with a CPG from the GA

5.8 Discussion

One interesting result from these button-pressing experiments is that the task has been set up in such a way that a trivial solution is possible - given the range over which the arm is allowed to translate, there is no need to use any of the joints, and so the nonlinearity may be removed. In fact, this trivial solution was never observed, and the learning always seemed to use all the available degrees of freedom. This is presumably because recurrent networks are strongly coupled systems, and it would be more difficult for the learning to remove those couplings than to cope with the nonlinearities. This means that a degree of co-articulation emerges directly from the choice of controller.

So far, we have discussed the use of CPGs for simple open-loop control tasks. Two methods for building CPGs have been used: using a controller trained on a simpler version of the task ('shaping'), and blind optimisation by a GA. The shaping method is the more straightforward of the two, although the GA is potentially a more powerful approach. GAs have been used by some researchers to build controllers from scratch, with some successes (Beer and Gallagher, 1992; Floreano and Mondada, 1995; Parisi et al., 1990; Nolfi et al., 1994). However, as the present approach uses a computationally demanding form of learning within the evaluation function, the whole procedure would be expected to scale very badly with the size of the problem, and so this is not really suitable as a

practical method for building controllers. Nevertheless, there are several points to note about this use of learning and evolution. Firstly, due to the computationally intensive nature of this approach, the training times within the evaluation were necessarily kept short. It is hoped that the training times are sufficiently long to be informative, so that CPGs are selected that would perform best if the training times were much longer. Given the opportunistic nature of GAs, it is at least possible that the 'best' CPGs would be precisely those sets of weights that would perform best over this training time, but may actually perform worse when the training times are increased. This does not seem to be the case for the experiments reported here, but, in general, this is hard to test. One reason for the difficulty is that the training runs used within the GA may only be long enough to reach a partial solution to the learning task that defines the evaluation function, and so the GA would be expected to then optimise with respect to that partial solution.

The other point concerns the nature of the evaluation procedure itself. Recall that the GA builds a set of weights which are then used as a component within a larger system. In fact, the weights from the GA are the only parts in this system that are not changed during learning. Given this, and the fact that the weights for the other components are randomised for each individual, the evaluation function is a complex function of the CPG's weights, and it is of interest that this procedure works as well as it does. Clearly, this procedure could not be used every time a controller was to be optimised, but it does raise the question of whether it is possible to find sets of rather general purpose CPGs that facilitate the learning of a variety of tasks. The notion of 'basis' sets has found widespread utility in such areas as function approximation and vision, and it seems reasonable to ask whether it would be possible to build sets of basis controllers; this is beyond the scope of this thesis, but will be investigated in future work.

5.9 Learning with a multiarm manipulator

In this set of experiments, a different robot arm was used; this is shown in Fig 5.19. This manipulator has two arms, each with 2 joints, so again there are 4 excess degrees of freedom. The redundancy for this arm is slightly different, as the targets may be assigned to either of the arms. Which arm receives the targets may decided in several ways; in this

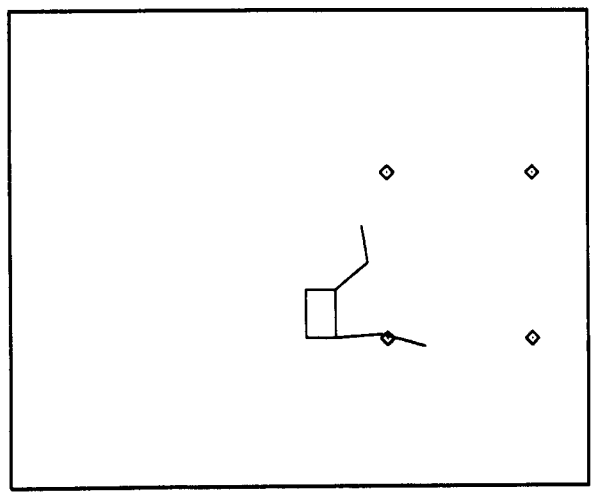

FIGURE 5.19. The second robot. This robot has two arms, each with 2 joints, and is able to translate, so again there are six degrees of freedom. The length of the links was the same as the previous robot

work, the arm is chosen simply by using whichever one is nearest the target at the time that the target appears. This means that the other arm is unconstrained; this also means that different arms may be used for different parts of the movement. As will be seen, this enables a hierarchical controller to de-couple the subproblem of choosing which arm to use, from the pattern generation problem.

5.10 Learning a fixed sequence

A single controller was trained on the fixed position version of the button-touching task. A plant model was trained on a set of random arm configurations, and its weights were frozen. The model was used to convert target information into the space of the controllers output.

5.10.1 Results

The learning curves for 20 controllers are shown in figure 5.20. Figure 5.21 shows the behaviour of one trained controller on this task; the controller is seen to be able to assign each of the button positions to the appropriate endpoint. In this example, each arm

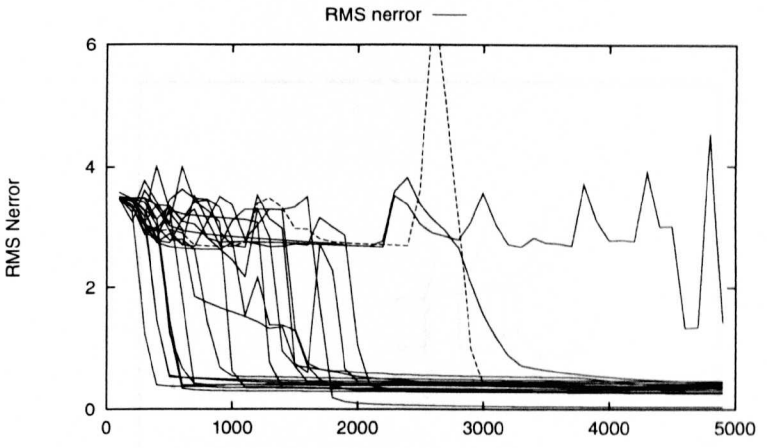

FIGURE 5.20. Learning curves for 20 controllers on the fixed position task, using the 2-arm system

receives just 2 targets per sequence, and is unconstrained the rest of the time. The effect of temporal smoothness in the network dynamics is to smooth this target information over time. This enables the system to show a degree of co-articulation - as one arm reaches towards its target, the other anticipates its own upcoming target.

5.11 Learning variable Patterns

In this section, the position of the buttons was continuously varied in the plane. The main question addressed here is: to what extent can the subproblems of choosing the appropriate arm, and generating the actual pattern be separated? As before, a two-stage controller was constructed. Unlike the previous experiment, the lower controller cannot be trained by using a fixed position pattern. This is because we want the final control system to be flexible in the way it assigns arms to targets, and this would involve complex transformations on the activity of a CPG trained on one pattern. Instead, the lower controller was built by training a system to generate reaching movements towards random points in the workspace. For each learning trial, the arm was randomly positioned in its workspace, and the targets appeared as four random points, equally spaced in time. This randomisation meant that, on average, each arm was selected the same number of times, and so received the same amount of training information. The inputs for each target were continuously applied until the point at which the target appeared. Applying the inputs

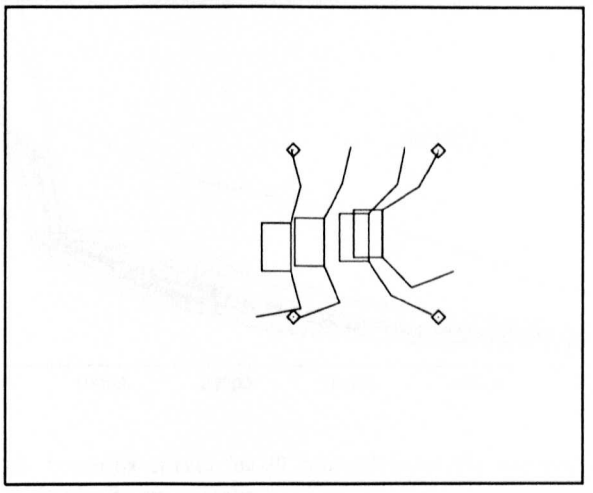

FIGURE 5.21. Behaviour of the 2-arm system on the fixed position task. The robots task is to touch the buttons in the correct order; top-right, bottom-right, bottom-left, top-left. The robot is shown for 4 points in time, corresponding to the times at which the targets appear

briefly seemed to prevent learning, as the information from the input had decayed away by the time the target arrived.

Once trained, such a network may be used as a CPG for the button-pressing task. The role of the CPG is to coordinate the activity of the two arms; the CPG has been trained to take two inputs, and generate reaching movements towards points in the plane. The role of the controller is to supply the CPG with a time series of such inputs, in order to touch the buttons at the correct times. The higher controller is now trained to press the buttons in the plane, where the position of the buttons were continuously varied.

5.11.1 Results

Learning curves for 20 controllers on the variable position task are presented in figure 5.22. Learning is seen to proceed rather slowly, as the controller has to learn oscillatory activity from scratch. The behaviour of one trained controller is shown in figures 5.23 to 5.26.

The final behaviour is seen to be flexible, in that the appropriate arm is chosen for each

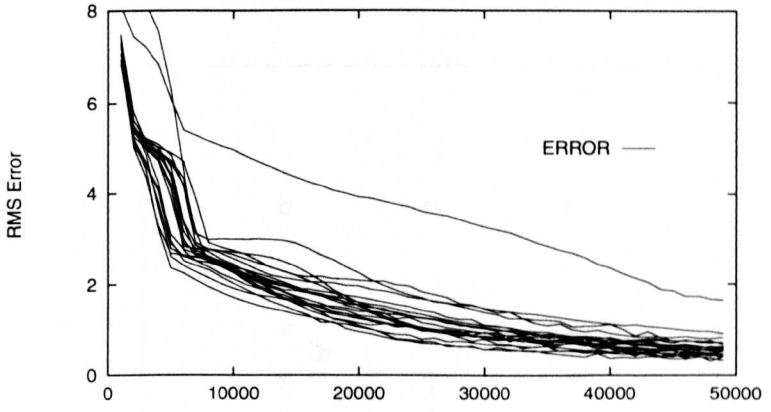

FIGURE 5.22. Learning curves for 20 controllers on the variable position task, using the 2-arm system

target. This means that different arms are used to reach for the same target, depending on the sequence in which the target is embedded.

What is interesting about this experiment is that it shows how the subparts of a control task may be decoupled. In this case, the assignment of each arm is handled by the lower controller, which means that the higher controller is now functioning in an effector-independent manner. Although the control system is a simple two-level hierarchy, the lower controller is the kind of component that could find use in a variety of disparate systems, each adapted for a different task. Of course, the notion of effector-independent motor programs has been extensively discussed in the experimental literature - the concept of a motor program was introduced partly to account for effector-independent behaviour. Perhaps the best-known example of this is handwriting - people produce approximately the same shape handwriting when they use their other hand, and also if they hold a pen in their mouths, or toes. This argues very strongly for a representation that is independent of the actual muscles that are used to perform the behaviour. The natural consequence of this is that there is a system that takes such an abstract entity, and produces a detailed motor command for the particular muscles chosen. The use of modular control systems to reproduce this kind of behaviour seems worthwhile, and deserves further investigation.

Finally, as the higher controller is producing outputs that are effector-independent, those same outputs could be given to a similar motor program for a different set of effectors.

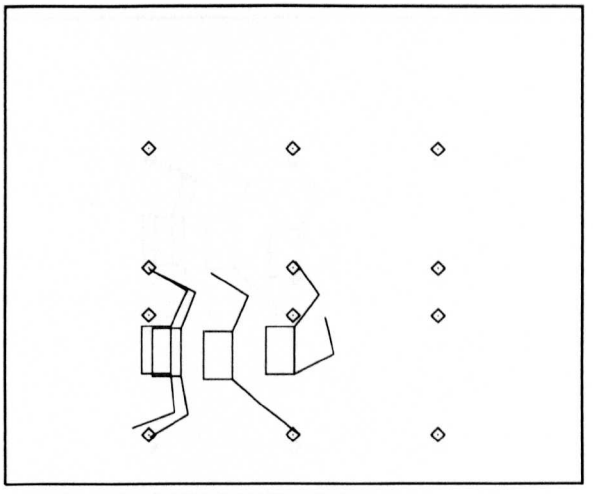

FIGURE 5.23. Performance of the controller with the 2-arm system on the variable position task. Again, the robot is shown for 4 points in time, the times at which the targets appear. The target pattern is at the bottom-left

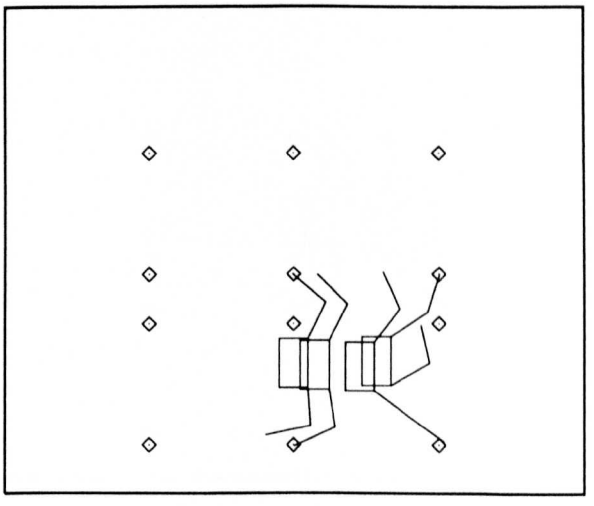

FIGURE 5.24. Performance of the controller with the 2-arm system on the variable position task; the target pattern is at bottom-right

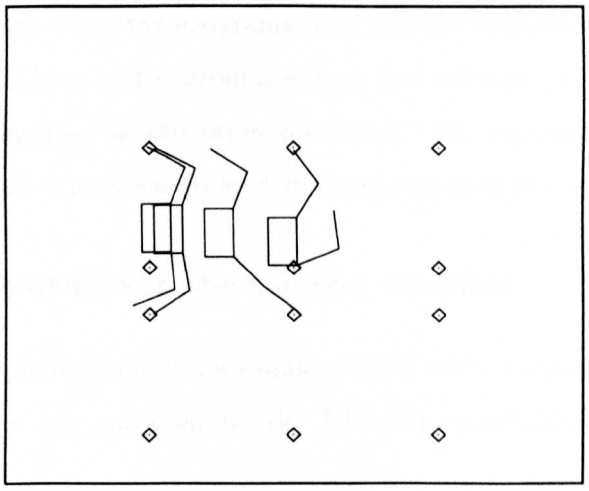

FIGURE 5.25. Performance of the controller with the 2-arm system on the variable position task; the target pattern in at the top-left

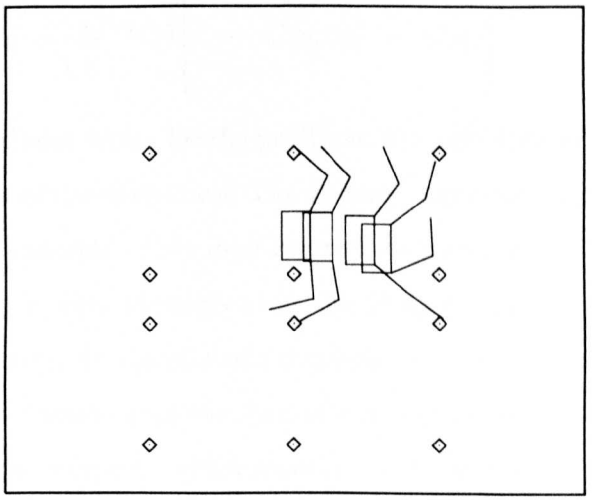

FIGURE 5.26. Performance of the controller with the 2-arm system on the variable position task; the target pattern is at the top-right

Ultimately, there would be a whole set of motor programs whose responsibility it was to flexibly assign outputs to actuators, so a single representation of an action could be given to any one of the set of motor programs, and similar behaviour would be shown with different actuators. These motor programs may just correspond to different sets of actuators, or different couplings of the same actuators. The separation of this kind of redundancy reduction enables patterns to be defined and learned at a more abstract level.

5.12 Introducing Dynamics in the Control Problem

To consider the effects of plant dynamics, a simulated arm with two joints was considered. The dynamics of the arm are governed by the following equation (from (Jordan and Rumelhart, 1992)):

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \tau$$

where q is the vector of joint angles, $M(q)$ is an inertia matrix, $C(q,\dot{q})$ represents the Coriolis and centripetal terms, and $G(q)$ is the effect of gravity. The control inputs specify τ, a vector of torques applied to the joints. This may be re-written as

$$\ddot{q} = M^{-1}(q)\left[\tau - C(q,\dot{q})\dot{q} - G(q)\right]$$

The arm state now includes terms for the positions and velocities of the limbs, and the acceleration plays the part of the next-state. The dynamic equations are exactly analogous to those in the kinematic example - they map current state and input onto the new state.

As supervised learning is used throughout in this chapter, a plant model is necessary to obtain the error derivatives. In the case of a dynamic system, there are several different approaches to modelling, depending on the kind of measurements that are available. The method adopted here is the simplest, which assumes that the state of the arm is able to be directly measured (which is not usually the case for terms such as the instantaneous velocity and acceleration). As the new state is just a function of the current state and the current inputs, a purely feedforward net may be used as a plant model. If this were not the case, some method would have to be used to estimate the plant state from observation

of the time series of control inputs and plant outputs. Note also that the use of a recurrent net to model this kind of data implicitly performs the same kind of state estimation, where the hidden units form features that represent information about the state variables which cannot be directly measured.

The model was implemented as a feedforward network, which maps from the current state of the arm, and the control inputs, to the next state - in this case, the next state is represented by the acceleration of the arm. The model had 50 hidden units, and was trained until learning was convergent, and then used throughout these experiments. The dynamic model was then used in exactly the same way as the plant model of the kinematic arm: it transforms error derivatives in the space of the plant outputs to that of the plant inputs. The dynamics equations govern the behaviour of the arm in joint space, and so a kinematic model is still needed to account for the mapping from joints to position. These two models were cascaded in sequence.

5.12.1 Learning Simple Reaching Movements

To generate reaching movements, consider the use of an adaptive feedback controller. This controller takes as input the state of the arm, as well as inputs specifying a target position. The goal of the feedback controller is to augment the plant dynamics in such a way that the target position is an attractor, so that the arm converges on the target from any initial state. Applying the desired position to such a system should cause the arm to relax onto the goal state. This controller is trained in the simplest way, by randomising the initial and goal state of the arm, and applying the desired position as a target. The target only appears at the end of the movement, so the arm is unconstrained the rest of the time. 20 such controllers were implemented as recurrent networks, and trained for 5000 epochs.

The trained controllers were then used as CPGs for another reaching experiment. In this case, a combined system is trained on a reaching movement, and the outputs of the controller specify a time series of target positions for the CPG. The controller is trained with more detailed target information; a target trajectory from the initial state to the goal was constructed by planning a sequence of states for the arm. The sequence was such that the velocity of the joints was described by a bell-shaped function, which is characteristic

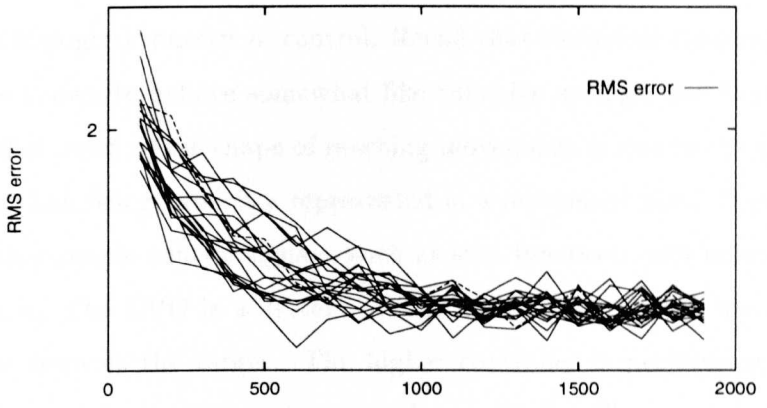

FIGURE 5.27. Learning curves for 20 controllers on the reaching task.

of primate hand movements. This plan corresponds to drawing a straight line between the initial and goal states (in joint space), and having the distance to the goal a cummulative Gaussian function of time.

The higher controller for this task was another recurrent network, fully connected to the CPG network. This meant that the controller was able to produce a movement by supplying the CPG with a time series of inputs. For a reaching movement of the same speed as that used to train the CPG, simple step inputs (to the CPG) are all that is required of the controller. For faster movements, there are several strategies available to the controller. The inputs to the CPG can be considered as virtual equilibrium positions for the system, so a fast movement could be produced by modifying the value of this attractor, moving it beyond the target to accelerate the arm, then back again to slow the arm down. Alternatively, as the controller also has connections to the plant inputs, fast movements could be produced by adding a transient input to the plant, in addition to that from the CPG.

This higher controller is trained with Distal Learning, as before. 20 controllers were implemented as recurrent networks, and trained for 2000 epochs.

5.12.2 Results

Learning curves for 20 controllers on this task are shown in fig 5.27.

The behaviour of the learned system is interesting, particularly in relation to the data on the effect of biological muscles in control. Recall that biological muscles, together with reflex loops are known to behave somewhat like tuneable springs, and that this has been used to argue that much of the shape of reaching movements is due to the dynamics of the muscles, rather than being explicitly represented in a movement plan. One implication of this is that rather simple control signals, such as step functions, can be used to generate plausible reaches. The CPG is a system of this nature; step inputs are converted into smooth reaches towards the target. The higher controller is performing a similar role to that hypothesised for supraspinal centres in reaching. The role of these centres is seen as specifying a time series of targets for the low-level feedback controller. For slow movements, the task for the higher centre is very simple - simply moving the target to the goal will cause the arm to converge onto the goal state, by virtue of feedback controller. Fast movements are more difficult to produce with this scheme - they generally require the stiffness of the arm to be modulated, or the position of the virtual target to be moved beyond the actual target.

In producing reaching movements of the same speed as those used to train the CPG, the outputs of the trained controller broadly resemble step functions. Perhaps more interesting is the situation where the reaching movements used to train the controller are different from those used for the motor program. For this case, the controller was trained on movements of varying speeds, ranging from 1.5 to 3 times those used for the motor programs. The faster movements are difficult for this system, as the motor program is a feedback system with a certain response time; this response time is too slow for the fast movements.

The consequence is that the controller tries to compensate for the slow response time by supplying the motor programs with more complex inputs than the near-step functions used in the slower movements. As can be seen in Fig 5.28, the controller supplies inputs that correspond to a 'virtual' target; this target is moved beyond the actual target to cause the arm to accelerate quickly, and then moved back to the target position.

This is not a very effective method for generating fast movements, as the 'virtual' target has to be accurately moved through a large range during a small time period, and the actual movement produced will depend quite critically on the timing of the virtual

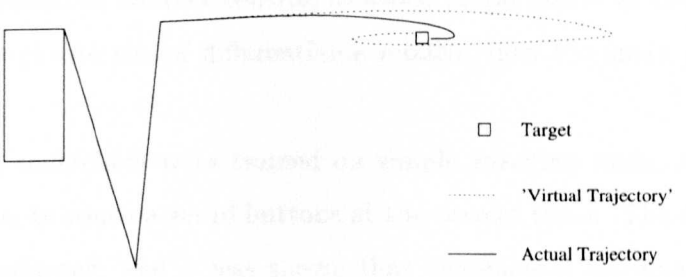

FIGURE 5.28. A fast reaching movement. The 'virtual trajectory' shows the time-series of inputs to the CPG; the actual trajectory is the time series of positions of the arm's endpoint. The fast movement is produced by moving the virtual point beyond the target to rapidly accelerate the arm, and then back towards the initial state to 'brake' the system

trajectory. A better approach would be to dynamically modulate the gain of the feedback in the motor program, or to include a feedforward element. Note that in the case of fast movements in humans, it is not clear what strategy is being used. This experiment considered learning in the absence of noise; it is possible that noise would bias the learning towards a different solution.

5.13 Limitations and Future Work

Perhaps the major limitation of this chapter lies in the use of simulated rather than real robots, and simple simulations at that. The effects of noise, measurement errors and signal delays have not been considered. Nevertheless, these abstract studies are of use in making clear certain ideas about improving the learning of complex controllers, and the systems discussed have a relevance for understanding the nature of motor programs. Future work will consider the learning of more complex tasks, particularly those that involve producing sets of behaviours. It is also planned to apply these ideas to robots that locomote, particularly walking systems.

5.14 Discussion

In this chapter, the use of recurrent neural networks in motor control has been considered, in the context of some simplified robotics tasks. The control structures used in these

experiments correspond to indirect control, in which target values at the plant output are transformed, through the use of differentiable models, into the space of the controllers parameters.

In section 5.4, controllers were trained on simple reaching tasks, which require the endpoint of the arm to touch a set of buttons at the correct times. The use of hierarchical methods was investigated, and it was shown that incremental learning can build CPGs in which learning is improved, in terms of time to convergence, and stability. A Genetic Algorithm was also used to build CPGs, and the learning is improved again. Although not practical as a method for building controllers in general, this combination of GAs and learning does offer a way of investigating motor circuits that are able to operate in many different environments and problems.

In section 5.9 a robot with two arms was used, which introduces a different kind of redundancy. It was shown that the use of modular architectures allows the different parts of the control problem to be de-coupled, and that the presence of a CPG can mean that the controller produces outputs which are independent of the particular arm used at each time step.

Finally, a dynamic robot arm was investigated. The CPGs were trained as feedback systems, and the controller was subsequently trained to drive the CPGs in an open-loop fashion to solve the task. This is a similar strategy to that seen in biological systems, where the dynamics of the muscles and reflex loops provide a simple feedback system. It has been hypothesised elsewhere that the role of motor cortex is to drive the equilibrium points of the lower feedback system.

These methods offer a powerful way of approaching difficult control tasks. The main difference with the usual use of shaping methods is that a variety of methods may be used to build the motor programs, not necessarily the current learning problem. This use of motor programs offers a way of de-coupling the various aspects of the control task, a fact that may have greater importance in more complex problems. This is partly because such incremental approaches are known to be desirable for difficult learning problems, and also because the notion of re-usable motor programs is attractive when a robot is required to carry out a range of different tasks. This is particularly clear in the GA experiments, where

the CPGs are optimised by increasing the learning performance for randomly initialised controllers. It is interesting that the GA experiments worked as well as they did, as the GA optimises a very complex function of the CPG weights. It would be interesting to see if these ideas could be extended, by building CPGs for sets of tasks; the hypothesis would be that it may be possible to build rather general-purpose CPGs, which improved learning for classes of controllers, and classes of tasks.

This work is related to other studies using motor programs in robotics, where those motor programs are often implemented as basis functions (Lipitkas et al., 1992; Lipitkas et al., 1993; Bock et al., 1993; Bock et al., 1996). The main difference in this work lies in the use of recurrent networks. This representation of motor programs means that the whole dynamic environment of the system is specified, and it was shown that the particular form of the learning task has a direct effect on the nature of the motor program dynamics. This also makes the addition of feedback a straightforward matter, which would not be the case if torque profiles were used. The price for this is an increase in complexity: this is manifested in the use of complex training algorithms for building the motor programs, and in the use of the same algorithms for performing credit assignment for the higher controller. In fact, the existence of such motor programs can often mean that simpler methods may be used to optimise the higher controller, as the controller's outputs have a simple relation with the behaviour of the plant. This is the subject of the next chapter.

6 Direct Motor Control and Reinforcement Learning

This chapter considers the use of reinforcement learning for motor control tasks. The specific problem addressed is the production of gait patterns with a system of coupled nonlinear oscillators. The use of motor programs for reinforcement agents is discussed, and a learning agent is introduced, whose actions control the parameters of the coupled oscillator system.

In section 6.6, this agent is trained to produce a set of gait patterns. The use of plant models is also discussed, and a model of the oscillator dynamics is learned concurrently with the reinforcement task.

In section 6.8, the learning agent is given control over the couplings between the oscillators; this is seen to have a mixed effect on performance.

6.1 Introduction

Many learning situations are best described as reinforcement learning problems. Reinforcement learning is characterised by a lack of target values for the system's response; rather, there is some performance measure that is to be maximised. Often, this feedback is delayed: for example the effectiveness of a reaching movement can only be evaluated once the whole of the movement has been performed. This introduces a significant *credit assignment* problem: in the light of the reinforcement received, how should the control actions be modified to increase the reinforcement? There are several components to this credit assignment problem. Firstly, *structural* credit assignment refers to understanding the differing contributions to the current reinforcement made by the various elements of the controller. An obvious case of this would be a controller with multiple outputs - a

simple scalar signal gives no information about which of the outputs was responsible for that reinforcement, and by how much. Other cases would include controllers that have internal components, for example, the hidden units in neural networks.

Temporal credit assignment refers to problems in which the reinforcement is delayed with respect to the relevant actions. This arises in problems such as game-playing, when the only evaluation occurs at the end of the game. A problem with a similar structure would be the navigation of a mobile robot towards a goal - the evaluation only changes in the goal state.

Note that one way of thinking about these credit assignment problems is that they constitute a distal learning problem, in that the training data is distal to the controller's outputs. The function which maps from controller outputs to reinforcement may be many-to-one, in which case the situation is similar to that of learning to invert the system dynamics (see 1.4.2). The temporal credit assignment problem arises when the reinforcement function is some time integral of the controller's output. In this case, the training data is still distal, not in terms of a different co-ordinate system, but distal in time.

This perspective implies that one way of solving credit assignment is to build models of the relevant mappings, and to use backpropagation to transform error derivatives in one space to the other. This is the approach taken in (Jordan and Jacobs, 1990), in which a model is learned of expected reinforcement as a function of controller output. If the maximum possible reinforcement is known, then this may be used as a target value, and distal learning used to generate targets for the controller. In contrast, most of the work in reinforcement learning takes a *direct* approach, in which the controller is optimised without learning such mappings (Barto et al., 1989; Barto et al., 1983; Gullapalli et al., 1992).

Reinforcement agents typically have to work with quite impoverished learning information. As a result, training times are often very high, even for relatively simple tasks. Learning of complex tasks from scratch is often impossible, as this usually requires that the controller has a significant probability of finding the solution by chance. With complex problems, this probability can be so low that no informative reinforcement is ever received. As a result, some researchers have taken an incremental approach, in much the

same way as that used in training recurrent nets. In reinforcement learning, this is usually called *'shaping'*, by analogy to incremental methods used in animal training. There are two basic approaches.

Temporal shaping refers to the incremental training of one controller over time. This usually means that the controller is trained on a series of approximations to the control task, each one more complex than the previous. The final task is the original control problem. During learning, the controller is trained to some predefined degree of competence before moving on to the next task. It is the role of the experimenter to decide on this criterion, and also to design the series of tasks.

In *structural* shaping, the architecture of the controller is decomposed into a hierarchy, with controllers at the lower level that are responsible for elemental behaviours, and controllers at higher levels that learn to co-ordinate the activity of the lower controllers. The lower controllers are trained on subparts of the control problem. Once trained, they may be thought of as providing a mapping from a static variable (their triggering signal) to a series of actions. The higher controller is now able to select these action sequences as single actions. If the low level controllers have learned subparts of the task, then the learning time for the higher controller should be greatly reduced. The experimenter is again faced with a design problem, in this case, the problem of how best to decompose the control task into elemental behaviours.

Clearly the timing of the controllers at different levels is crucial. The low-level controllers must be able to finish their action sequences before a new action sequence is selected, and any output of the higher controller in the meantime has to be dealt with, possibly by placing it on a stack. This implies a degree of co-ordination between the different levels. There are several ways of achieving this co-ordination, depending on the assumptions that are made. Perhaps the simplest method would be to assume (or ensure) that the low-level controllers would produce action sequences that would always be below a fixed number of steps. The higher controller could then simply wait for that number of steps before issuing a new command signal. This is obviously a rather inflexible system. A more sophisticated system would be to have each low level controller send a special 'finish' signal back to the higher controller when its action sequence had terminated. Alterna-

tively, it could be the responsibility of the higher controller to detect when the low level system had finished; this has to be achieved by detecting features in the sensory inputs that signal the end of the action sequence. This is probably the most general, and the most difficult method.

The experimenter has a considerable amount of freedom in the manner in which the control task is decomposed; this also provides an opportunity to represent any prior knowledge that may be available. This is also true in temporal shaping, although in a less obvious way. With a hierarchical system, modules that represent elemental behaviours may be treated as functional units, and provide a way to represent what is similar across different, but related tasks. Some tasks have an obvious relation to biological systems, and it may be desirable to include knowledge of the functioning of these biological systems in the design of the controller. This knowledge can occur at a variety of levels, ranging from detailed models of physiology, to general understanding about the functioning of systems. The hierarchical nature of structural shaping methods means that all these types of knowledge may be included. In the previous chapter, hierarchical controllers were built that use modules that were clearly related to pattern generation systems. The co-ordination between the various levels was performed with the machinery of backpropagation. One interesting consequence of this is that a strict hierarchy is not necessary, as the backpropagation makes no assumptions about the particular pattern of connectivity, nor about the place at which inputs and targets appear. In this chapter, direct methods are used to optimise the controller.

In some cases, particularly discrete movements, the situation is simplified, and the higher level controller can be considered to be a 'command system' for the CPG. There are generally two ways to implement this. In the first case, the controller continuously supplies a signal to the CPG, although this signal may in fact be constant. The other possibility is that the controller supplies a brief 'go' signal to the CPG, which is somewhat similar to setting the initial conditions of the CPG. In this case, the controller is required to be silent during the operation of the CPG.

In the systems used in this thesis, the situation is slightly different. This is because the higher controllers will usually be continuously sending control signals to the CPGs,

rather than simple on-off signals. Also, the CPG is a largely distributed system, so there are no separate modules for individual behaviours. It is possible to enforce modularity on the CPG, by dividing it into subnetworks, and assigning a different behavioural role to each of the subnets. However, this chapter considers CPGs which do not have separate systems for each behaviour; the individual behaviours emerge from the CPG dynamics.

6.2 Direct vs. Indirect methods

It is commonly assumed that indirect control methods are superior to direct techniques, and that direct *learning* will always proceed more slowly that indirect learning. In fact, there are several trade-offs involved in both methods, and this means that reinforcement techniques will outperform supervised learning on some tasks (Gullapalli, 1991). The main trade-off with indirect control is that it requires an accurate plant model, and learning such a model increases training time. These issues were mentioned in section 5.3. There are other reasons why reinforcement learning may be attractive. Part of the difficulty of training large networks with supervised learning is that the hidden units often receive quite distorted training information, as the error derivatives are backpropagated through the network. This is particularly true for those units which are distant from the output. In such a situation, a reinforcement signal, broadcast to all the units, may be used to ensure that all units receive some useful training information, independently of the distorting effects of backpropagation.

A separate issue concerns the availability of training information. Indirect control assumes the presence of a plant model; it also assumes that training data is available in the space of that model, or in a form that can be easily transformed into that space. For example, a forward model of a robot arm has its outputs in the space of the positions of the arm's joints. For this model to be useful, there has to be training data defined in that same space, or in a form which may be readily transformed into that space. In the case of the robot arm, there is a natural co-ordinate system to describe the requirements of a reaching behaviour, but this is not always the case. Consider the case of a walking robot with many legs. The state space of this system will have to have terms for each of the degrees of freedom, i.e. the individual joints, of which there are many. We might call this the fine-

scale state of the system. The desired behaviour of the system might be defined at a quite abstract level, including terms for the direction and speed of locomotion, and the stability of the robot. These measures are *some* function of the fine-scale state of the system, but that function might be very complex, and its inverse even more so, particularly if the system is highly redundant. However, redundancy is prevalent in walking systems, and so control of the fine-scale state of the plant is likely to involve learning highly complex mappings. To use a fine-scale model in control, the desired behaviour, coded in the abstract space, would first have to be transformed into the fine-scale state space, and then transformed via the model itself to the space of the controller's outputs.

This is all possible in principle, but it is far from obvious that this is always the best approach. The attraction of reinforcement learning is that no attempt is made to model *any* of these mappings; the controller is adapted directly from the reinforcement signal. This removes two kinds of models: the model of the behaviour of the plant, and the model that maps the task demands into the plant's state space.

6.3 Motor Programs for Reinforcement Agents

There is one further distinction between indirect and direct methods, concerning the use of motor programs in control. Motor programs have been described as essentially fixed dynamic systems that are used as components in a control scheme. Part of the attraction of using recurrent nets to implement these systems is that the same methods for performing distal learning may still be used - the error derivatives now get backpropagated through an additional system before they reach the controller. In fact, using distal learning, this is *necessarily* the case; any component that is added to the control scheme has to have its behaviour accounted for, in this case by recurrent backpropagation.

In contrast, the outputs of a reinforcement learning agent have completely unmodelled effects. This gives the designer an enormous amount of freedom in the way the control system is constructed. To see this, consider an architecture prevalent in biology, where the motor program is a set of coupled oscillators. Coupled oscillators have been discussed previously in the context of animal walking, and it was seen that these circuits perform the task of generating fine-scale control signals on the basis of quite general inputs, and

that they co-ordinate the behaviour of the legs. One of the interesting properties of these circuits it that they are usually capable of generating several discrete *gaits*, or sets of relative phases. It is not clear in general whether the various gaits are generated by essentially the same circuit, or whether there are sets of circuits, each responsible for different gaits. A related issue concerns which parameters of the system have to be altered to change the current gait, in particular, whether the couplings between oscillators have to be altered, or whether a simple driving signal can switch between different gaits.

This may seem a rather moot point, but it would be very important for any artificial control system which used coupled oscillators as one of its components. Again, consider the case of a legged robot, in which a motor program is constructed from a system of coupled oscillators. If production of the required gaits involved changing the strengths of the couplings between oscillators, then it would be desirable to have a controller that could select these couplings *as a control action*. Now, this might be problematic for a controller trained with supervised learning, as the effect of control actions have to be modelled explicitly. The actions of the controller are to set the values of the *weights* in another network. As a result, obtaining error derivatives for the controller's output is now more complicated.

The point is that a reinforcement agent produces outputs whose effects are completely unmodelled, and so it is straightforward to build a controller whose actions control the strength of connections in other networks. The reason reinforcement agents have this freedom is that they use a different method for solving the credit assignment problem. It *may* be the case that having such complex actions makes credit assignment more difficult; alternatively, it is not too difficult to think of cases where this approach might actually help credit assignment. For this to happen, the mapping from control actions to reinforcement received would have to have been simplified as a consequence of the motor program.

In this chapter, we consider reinforcement agents whose actions select the couplings between dynamic systems: in the present case, the individual systems are nonlinear oscillators. This is a rather general approach; it is possible to see how a set of motor programs could be built, each showing different types of dynamical behaviour, and a controller, selecting the couplings between the systems, could generate a very complex repertoire of

behaviours.

There are other possibilities - for example, the work on neuromodulators in invertebrate circuits suggests the use of controllers that set *global* properties of a motor circuit (Marder, 1988). These might correspond to the time constants in a subpart of the motor program, or scale the magnitude of all the weights in a particular subnet. The ability of a controller to re-configure motor circuits by control actions is a powerful way for a system with few outputs to show a large degree of variability. There is a tradeoff here however: if the effects of the controller outputs are to radically change the dynamics of the motor circuit, then small changes in output may correspond to large changes in behaviour. Such strong nonlinearity makes the credit assignment problem correspondingly harder, as the performance index will have a complex relation with the output of the controller. Perhaps the most interesting cases are those in which the mapping from controls to performance index is relatively simple, although the controls have highly nonlinear effects on the plant output. In these cases, the complexity of the control-to-plant state mapping is partly undone by the control-to-evaluation function. Many motor behaviours are like this: gait transitions correspond to large changes in the limb trajectories, but to a description at the level of 'speed and direction of locomotion', the behaviour changes smoothly with controls.

One final possibility is to remove the neural element from the motor program altogether. This was seen to be useful in the work of Lipitkas *etal*, where the motor program was a simple waveform generator. In general, the motor program could be described by any set of differential equations, with parameters set by the controller. In principle, this is no more general than using a recurrent net, but it may give a more natural way to build in the kind of dynamics we wish, without having to model those dynamics with a recurrent net.

6.4 Coupled Oscillators as CPGs

This chapter will consider the use of coupled oscillators as a motor program. Coupled oscillators are widespread in biology, and are the central mechanisms for the production of diverse forms of locomotion, from swimming in the lamprey to flying in insects. One particularly interesting body of work is concerned with the generation of gaits and gait

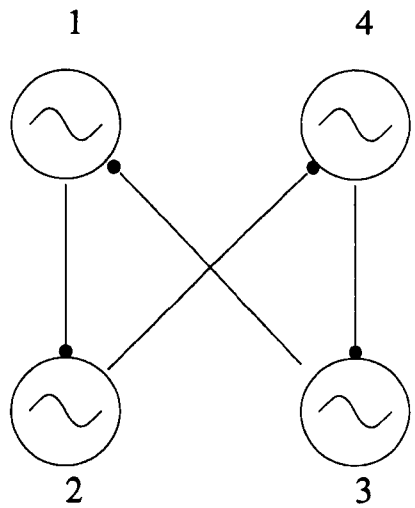

FIGURE 6.1. Locomotor CPG consisting of coupled oscillators. The solid lines are inhibitory couplings, with the filled circle next to the inhibited oscillator. The arrangement of the oscillators corresponds to the positions of the legs; oscillators 1 and 4 are the front-left and front-right respectively; 2 and 3 are the back-left and back-right respectively

transitions in legged animals, and the oscillatory mechanisms that are responsible for the behaviour. Gaits are considered to be stable relative phase relations between the limbs; these occur when the limbs are entrained at a common frequency. Quadrupeds are capable of producing a number of gaits, most notably the walk, trot and gallop. The architectures responsible are still largely unknown, so most of the modelling in this area is at an abstract level. A typical model would be a system which has one oscillator for each limb, and a set of interconnections to co-ordinate the behaviour of each oscillator. Some researchers use systems in which the couplings between the oscillators are changed to produce the various gaits (Yuasa and Ito, 1990), while others have shown that some systems can produce different gaits by changing a driving signal comprising a constant and a periodic component (Collins and Richmond, 1994).

A typical oscillator network is shown in Fig 6.1. In the context of the present thesis, a coupled oscillator system is a particularly powerful kind of CPG net, which has its own complex dynamics, but can use external signals to modify its behaviour. As far as a higher controller is concerned, its presence should make gait-related tasks much easier. The previous chapters have used Distal Learning to train the controller, where the effect

of adding the CPG was accounted for by backpropagation. However, this procedure is problematic in the present case, for two reasons. Firstly, it was seen in section 4.9.1 that backpropagation is generally uninformative when the system is distant from the desired behaviour. A system which produces a number of distinct gaits has a number of modes of behaviour, corresponding to a set of limit cycle attractors. In order for the system to switch from one gait to another, the learning has to know in which direction to change each of the inputs to jump out of the current attractor and into another. In other words, the desired situation is that the system is in one gait, another gait is applied as target, and the learning procedure is informative about which way to change the inputs to cause the system to move closer to the target gait.

This is precisely the circumstances in which backpropagation would be expected to perform poorly, as the learning procedure essentially calculates a linear approximation of the system dynamics, around the current trajectory. Suppose that the system is in the walk gait, and the target corresponds to a running gait. The flow of a system which is on or near an attractor is a vector field which pulls the system towards the attractor, so a linear approximation of the dynamics around the attractor will simply give a system which contracts uniformly onto the attractor. There is no information in this approximation about the presence of any of the other attractors, nor about the behaviour of the system as it loses stability (i.e. at the boundaries between the attractors' basins). In that case, the error derivatives are unlikely to lead the system towards the target gait. If gradient information is only informative in a small region, direct methods might be expected to perform best.

The other reason has to do with the training data. Supervised learning requires a set of input/target pairs for the system. For the coupled oscillator system, the inputs are the descending controls that select the various gaits, and the targets are the outputs of the individual oscillators in each of the gaits. Essentially, this requires knowing in advance what the outputs of the oscillators would be in the different gaits. However, the task is more likely to be specified as a set of relative phase relations between the oscillators. It is possible to perform supervised learning with such target information, but this is more naturally represented as a reinforcement problem. In a later section, forms of supervised

learning will be considered which deal with more abstract measures of the system output, such as relative phase and frequency of oscillation.

The particular form of the oscillator system is chosen from one of the setups used in (Collins and Richmond, 1994). The individual oscillators are described by the Stein model for an oscillatory neuron. The neuron dynamics are given by

$$\dot{x}_i = a\left[-x_i + \frac{1}{1 + exp(-f_{ci} - by_i + bz_i)}\right] \tag{6.1}$$

where

$$\dot{y}_i = x_i - py_i$$

$$\dot{z}_i = x_i - qz_i$$

and the index i refers to the four oscillators. The term f_{ci} is a forcing term, consisting of a constant component, and a periodic component:

$$f_{ci} = f\left[1 + k_1 sin(k_2 t) + \sum_{j=1}^{4} \lambda_{ji} x_j\right] \tag{6.2}$$

The coupling between the oscillators appears in the term $\lambda_{ji}x_j$, with λ_{ji} representing the strength of oscillator j's effect on oscillator i. This coupling term is set to -0.2 if there is a connection between the oscillators, and 0.0 otherwise. The other parameters, i.e. a, b, p, q, f, k_1 and k_2 were set to be identical in each of the oscillators. As the system dynamics are rather complex, a more sophisticated method was chosen to integrate the plant equations. All the simulations in this chapter used a 4th order Runge-Kutta integration method, with a fixed step size of 0.005 s, which corresponds exactly to the method used in the Collins and Richmond study.

The usefulness of a particular coupled oscillator system for gait production depends on both the number and stability of the different gaits it supports, as well as the ability to switch between the various gaits. The production of a gait consists of either starting the system with random initial conditions, and converging onto the gait, or switching from one gait to another. The switching task may be considered to be a subset of the

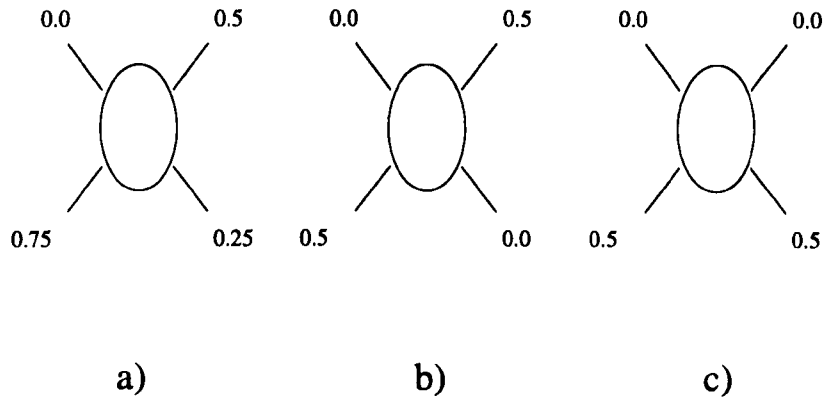

FIGURE 6.2. Phase relations for three common quadruped gaits: (a) walk, (b) trot and (c) bound. The numbers are fractions of whole cycles, so the bound gait corresponds to the front legs being π radians out of phase with the back legs

converging task. The gaits themselves correspond to attractors in the system dynamics, and switching between them corresponds to phase transitions of the system. The gaits are described by a set of relative phase relations between the outputs; these are shown for three common gaits in figure 6.2. The Stein model is described in the Collins study as supporting 3 gaits, the walk, trot and bound, and four possible transitions: walk-to-trot, walk-to-bound, trot-to-walk, and trot-to-bound. These gaits, and the gait transitions are produced by setting the values of four parameters: a, f, k_1 and k_2; the couplings λ_{ji} between the oscillators are kept constant. The values of the parameters were chosen by hand in Collins and Richmond; this chapter considers the use of reinforcement techniques to set the same parameters.

6.5 Gait Patterns and Coupled Oscillators

The experiments in this chapter are only concerned with the task of reliably producing a gait from random initial conditions; the switching task was not considered. This is partly because switching is a subset of the gait production task, and also because switching from gait 1 to gait 2 requires that gait 1 can be reliably produced. If gait 1 can only be produced some of the time, the simulation times increase dramatically, and the whole procedure becomes impractical. As will be seen, some of the gaits are of this nature.

6.5.1 Methodology: Stochastic Automata and Reinforcement Learning

Consider an agent that takes a set of sensory inputs, and produces an action which is drawn from a distribution with a certain mean and variance. The output of the agent would be stochastic, and real-valued, and this is the basis for the Stochastic Real-Valued (SRV) learning rule (Gullapalli, 1992; Gullapalli, 1990). The agent receives reinforcement from the environment, and modifies the distribution that generates the outputs so as to try to increase its future expected reinforcement. The reinforcement value is assumed to lie in $[0, 1]$; if this is not the case, then the agent will have to estimate the possible range of rewards, which complicates the situation significantly. There are only two parameters for this distribution, the mean μ and the variance σ, and they are both functions of the inputs.

The learning algorithm adjusts the two sets of parameters, thus adjusting the input-dependent distribution. The ideal behaviour of the SRV unit has the following properties:

- It should learn to associate with each sensory input an output which has the highest expected reinforcement.
- It should use exploratory behaviour to improve its performance when it is receiving low reinforcement
- It should discriminate between cases when its performance is poor, and cases when it is near to optimal, so that exploratory behaviour is reduced in cases where the performance is good.

In the simplest case of a single linear output unit, the mean is calculated as a linear combination of the inputs:

$$\mu(t) = \sum_{i=1}^{n} \theta_i(t) x_i(t) + \theta_{bias}(t) \tag{6.3}$$

where the x_i are the sensory inputs, and the θ_i are the weights. According to the desired behaviour of the agent, the variance of the output should depend on the expected reinforcement. If the expected reinforcement is high, then the performance is closer to being optimal, and so the variance should be low. Conversely, when the performance is

relatively low, the variance should be greater, so that alternative outputs may be explored. This requires an estimate of the current expected reinforcement. In the simplest case, this will be another linear function of the inputs:

$$\hat{r}(t) = \sum_{i=1}^{n} \phi_i(t)x_i(t) + \phi_{bias}(t) \tag{6.4}$$

The variance of the output is calculated as a function of the expected reinforcement for the current input, i.e. $\sigma = s(\hat{r})$. The function used is a monotonically decreasing, nonnegative function that also has the property that $s(1.0) = 0.0$. This means that when maximum reinforcement is expected, the output will be deterministic, so exploration is turned off. In this chapter, s is defined to be

$$s(\hat{r}(t)) = max\left[\left[\frac{1.0 - \hat{r}(t)}{5.0}\right], 0.0\right]$$

Note that one problem with this function is that exploration is never turned off if it is not possible to achieve an average reinforcement of 1.0; in other words, if perfect performance can never be achieved for the control task, exploration is never turned off. The activation of the unit is now given as a random variable, drawn from a Normal distribution with mean μ and variance σ: $a(t) = N(\mu, \sigma)$. Finally, the activation of the unit is passed through a squashing function to generate the output; the usual sigmoid f was chosen here, where

$$f(z) = \frac{1}{1 + e^{-z}}$$

THE SRV LEARNING RULE

Conceptually, the SRV learning rule is very simple. There are two sets of parameters to be adjusted: the ϕ_i that are used to estimate the expected reinforcement, and the θ_i that generate the output. The learning rule for the ϕ_i is straightforward. For a given input, we want to estimate the expected reinforcement. As the current reinforcement is an unbiased estimate of the expected reinforcement, we can just use the usual supervised learning with the current reinforcement as a target value:

$$\phi_i(t) = \phi_i(t) + \beta\Delta_\phi(t)x_i(t)$$

where β is a learning rate, and

$$\Delta_\phi(t) = r(t) - \hat{r}(t)$$

The learning for the agents output is a little more involved. Basically, the principle behind the algorithm is simple. If the output of the unit is higher than its mean value (as calculated by the weighted sum of inputs), and the reinforcement received is higher than expected, then the mean output should be increased. Conversely, if the reinforcement is lower than the expected value, then the output should be reduced. Essentially, the algorithm forms an estimate of the correlation between perturbations in output and perturbations in reinforcement. Here is the weight change rule:

$$\theta_i(t+1) = \theta_i(t) + \alpha\Delta_\theta(t)x_i(t)$$

where α is the learning rate, and

$$\Delta_w(t) = (r(t) - \hat{r}(t))\left[\frac{a(t) - \mu(t)}{\sigma(t)}\right] \tag{6.5}$$

In practice, the denominator in 6.5 has a small constant added to prevent the weight change diverging as the variance of the unit approaches 0. So far, a simple linear net has been assumed with one output unit. For a network of SRV units, the same reinforcement signal may be used to update the weights for all the units. Alternatively, we may use some form of credit assignment to attempt to separate out the differing contributions made by the different units.

It is a straightforward matter to extend this linear controller to a feedforward net, and this setup was used in (Gullapalli et al., 1992) to learn a peg-in-hole insertion task for an industrial robot. The controller was a multilayer feedforward net, with SRV units forming the output layer. The SRV learning rule specifies how to change the top layer of weights, and the procedure may be thought of as calculating error derivatives for the output units (with a change of sign: the gradient of expected reinforcement is in the opposite direction to the gradient of expected error). Standard backpropagation is then used to obtain error derivatives for the other layers in the controller.

Under certain circumstances, it can be shown that the SRV rule does in fact form an unbiased estimate of the gradient of the expected reinforcement, in which case the formulation is exactly that used to derive the backpropagation procedure. The only difference is that the algorithm performs gradient *ascent* of expected reinforcement, rather than gradient descent of a cost function.

In the basic SRV scheme described above, there is no modelling of the plant at all. In that case, the introduction of motor programs is completely straightforward, as the goal of the reinforcement learning is still to control an unmodelled, nonlinear system. All that has changed is that the system is now the coupled motor program-plant system. Of course, the motor programs may have a large effect on the difficulty of this control problem; whether or not the control problem is made easier depends on the dynamics of the plant, the motor programs, and the reinforcement task itself.

6.5.2 The evaluation function

Reinforcement learning involves the optimisation of an index of performance, or evaluation function. Most of the problems considered in the reinforcement learning literature have been relatively simple, and so the problem of designing an evaluation function does not usually arise. In the present experiment, the reinforcement function measures how close the relative phases are to the target relations. In practice, this means that the CPG is started from random initial conditions, and simulated for a fixed amount of time. This time period allows the system to settle down, and then the relative phases are measured over the last complete oscillation in the integration period. The measurement of relative phase was performed in a very simple way, by noting the times at which the oscillators' outputs crossed a threshold; the dynamic range of the oscillators was relatively consistent, so choosing such a threshold was straightforward. Taking oscillator 1 as being the origin, and assuming that all oscillators show the same frequency of oscillation, the output of the system is just the phase of oscillators 2, 3 and 4 relative to 1.

In all these experiments, the integration period was fixed at 3.0 s, which corresponds to 3000 integration steps. The random initial conditions were produced by setting each of the 12 state variables (there are 3 per oscillator) to a random number in the range $[0 : 1]$.

Evaluation is the process of assigning a score to a particular set of values for a, f, k_1 and k_2. The relative phase of the last complete oscillation is taken to be the output of the system. The target gait is described as a set of ideal relative phases, and so the evaluation function is just given by $1 - dist$, where $dist$ is the unsigned distance from the actual to the target phase, summed over oscillators 2, 3 and 4. The evaluation function is then scaled to lie in $[0 : 1]$. Some settings of the parameters are able to completely destroy the oscillations altogether; this is given an evaluation of 0.0.

Where the parameters a, f, k_1 and k_2 are set by a reinforcement agent, this evaluation comprises the training data. This whole process, involving the integration of 12 equations over a few thousand steps, corresponds to one learning epoch. This makes the learning procedure very computationally demanding; a training run of 50 000 iterations takes approximately a day on a 700 series HP workstation.

The evaluation is a function of the parameter settings used, i.e. the outputs of the reinforcement agent. This function is inherently noisy, and is also rather complex. The noise arises from the fact that the particular parameter settings do not uniquely specify the actual gait that will be produced, as there is a dependency on other variables. In the case of converging from random initial conditions to the target gait, the initial conditions themselves partly determine which gait is actually produced, so one combination of parameter values maps onto a set of gaits. The characteristics of this set differ from gait to gait, with some gaits being produced more reliably than others. There is also a dependency on the relative phase of $k_1 sin(k_2 t)$, the periodic component of the driving signal. All of these state-dependent effects are unmodelled by the reinforcement agent, and so they appear as noise added to the evaluation function.

Reinforcement learning is able to optimise noisy evaluation functions, but the evaluation function still needs to be *informative*. We may call an evaluation function informative if it supplies a gradient that informs the controller about how far it is from the solution. An example of a non-informative function would be one in which the goal state corresponded to an evaluation of 1.0, and all other states received 0.0. This kind of optimisation problem is a 'needle in a haystack', as the controller receives no information about the relative worth of any of the states until it reaches the goal, which it must do by a random walk.

Perhaps a worse case would be one in which non-goal states were assigned a variety of evaluations, but not in a way that supplied a gradient towards the goal. This case would be worse because it is likely to produce local minima, which would make the learning performance worse than a random walk.

In the present case, it is not clear whether the evaluation function will be informative or not. Consider the case in which the optimal parameters for a desired gait actually produce a *set* of gaits, with the target appearing so much of the time, and the others the rest of the time. The evaluation of that parameter set is just the expected reinforcement, averaged over the set of produced gaits. One way in which evaluation could be informative is if the frequency of occurrence of the target gait falls off as the parameters move away from their optimal values. This will provide a gradient in the evaluation function, at least in some region around the goal. One possible source of *deception* for learning would be if the incorrect gaits received very low evaluation, but there was some other gait, also incorrect, which would receive a higher evaluation, and could be produced more reliably. This would mean that the parameters that produced this incorrect gait would receive a higher evaluation than that which produces the correct gait. One simple way to reduce this effect is to treat all incorrect gaits equally; this is approximated in the current experiments by raising the evaluation to some positive power; the value 2 was used in these experiments. As this function is bounded by $0, 1$, it's square is a function which gives more discrimination to the high values, while low values are compressed together. Using larger values would tend to introduce the 'needle in a haystack' problem discussed above.

These problems arise because there is no natural distance metric for gaits, and the relative phase relations that are used to measure the gaits may have a very complex relationship with the parameters. One final problem is that coupled oscillators typically take a certain amount of time to settle into a stable phase relation, and this relaxation time also varies between the gaits. All of these factors serve to make this a very hard reinforcement learning problem. On the other hand, the theory describing the behaviour of these systems is very complex, and is generally not informative about which parameters produce which gaits. In this case, some kind of optimisation method is required, and it

Walk	Trot	Bound	
0.29	0.10	0.91	Mean Evaluation
0.33	0.13	0.16	Evaluation s.d.
0.07	0.0	0.82	Percent Correct

FIGURE 6.3. Performance of the parameters from the Collins study on the Walk, the Trot and the Bound. The percent correct scores are given by applying a threshold of 0.9 to the actual relative phase relations; if the relative phases were within 90 percent of those for the ideal gait, the output was counted as being correct

seems reasonable to assume that biological systems are also using optimisation to set parameters. Some of the parameters of biological oscillator systems are hard-wired by evolution, but others are likely to be optimised during the animal's lifetime, to cope with body growth and damage.

6.6 Reinforcement Learning of Gait Production

The performance of the parameters from Collins (1994) on three gaits is shown in table 6.3. These results represent the evaluation of the system as it is started with random initial conditions (each state variable randomised in [0:1]), and run for 3 secs. This process is repeated 1000 times. The percent correct scores are given by an accuracy criterion of 90 percent; if the evaluation is above 0.9, the gait is counted as being correct.

It is immediately clear that there is a large difference between the gaits, with the bound being by far the most reliable, followed by the walk, and then the trot. These performance measures serve to give a baseline against which a reinforcement agent may be judged.

An SRV controller was trained on same task. There are two ways to set up this task, either as an associative learning problem, or a non-associative one. Associative learning tasks are those in which the optimal outputs of the learning system are a function of the inputs. *Non-associative* tasks are those in which the ideal output is given without reference to any input. Most of the work in the reinforcement learning literature is concerned with

associative problems, as the actual reinforcement task is usually quite simple. This means that most of the difficulty of the task arises from elements of pattern recognition on the inputs. If the control problem is very complex, non-associative tasks can be very difficult in their own right.

A non-associative version of the gait production task entails training a controller on just one gait. The controller is a set of SRV units that take no inputs (other than a bias term of 1.0), and give 4 outputs, which are used to set the values of a, f, k_1 and k_2. Three such controllers were initialised with random weights (i.e. random values for both the ϕ and the θ parameters from equations 6.3 and 6.4). The learning rates for these weights were fixed at 0.1, with momentum terms of 0.2 for all the weights. Each controller was trained for 50 000 epochs.

6.6.1 Results

Learning curves for three controllers on the bound, trot and walk are shown in figures 6.4(a), 6.4(b) and 6.4(c) As with the hand-picked parameters, the bound gait is seen to be far easier to learn than the other two. The actual performance of the CPG parameters found by learning is shown in table 6.5. The performance of a controller is given by setting the variance of the output to 0.0, i.e. removing exploration altogether. The mean outputs are then used to set the parameter values in the CPG system, and the parameters are assessed in the same way as the hand-picked values; this involves simulating the CPG system with those parameters for 1000 samples.

6.6.2 Discussion

The poor performance on the walk and trot gaits suggests this is a difficult learning task; only the bound shows any kind of convergence. The difficulty of the control task has several sources; probably the most significant is the complexity of the CPG dynamics. If the CPG output depends sensitively on the values of the parameters, then the learning task for the controller becomes very hard. This is particularly true for this kind of reinforcement agent, where the control outputs are random variables. Such a controller estimates the way expected reinforcement varies with control actions, but the resolution of

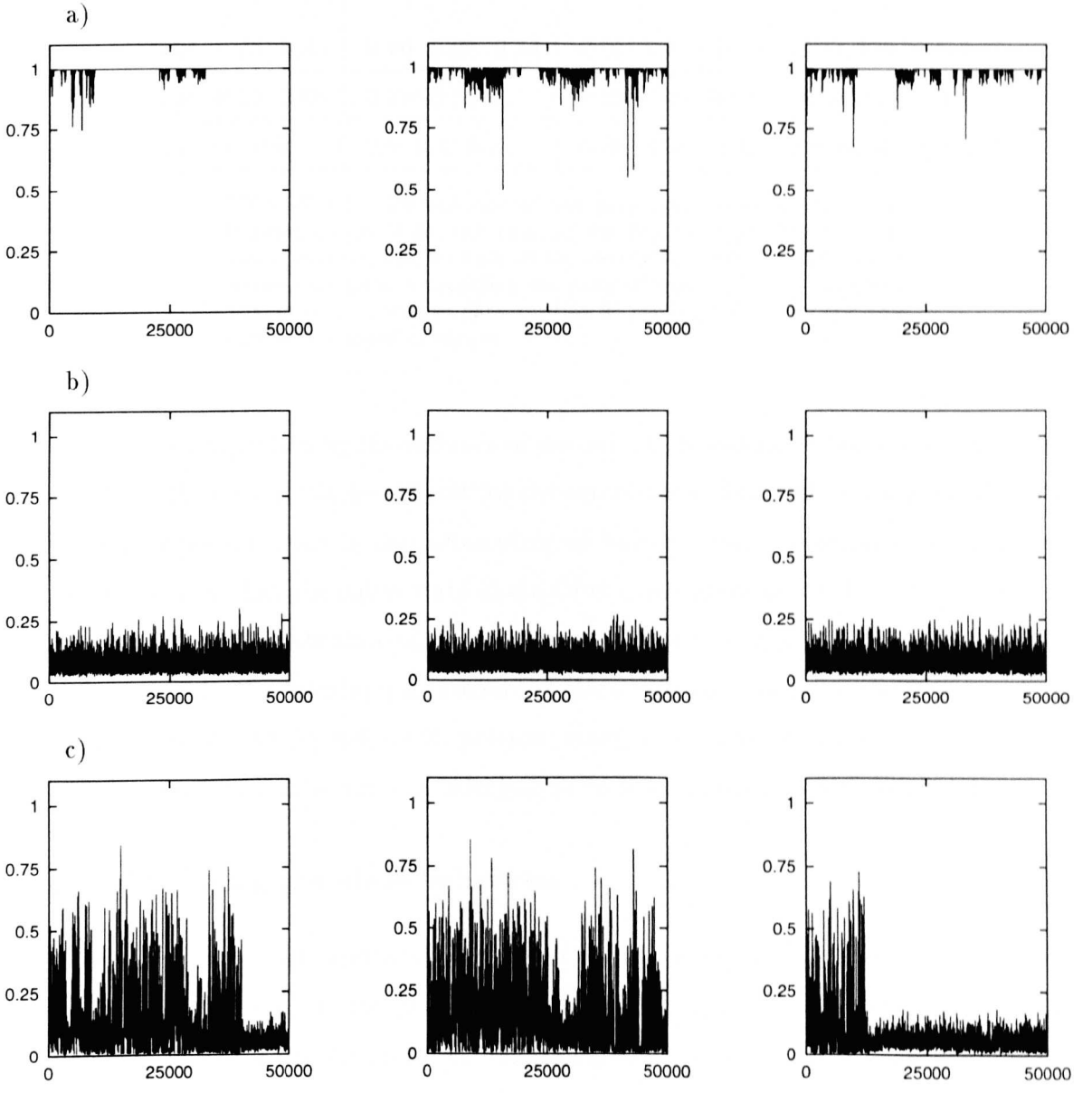

FIGURE 6.4. Learning performance of three SRV controllers on the a) bound, b) trot and c) walk. The axes of all the plots are identical: the y-axis represents reinforcement received, with 1.0 corresponding to perfect performance, and the x-axis denotes learning iteration

Walk	Trot	Bound	
0.04 0.13 0.05	0.16 0.09 0.14	0.95 1.0 1.0	Mean Evaluation
0.06 0.14 0.08	0.19 0.06 0.17	0.05 0.01 0.01	Evaluation s.d.
0.0 0.0 0.0	0.0 0.0 0.0	0.86 0.99 0.99	Percent Correct

FIGURE 6.5. Performance of the parameters from reinforcement learning on the Walk, the Trot and the Bound. Each column shows three numbers, one for each of the controllers. The percent correct scores were given by applying the same criterion of 0.9 to the actual relative phases; relative phases within 90 percent of those for the ideal gait were counted as correct

this estimate is limited by the variance of the output, and so an evaluation which changes rapidly with controls will be difficult for the controller to deal with. As was noted earlier, the reinforcement agent is also attempting to learn a control problem with incomplete information, in that the initial state of the CPG is not given as input.

One way to estimate the complexity of these dynamics is to explicitly model them with a plant model. The learning performance of this model gives some idea of how sensitively the plant behaviour depends on its previous state; an *accurate* model may also be used to supply more informative training information to the controller (see section 1.4.2).

6.7 Modelling the plant behaviour

A key issue with plant models concerns their scale of analysis. In control theory, systems are usually described at the level of the state equations, i.e. at a fine level of detail. However, in motor control problems, modelling may be performed at a variety of scales. Consider again the walking system in (Taga et al., 1991). The dynamics of the combined controller/plant are such that there is a simple relation between the descending control signal (one dimensional) and the speed of movement. This is despite a large variation in the co-ordination patterns of the legs, including a gait transition. In such a case, there seems little point in building a model which deals with the fine scale behaviour of the plant. Modelling at this more abstract level is simpler, but it is also more behaviourally

relevant, since the desired behaviour is more likely to be defined at this level.

This kind of abstract modelling approach is likely to be useful for a range of motor control tasks. However, there are problems with this approach that do not arise in more fine-scale models. The main problem is that the mapping captured by the model may only be well-defined in some parts of the state space. In the walking example above, very large values of the descending control could well destroy the oscillatory activity of the CPGs, in which case the behaviour is stopped completely. In this example, where the control signal is one-dimensional, it is probably straightforward to limit the range of inputs so that this does not happen. In multidimensional cases, this may not be so easy, as there could be sets of values which disrupt the behaviour, and the set of inputs which produce the correct behaviour may have a complex shape. In fact, walking is probably the most difficult of locomotion techniques, because it is relatively easy for the system to lose stability, whereas in flying and swimming supporting the body is more straightforward.

For the coupled oscillator system, a model is built which predicts the relative phases of the outputs. There is a choice to be made about the inputs to the model; the discussion above suggests that a useful set of inputs would be the outputs of the controller, i.e. the parameters a, f, k_1 and k_2. Again, the effects of the initial conditions are treated as noise. However, a baseline performance for such a model has to be a more complete model, which takes *all* variables into account. Building a simplified model only really makes sense if this more complete mapping is able to be approximated; to put this another way, it only makes sense to ignore independent variables in a regression if the regression can be performed with *all* the independent variables. Consequently, the first plant model considered took as input all the information required to predict the behaviour of the plant: the parameters a, f, k_1 and k_2, the initial conditions, and the relative phase of $k_1 sin(k_2 t)$.

The model is trained at the same time as the reinforcement agent, so the data used to train the model is a function of the behaviour of the controller. This means that the controller and the model are tightly coupled in their operation. An accurate model is able to supply more informative training information to the controller, but the controller is (potentially) able to restrict the training data for the model to those areas in which the mapping is relatively simple. This last point arises from the fact that the mapping approx-

imated by the model is closely related to the evaluation function itself. It seems reasonable to assume that maximum reinforcement is received in those areas which reliably give high reinforcement, which are likely to be the areas in which the mapping approximated by the model is simpler.

The model itself is implemented as a feedforward network, with 32 hidden units. The weights were initialised from a normal distribution with mean 1.0 and s.d 0.5. A learning rate of 0.2 was used, with a momentum value of 0.5. The model has a total of 17 inputs: the initial conditions (12 variables), the 4 parameters set by the controller, and the relative phase of $k_1 sin(k_2 t)$. The outputs are just the predicted relative phase of the CPG system, at the end of the integration period.

6.7.1 Results

Learning curves for the model are shown in figs 6.6(a), 6.6(b) and 6.6(c). Somewhat surprisingly, the model is unable to learn the CPG dynamics in the trot and walk gaits.

The learning curves for the model seem to show the same instabilities as those for the reinforcement agent, with positive and negative periods of learning for both systems occurring at the same points. The mapping approximated by the model is deterministic - given the parameters set by the controller, and the set of initial conditions, the relative phases are completely determined. However, the complexity of this mapping varies hugely in different regions of the CPGs parameter space. For some parameter settings, the dependence of relative phase on the initial conditions is quite critical; in other parameter regions, the dependence is relatively insensitive. These factors mean that learning problem for the model is extremely hard, as is the reinforcement task itself.

The fact that the plant model is unable to learn the CPG dynamics (except in the bound) suggests that the poor performance of the controller is due to the complexity of the dynamics, rather than the use of reinforcement learning. It could be the case that other settings of the oscillator's couplings λ_{ij} could be found which gave more tractable dynamics. To investigate this, the controller was given four additional outputs which set the values of the four λ_{ij}.

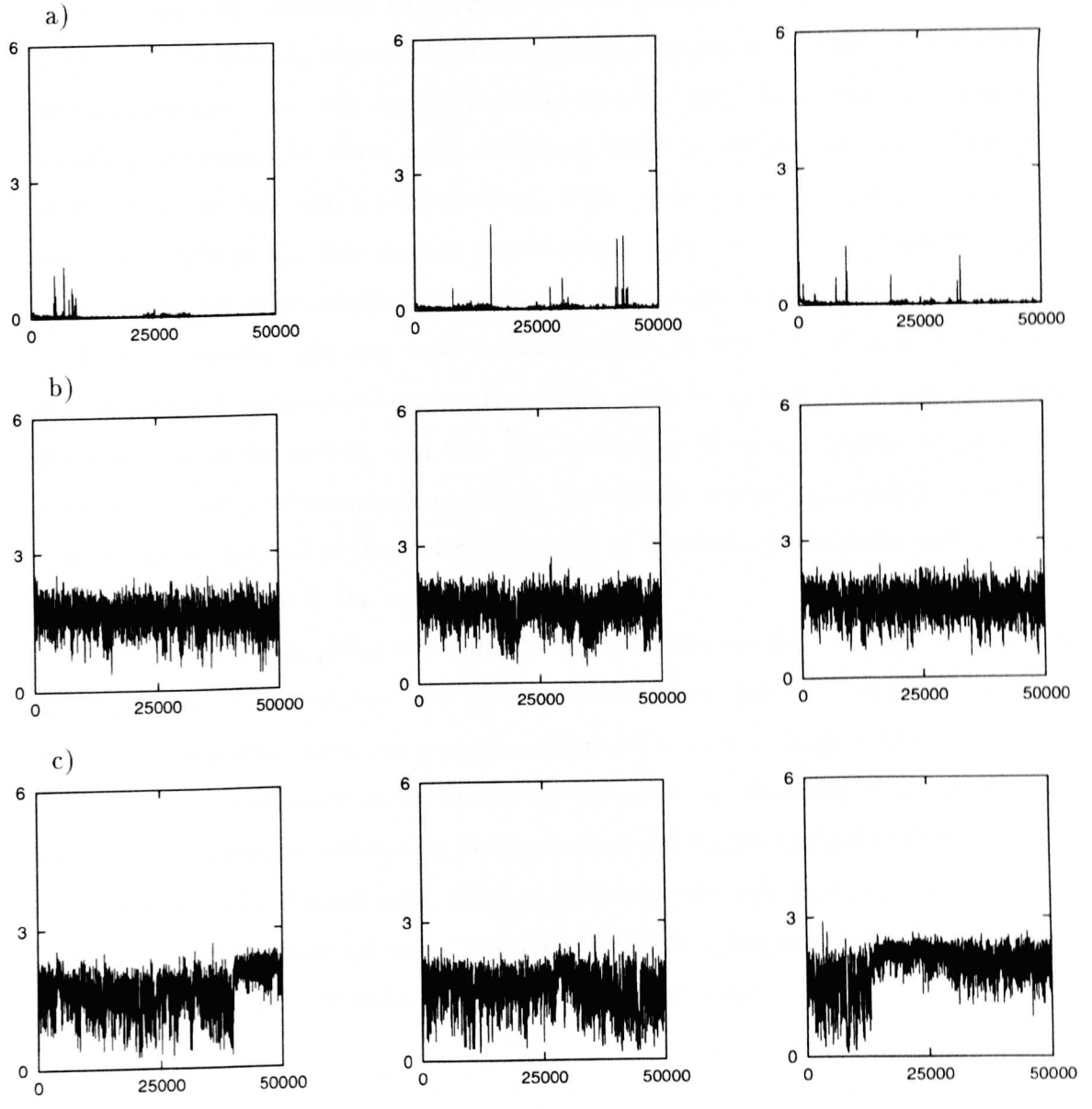

FIGURE 6.6. Learning curves for three plant models on the a) bound, b) trot and c) walk. The y-axis represents RMS error of the model, so perfect prediction now corresponds to a value of 0.0; the x-axis denotes learning iteration. The models are trained concurrently with the reinforcement agents; each model takes as input the initial state of the oscillator system, together with any parameters set by the agent

6.8 Increasing the Power of the Controller

Consider again the equations describing the CPG dynamics, equations 6.2 and 6.1. The CPG system is basically symmetric: if the individual oscillators are started with identical initial conditions, they will track each other exactly. Gait production is a symmetry-breaking phenomena, in which small differences in the oscillators states are exaggerated, until the system falls into a stable pattern. Part of the symmetry in the system comes from the couplings λ_{ji}. One obvious way to enhance the power of the controller is to let the controller set these couplings. Varying the values of the weights breaks the symmetry of the CPG system, and may lead to improvements in learning. It could be the case that quite small differences between the values of the couplings are enough to change the behaviour of the system, and that it is only when the values become *identical* that symmetry breaking becomes such a problem. In this case, asymmetric weights are probably a better approximation to the biological reality, as it would seem unlikely that evolution is able to exactly specify the values of synapses.

On the other hand, giving the learning agent control over the couplings increases the dimensionality of the outputs, and each of those outputs has to be optimised with the single reinforcement value. In general, optimisation from a single evaluative signal is expected to become harder as the output size increases, as estimation of the gradients for all the outputs requires estimating the parameters of a multidimensional distribution.

The 3 gaits were learned again, using an SRV controller with 8 outputs. Again, a plant model was learned simultaneously; this model had four additional inputs, corresponding to the four couplings now under the control of the SRV agent.

6.8.1 Results

The learning curves for these controllers are shown in figures 6.7(a), 6.7(b) and 6.7(c). Table 6.8 gives the performance measures for the final sets of learned parameters. The mean evaluation of the walk has improved, but the correct gait is still never produced. The performance of the bound has actually decreased; having control over the couplings has increased the dimensionality of task without reducing the complexity of the problem. For

the trot gait, the performance has improved greatly. In all three runs, learning converges on values of the couplings that are nearly identical. These consist of two positive values, and two negative.

The learning curves for the plant models are shown in figures 6.9(a), 6.9(b) and 6.9(c). As with the fixed couplings, the model and the controller seem to show related performance; there do not seem to be regions in which the model is able to learn when the controller is not.

These considerations lead us to the following conclusion. Control of this system of coupled oscillators is a difficult task, due to the complexity of the system dynamics. This complexity seems to vary a great deal in parameter space (i.e. the parameters of the CPG); in some regions, the dynamics are relatively simple. These are the regions in which the plant model is able to learn; they are also the regions in which the controller is able to optimise its reinforcement function. In other regions, the dynamics are very complex, and both the controller and the model fail to learn. The reason this is surprising is that the model has access to the entire state of the plant, so it would be expected that its learning could proceed when the controller's could not. That this is not the case suggests that the CPG dynamics are of an all-or-nothing nature; either they are simple, or they are extremely complex.

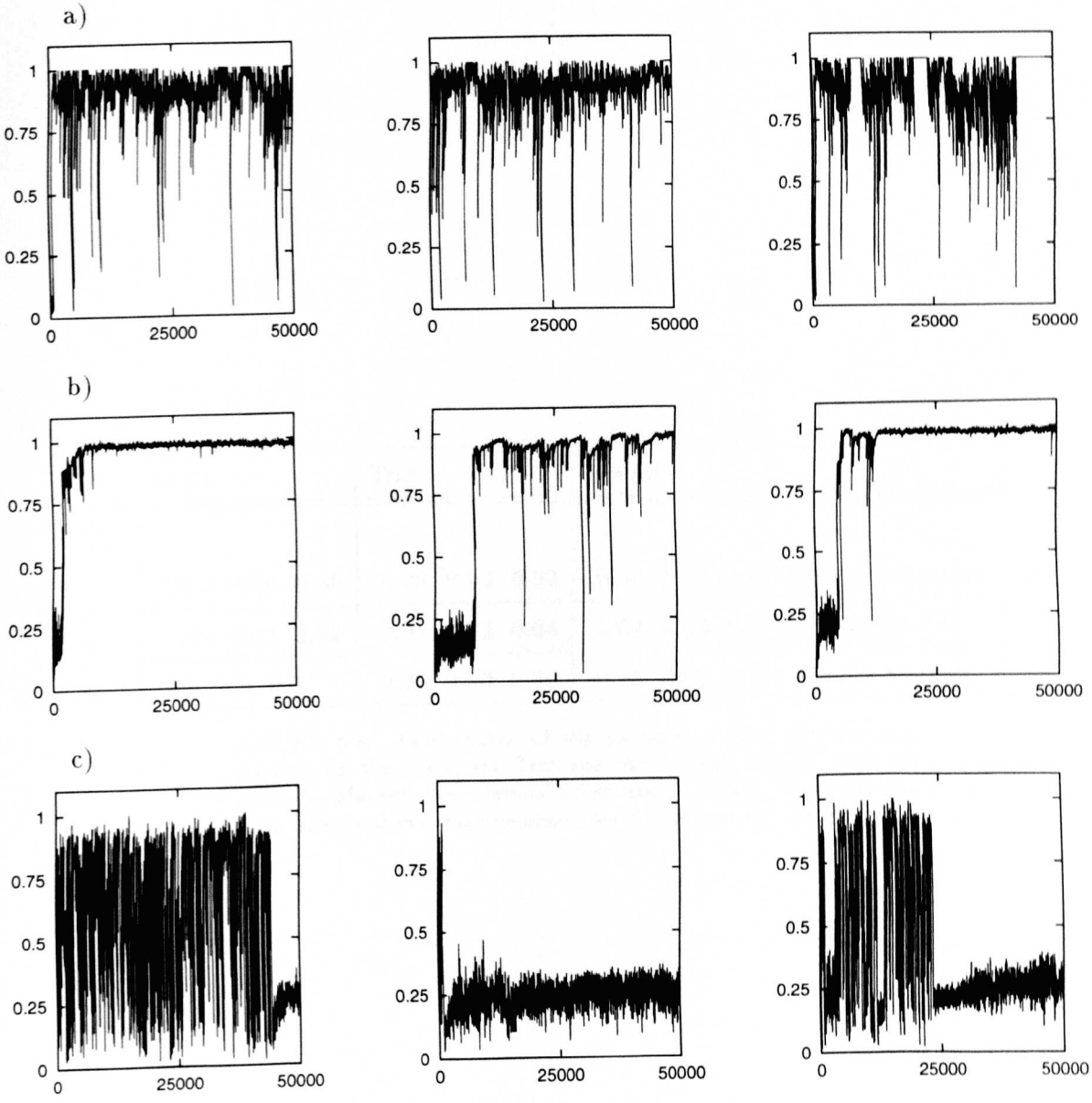

FIGURE 6.7. Learning performance of three SRV controllers on the a) bound, b) trot and c) walk. The y-axis represents reinforcement; the x-axis shows learning iteration. In this case, the reinforcement agent had control over the four couplings in the oscillator system, in addition to the four parameters a, f, k_1 and k_2

Walk	Trot	Bound	
0.33 0.30 0.30	0.98 0.93 0.92	0.86 0.92 1.0	Mean Evaluation
0.02 0.03 0.02	0.04 0.03 0.04	0.08 0.03 0.0	Evaluation s.d.
0.0 0.0 0.0	0.95 0.90 0.86	0.18 0.83 1.0	Percent Correct

FIGURE 6.8. Performance of the parameters from reinforcement learning on the Walk, the Trot and the Bound. In this case, the controller also sets the values of the couplings between the oscillators. Each column shows three numbers, one for each of the controllers

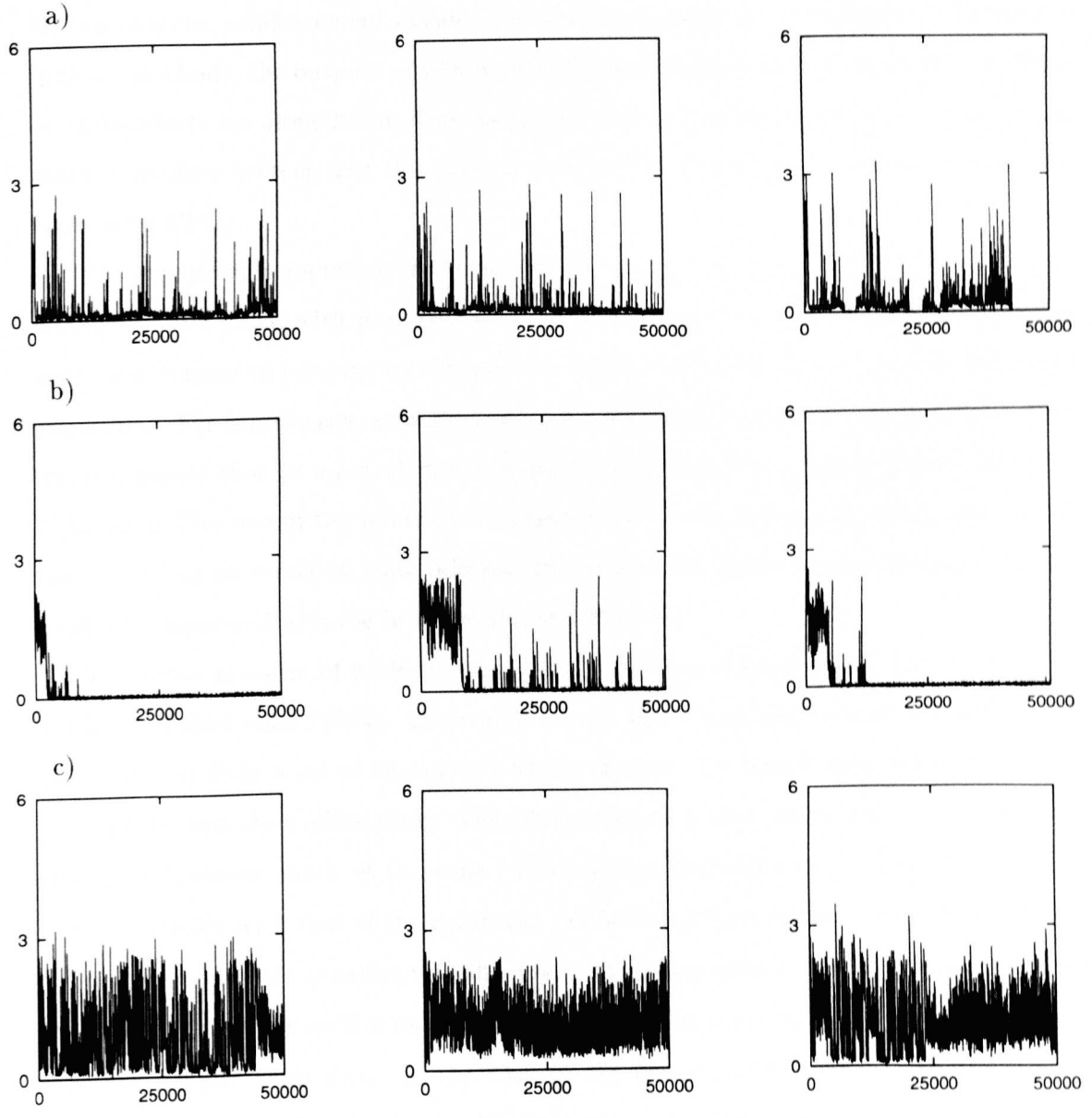

FIGURE 6.9. Learning curves for three plant models on the a) bound, b) trot and c) walk. Plots show RMS error of the model on the y-axis; the x-axis represents learning iteration. The models are trained concurrently with the reinforcement agents; each model takes as input the initial state of the oscillator system, together with any parameters set by the agent. In this case, the controller also set the value of the couplings between oscillators

6.9 Conclusions

In this chapter, reinforcement agents have been considered as controllers. In contrast to indirect methods, the outputs of reinforcement learning agents can have arbitrary effects, as these effects are unmodelled. One particular kind of control action is to set the parameters in another system, and this gives a very flexible way to build control architectures containing CPGs.

The experiments reported here consider the use of a system of coupled nonlinear oscillators as the CPG, with parameters set by the outputs of a reinforcement agent. This agent was trained to produce a set of gaits, or relative phase patterns, from random initial conditions. For comparison, a feedforward net was trained as a model of the CPG dynamics; this model took as input all the information required to completely predict the CPG behaviour. The role of the model is to approximate a well-defined mapping, whereas the controller had no access to state information, and so the effects of the randomised initial conditions appeared as noise in the evaluation function.

A baseline measure of performance is given by a set of hand-picked parameters from Collins and Richmond (1994). The reinforcement agent is unable to learn the walk or the trot gait, but finds a set of parameters which produce the bound more reliably than the parameters from the Collins study. The results also show that the model is unable to learn the CPG dynamics much of the time. The learning performance of the model was also seen to coincide with that of the controller - either they both learned, or neither of them.

A more powerful controller was also used, which was able to modify the values of the couplings between the oscillators. In walk gait, there was a modest increase in performance over the simpler controller. In the case of the bound, performance with the simpler controller was close to optimal, and allowing control over the couplings was seen to reduce performance. In the trot, the performance was greatly improved, and learning converged in all three runs to a set of couplings with two positive values, and two negative.

The fact that the model and controller showed related performance is interesting, particularly in light of the fact that the model had access to complete information about the plant state, and so was approximating a well-defined mapping. The dynamics of the oscillator system seem to vary enormously in complexity over its parameter space, with

some regions showing a simple dependence on the parameters, and the dependency in other regions being critical.

This chapter has considered the use of systems of coupled oscillators as a complex kind of CPG. The particular system chosen is probably not ideal, as the dynamics have been shown to be particularly complex, and other systems may exist which may be driven more easily to the various gaits. Nevertheless, it has been shown that reinforcement learning can be an effective method for setting parameters in complex systems. As a result, a rather simple learning agent may show very complex behaviour, by virtue of the complexity of the CPG dynamics.

7 Conclusions and Discussion

This thesis has been concerned with the use of recurrent neural networks for motor control tasks. Throughout this work, modular networks have been considered, which bear a resemblance to the hierarchical structures seen in biological control systems. The current chapter reviews the results of the experimental studies, and discusses where this work might be taken further.

7.1 Results of Experimental Studies

7.1.1 Recurrent Networks and Motor Pattern Generation

Chapter 4 considered training recurrent networks for generating temporal patterns. Networks were trained on fixed and variable oscillations, and it was seen that learning suffers from instabilities. Learning of a single pattern can be improved through the use of simple methods for setting learning rates for the network parameters. Learning input-dependent patterns is more difficult, and the use of modular architectures was considered. This involved the design of a smaller network, called a CPG by analogy with biological systems, which was subsequently used as a fixed component in a larger network (the other part is called the controller). Related work in the field has typically taken the approach of training a single network on a series of tasks, gradually increasing the task complexity during learning (Giles et al., 1992; Elman, 1993). The work reported here is a different method for achieving the same goal of improving trainability of recurrent nets.

Two methods for building CPGs were discussed: training the CPG on a related, but simpler task, and using a Genetic Algorithm. Both methods were shown to improve learning performance compared to training a single network, with the GA giving the best

performance of all. The goal of the GA was to find networks which maximally improved the learning of another network, so the GA is seen as optimising a complex function of the CPG's weights. It is of interest that this experiment works as well as it does, because it shows how it is possible to optimise networks which function as a part of a larger system.

GAs have been used by other researchers to build neural networks, including recurrent nets (Beer and Gallagher, 1992; Floreano and Mondada, 1995). The work reported here considered the use of GAs to build networks which operate as components within a larger system. This is an interesting optimisation problem, because the GA is required to optimise a network without having the other parts of the system specified. The fact that this experiment worked as well as it did is of interest, because it suggests that GAs are capable of optimising complex functions of a network's weights, and thus able to find structures with rather general properties.

Finally, chapter 4 discussed the use of feedforward networks in feedback loops with recurrent networks. This offers one way of decreasing the complexity of the controller, and changing the relative contributions of the controller and the CPG. The backpropagation procedure for this case was discussed, and this procedure shown to be effective at training the simplified controller on a variable oscillation task. It was also demonstrated that backpropagating through systems with complex dynamics can lead to difficulties in learning, when the system output is far from the target trajectory.

The main conclusion from this chapter is it is straightforward to impose structure on the network architecture, and that this offers a powerful way of constraining the network dynamics. Such constraints can serve to improve the stability and performance of learning.

7.1.2 Learning Control with Motor Programs

In chapter 5, recurrent networks were used as control systems for a simple simulated robot arm. The first set of experiments involved the network making sequential reaching movements, in which the endpoint of the arm was to touch a series of buttons. Modular architectures were again used, with the CPG component built by learning on a simpler task, and by a GA. Recurrent networks are *strongly coupled* systems, in which the activity on one unit is usually strongly dependent on the activity of the others. One consequence of

this is that the system shows a degree of co-articulation by default, which serves to make movements smooth and well coordinated. The use of CPG systems is shown to improve learning performance, at least for these simple tasks.

A further set of experiments used a robot with two arms, which introduces a different kind of redundancy. It was shown that modular architectures enable the different parts of the control task to be de-coupled, so that the controller is able give outputs which are independent of the particular arm to be used. Finally, control of a dynamic robot arm was considered, and it was seen that a low-level feedback controller may be driven with a higher open-loop system, much as biological muscles implement simple feedback loops.

Other reseachers have investigated the use of motor programs for motor control problems, but these have usually been static structures, which have been hard-wired in advance. For example, the work of Lipitkas *et al* (Lipitkas et al., 1992; Lipitkas et al., 1993; Bock et al., 1993; Bock et al., 1996) implements the motor programs as waveform generators, which map a static representation of the movement into a time series of torques for the arm. The approach taken here is more general, in that the controller may modify the activity of the motor program during the movement, and it is also straightforward to implement feedback control in this scheme. The cost of this increase in power is that recurrent backpropagation is used, a more complex form of learning than the feedforward learning rules used in the work of Lipitkas *et al.*

7.1.3 Direct Motor Control and Reinforcement Learning

Chapter 6 considered the use of reinforcement learning agents for motor control tasks. Reinforcement learning often requires very long training times, due to the lack of gradient information. One way to approach the learning of difficult control problems is to enhance the complexity of the controller's actions; this is the general role of the CPG systems discussed throughout the thesis. One benefit of using reinforcement learning over supervised techniques is that the controller's outputs can have arbitrary effects, as these effects are never modelled. In chapter 6, controllers are used whose actions set the parameters of a system of coupled oscillators; the controller is then trained to drive the oscillator system to a set of gait patterns.

The paricular system studied is a set of coupled oscillators proposed by other researchers as a possible mechanism for the generation of quadrupedal gaits (Collins and Richmond, 1994; Yuasa and Ito, 1990). In (Collins and Richmond, 1994), the free parameters of the model were chosen by hand, and the resultant system seen to be able to generate plausible gaits and gait transitions. Chapter 6 considers a learning agent which sets the values of these free parameters, and uses a measure of the correctness of the resulting gait to improve its performance. On the one hand, this can be seen as an attempt to automate part of the modelling procedure, when there is no clear way to use a regression technique. However, this work is also an investigation into learning in reinforcement agents with motor programs, which is closely related to structural shaping methods, and 'chunking' in A.I. architectures (Iba, 1989).

The learning performance on this task is shown to be rather poor, with only one gait being produced reliably. The reinforcement agent is attempting to control a system with complex dynamics, and the complexity of those dynamics was investigated through the use of a plant model. This model was trained with complete information about the state of the oscillator system, so the learning task is to approximate a well-defined function. Despite this fact, the model also shows poor performance, and learns successfully only in those situations that the controller also learns.

A second set of experiments involved increasing the power of the controller, setting the values of the couplings between the oscillators by the controller's outputs. This is seen to have a mixed effect on the performance: the trot gait, which was not learned at all in the first experiment, was seen to be produced reliably. However, in the case of the bound, which was produced nearly perfectly in the previous experiment, the performance deteriorated.

As with the first experiment, a plant model was trained concurrently with the controller, and again, the model and controller showed related performance. The fact that the model is only able to learn at the same points as the controller is interesting, and suggests that the complexity of the plant dynamics varies widely in parameter space. The dynamics are relatively simple in some regions, and both the controller and the plant are able to optimise their behaviour. In other regions, the dynamics are complex, and neither

system is able to learn.

The main conclusion from this chapter is that the concept of a motor program, or a CPG system may have particular utility for reinforcement learning, in which difficult control problems may result in unacceptably long training times. CPGs have the effect of increasing the complexity of the controller's actions, and the model-free nature of reinforcement learning means that those actions may have arbitrary effects. This makes it possible to design simple controllers which produce complex behaviours, a fact which has particular significance for the learning of motor control.

7.2 Suggestion for Future Research

There are a number of directions in which this research might be extended. The CPGs used in the present thesis have a variety of sources, from learning, Genetic Algorithms, and mathematical models of gaits. A fruitful line of research might be to use data from physiological studies of invertebrate circuits to build these systems. *These* CPGs could then be used as components in a larger system, which is tested on complete motor tasks. The attraction of such an approach is that it gives an alternative way to try to understand the computational role of different motor circuits. There is a wealth of data on invertebrate motor circuits, but its interpretation is problematic because there is no clear relation to the computational demands of the tasks which animals have to solve.

Computational studies offer a different approach to this problem, and CPGs give a way of including knowledge of the physiology in an artificial system, and may help us understand the interaction between the motor circuits, and the rest of the animal's nervous system and body. In other words, this gives a way of moving from 'fictive' motor patterns, to systems producing adaptive behaviour.

The experiments in chapter 5 were concerned with very simple tasks, and used a simple simulated robot. It would be very interesting to see if these ideas are useful for the control of real robots, which would involve dealing with many difficult problems not addressed here. Perhaps the most interesting problem would be the control of walking systems, which seems to involve very delicate co-ordination of pattern generating elements and sensory feedback. The use of modular architectures seems a promising approach for

such problems, and gives a way of incorporating varying amounts of prior knowledge, and having learning occurring at selected points in a system.

References

Alexander, G., Delong, M., and Crutcher, M. (1992). Do cortical and basal ganglionic motor areas use motor programs to control movement. *Behavioral and Brain Sciences*, **15**(4):656–665.

Baldwin, J. (1896). A new factor in evolution. *American Naturalist*, **30**(1):441–451.

Barnett, S. and Cameron, R. (1985). *Introduction to Mathematical Control Theory*. OUP, London.

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*, **13**:834–846.

Barto, A., Sutton, R., and Watkins, C. (1989). Learning and sequential decision making. Technical Report 89-95, COINS, Uni. of Massachusetts, Amherst.

Beer, R., Chiel, H., Quinn, R., Espenschied, K., and Larsson, P. (1992). A distributed neural network architecture for hexapod robot locomotion. *Neural Computation*, **4**(3):356–365.

Beer, R. and Gallagher, J. (1992). Evolving dynamical neural networks for adaptive behaviour. *Adaptive Behaviour*, **1**(1):91–122.

Bernstein, N. (1967). *The co-ordination and regulation of movements*. Pergamon Press, London.

Berthier, N., Singh, S., Barto, A., and Houk, J. (1992). A cortico-cerebellar model that learns to generate distributed motor commands to control a kinematic arm. In Moody,

J., editor, *Advances in Neural Information Processing Systems*, volume 4, pages 611–618, San Mateo, CA. Morgan Kaufmann.

Berthier, N., Singh, S., Barto, A., and Houk, J. (1993). Distributed representation of limb motor programs in arrays of adjustable pattern generators. *Journal of Cognitive Neuroscience*, **5**:56–78.

Bizzi, E., Hogan, N Mussa-Ivaldi, F., and Giszter, S. (1992). Does the nervous system use equilibrium point control to guide single and multiple joint movements? *Behavioural and Brain Sciences*, **15**:603–613.

Bizzi, E. and Mussa-Invaldi, A. (1990). Muscle properties and the control of arm movements. In Osherson, N., Kosslyn, M., and Hollerbach, J., editors, *An Invitation To Cognitive Science: Visual Cognition and Action*, volume 2, pages 213–243. MIT Press.

Bock, O., D'Eleuterio, G., Lipitkas, J., and Grodski, J. (1993). Parametric motion control of robotic arms: A biologically based approach using neural networks. *Telematics and Informatics*, **10**:179–185.

Bock, O., D'Eleuterio, G., Lipitkas, J., and Grodski, J. (1996). Parametric motor control: a new approach to the control of point to point manipulator movements. *Journal of Robotic Systems*, **13**:35–40.

Brooks, R. (1991). Intelligence without representation. *Artificial Intelligence*, **47**:139–159.

Buchanan, J. Locomotion in a lower vertebrate: Studies of the cellular basis of rhythmogenesis and oscillator coupling. In *Advances in Neural Information Processing Systems*, pages 101–108. MIT Press.

Chen, S. and Billings, S. (1992). Neural networks for nonlinear dynamic system modelling and identification. *International Journal of Control*, **56**:319–346.

Clark, A. (1994). Autonomous agents and real-time success: Some foundational issues. Technical Report 94-08, Washington University: Philosophy Program.

Cohen, A., Rossignol, S., and Grillner, S. (1988). *Neural Control of Rhythmic Movements in Vertebrates*. Wiley, New York.

Collins, J. and Richmond, S. (1994). Hard-wired central pattern generators for quadrupedal locomotion. *Biological Cybernetics*, **71**:375–385.

Dayan, P. and Sejnowski, T. (1993). Td (λ) converges with probability 1. *Machine Learning*, **7**:295–301.

Delcomyn, F. (1980). Neural basis of rhythmic behaviour in animals. *Science*, **210**:492–497.

Doya, K. (1992). Bifurcations in the learning of recurrent neural networks. In *Proceedings of 1992 IEEE International Symposium on Circuits and Systems*, pages 2777–2780.

Doya, K. (1993). Universality of fully connected recurrent neural networks. *submitted to IEEE Transactions on Neural Networks.*

Doya, K., Boyle, M., and AIS, S. (1993). Mapping between neural and physical activities of the lobster gastric mill. In Giles, C., Hanson, S., and Cowan, J., editors, *Advances in Neural Information Processing Systems 5*, pages 913–920, San Mateo, CA. Morgan Kaufmann.

Doya, K. and Yoshizawa, S. (1990). Adaptive synchronisation of neural and physical oscillators. In Touretzky, D., editor, *Advances in Neural Information Processing Systems 2*, pages 147–148, Hillsdale, NJ. Morgan Kaufmann.

Eckmiller, R. (1989). Generation of movement trajectories in primates and robots. *Neural Computing Architectures.*

Ekeberg, O. (1993). A combined neuronal and mechanical model of fish swimming. *Biological Cybernetics*, **69**:363–374.

Elman, J. (1990). Finding structure in time. *Cognitive Science*, **14**:179–211.

Elman, J. (1993). Learning and development in neural networks: the importance of starting small. *Cognition*, **48**:71–99.

Elman, J. and Zipser, D. (1988). Learning the hidden structure of speech. *Journal of the Acoustical Society of America*, **83**.

Fang, Y. and Sejnowski, T. (1990). Faster learning for dynamic recurrent backpropagation. *Neural Computation*, **2**:270–273.

Floreano, D. and Mondada, F. (1995). From evolution of innate behaviours to evolution of learning in robotic agents. *to appear in Adaptive Behaviour.*

G, D. and Kelso, J. (1990). Multifrequency behavioural patterns and the phase attractive circle map. *Biological Cybernetics*, **64**:485–495.

Gallistel, C. (1980). *The Organisation of Action: a New Synthesis.* Erlbaum, Hillsdale, NJ.

Georgopoulos, A., Schwartz, A., and Kettner, R. (1986). Neuronal population coding of movement direction. *Science*, **233**:1416–1419.

Georgopoulos, A., Taira, M., and Lukashin, A. (1993). Cognitive neurophysiology of the motor cortex. *Science*, **260**:47–52.

Getting, P. and Dekin, M. (1985). Tritonia swimming: A model system for integration within rhythmic motor systems. In Selverston, A., editor, *Model Neural Networks and Behaviour*, New York. Plenum Press.

Giles, L., Miller, C., Cheng, D., Chen, H., Sun, G., and Lee, Y. (1992). Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, **4**:393–405.

Gomi, H. and Kawato, M. (1993). Neural network control for a closed-loop system using feedback-error-learning. *Neural Networks*, **6**(7):933–946.

Gorinevsky, D. (1993). Modelling of direct motor program learning in fast human arm motions. *Biological Cybernetics*, **69**:219–228.

Grillner, S. (1985). Neurobiological basis of rhythmic motor acts in vertebrates. *Science*, **228**:143–149.

Grillner, S. and Wallen, P. (1984). How does the lamprey nervous system make the lamprey swim? *Journal of Experimental Biology*, **112**:337–357.

Grossberg, S. (1986). *The Adaptive Brain:II Vision, Speech, Language and Motor Control.*

Gullapalli, V. (1990). A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural Networks*, **3**:671–692.

Gullapalli, V. (1991). A comparison of supervised and reinforcement learning methods on a reinforcement learning task. In *Proc of the 1991 IEEE International Symposium on Intelligent Control.*

Gullapalli, V. (1992). *PhD Thesis: Reinforcement Learning and its Application to Control.* University of Massachusetts, Amherst.

Gullapalli, V., Grupen, R., and Barto, A. (1992). Learning reactive admittance control. In *Proc of the 1992 IEEE Int Conf on Robotics and Automation*, pages 1475–1480, Nice, France. IEEE.

Haken, H. (1983). *Synergetics - an Introduction.* Springer-Verlag, Heidelberg.

Heinzel, H. and Selverston, A. (1988). Gastric mill activity in the lobster. *Journal of Neurophysiology*, **59**:566–585.

Hertz, J., Krogh, A., and Palmer, R. (1991). *Introduction to the theory of Neural Computation.* Addison Wesley, Redwood City, CA.

Hinton, G. and Nowlan, S. (1987). How learning can guide evolution. *Complex Systems*, 1:495–502.

Hopfield, J. and Tank, D. (1985). Neural computation of decisions in optimization problems. *Biological Cybernetics*, **52.**

Houk, J., Barto, A., Eisenman, L., Keifer, J., Singh, S., Sinkjaer, T., and Vyas, D. (1990). Motor programs and sensorimotor integration. In Caudill, M., editor, *Proceedings of the International Joint Conference on Neural Networks*, pages 147–148, Hillsdale, NJ. Lauwrence Elbaum.

Husbands, P., Harvey, I., and Cliff, D. (1993). An evolutionary approach to situated ai. In Sloman, A., Hogg, D., Humphreys, D., Ramsay, A., and Patridge, D., editors,

AISB93: Proceedings of the 9th Biannual Conference of the Society for the study of Artificial Intelligence and the Simulation of Adaptive Behaviour., pages 61–70, Amsterdam. A.I.S.B., IOS Press.

Iba, G. (1989). A heuristic approach to the discovery of macro operators. *Machine Learning*, **3**:285–317.

Jordan, M. (1988). Supervised learning and systems with excess degrees of freedom. Technical Report 8827, COINS, MIT.

Jordan, M. and Jacobs, R. (1990). Learning to control an unstable system with forwards modelling. In Touretzky, D., editor, *Advances in Neural Information Processing Systems 2*, pages 147–148, Hillsdale, NJ. Morgan Kaufmann.

Jordan, M. and Rumelhart, D. (1992). Forward models: supervised learning with a distal teacher. *Cognitive Science*, **16**:307–354.

Kawato, M., Furukawa, K., and Suzuki, R. (1987). A hierarchical neural network model for control and learning of voluntary movements. *Biological Cybernetics*, **57**:169–185.

Kelso, J., Scholz, J., and Schoner, G. (1988). Dynamics governs switching among patterns of co-ordination in biological movement. *Physics Letters A*, **134**(1):8–12.

Kelso, J. and Schoner, G. (1988). Self-organisation of coordinative movement patterns. *Human Movement Science*, **7**:27–46.

Kumaresan, M. and Sharkey, N. (1993). Generalisation and extension of motor programs for a sequential recurrent network. In Bekey, G. and Goldberg, K., editors, *Neural Networks in Robotics*. Kluwer Academic Publishers.

Kuperstein, M. (1988). Neural model of adaptive hand-eye coordination for single postures. *Science*, **239**:1308–1311.

Kuperstein, M. (1992). Adaptive sensory-motor coordination through self-consistency. In Weschler, H., editor, *Neural Networks for Perception*, volume 1, pages 285–314, New York. Academic Press.

Lane, S., Handelman, D., and Gelfand, J. (1993). Modulation of robotic motor synergies using reinforcement learning optimisation. In Bekey, G. and Goldberg, K., editors, *Neural Networks in Robotics.* Kluwer Academic Publishers.

Lipitkas, J., Bock, O., D'Eleuterio, G., and Grodski, J. (1992). Parametric control of point to point robotic arm movements using neural networks. In *Proc 7th CASI Conf on Astronautics*, pages 167–173, Ottawa, Can.

Lipitkas, J., D'Eleuterio, G., Bock, O., and Grodski, J. (1993). Reinforcement learning and the parametric motor control hypothesis applied to robotic arm movements. In *Proceedings of the Knowledge-Based Systems and Robotics Workshop*, pages 101–105, Gloucester, Ont. Business Intelligence Systems, Academic Press.

Marder, E. (1988). Modulating a neuronal network. *Nature*, **335**:296–297.

Marr, D. (1977). Artificial intelligence - a personal view. *Artificial Intelligence*, **9**:37–48.

Massone, L. (1994). A neural network system for control of eye movements - basic mechanisms. *Biological Cybernetics*, **71**:293–305.

Massone, L. and Myers, J. (1996). The role of plant properties in arm trajectory formation - a neural network study. *IEEE Transactions on Systems, Man and Cybernetics*, **5**:719–732.

Meeden, L. (1984). *PhD Thesis: Towards Planning: Incremental Investigations into Adaptive Robot Control.* Indiana University.

Miller, P. (1997). Recurrent neural networks and motor programs. In *to appear in Proc of the 1997 European Symposium on Artificial Neural Networks.*

Miller, W., Sutton, R., and Werbos, P. (1990). *Neural Networks and Control.* MIT Press, Cambridge.

Narendra, K. and Parthasarathy, K. (1991). Gradient methods for the optimisation of dynamical systems containing neural networks. *IEEE Transactions on Neural Networks*, **2**:252–262.

Nolfi, S., Elman, J., and Parisi, D. (1994). Learning and evolution in neural networks. *Adaptive Behaviour*, **3**:5–28.

Oku, Y., Dick, T., and Cherniack, N. (1993). Phase-dependent dynamic responses of respiratory motor activities following perturbation of the cycle in the cat. *Journal of Physiology*, **461**:321–337.

Parisi, D., Cecconi, F., and Nolfi, S. (1990). Econets: Neural networks that learn in an environment. *Network*, **1**:149–168.

Pearlmutter, B. (1990). Learning state space trajectories in recurrent neural nets. *Neural Computation*, **1**:263–269.

Pearson, K. (1993). Common principles of motor control in vertebrates and invertevbates. *Annual Review of Neuroscience*, **6**:65–297.

Pham, D. and Lin, X. (1993). Identification of linear and nonlinear dynamic systems using recurrent neural networks. *Artificial Intelligence in Engineering*, **8**:67–75.

Pineda, F. (1987). Generalisation of backpropagation to recurrent neural networks. *Physical Review Letters*, **19(59)**:2229–2232.

Poggio, T., Torre, V., and Koch, C. (1985). Computational vision and regularisation theory. *Nature*, **317**:314–319.

Qin, S., Su, H., and McAvoy, T. (1992). Comparison of 4 neural net learning methods for dynamic system identification. *IEEE Transactions on Neural Networks*, **3**:122–130.

Ross, S. (1983). *Introduction to Stochastic Dynamic Programming.* Academic Press, New York.

Rowat, P. and Selverston, A. (1991). Learning algorithms for oscillatory networks with gap junctions and membrane currents. *Network*, **2**:17–41.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning internal representations by error backpropagation. In *Parallel Distributed Processing*, volume 1, Cambridge, MA. MIT Press.

Schoner, G., Jiang, W., and Kelso, J. (1990). A synergetic theory of quadrapedal gaits and gait transitions. *Journal of Theoretical Biology*, **142**:359–391.

Schoner, G. and Kelso, J. (1988a). A synergetic theory of environmentally specified and learned patterns of movement coordination: I relative phase dynamics. *Biological Cybernetics*, **58**:71–80.

Schoner, G. and Kelso, J. (1988b). A synergetic theory of environmentally specified and learned patterns of movement coordination: I.i. component oscillator dynamics. *Biological Cybernetics*, **58**:81–89.

Schoner, G Zanone, P. and Kelso, J. (1992). Learning as change of co-ordination dynamics: Theory and experiment. *Journal of Motor Behaviour*, **24**:29–48.

Selverston, A. (1985). *Model Neural Networks and Behaviour.* Plenum Press, New York.

Selverston, A. (1994). Modelling of neural circuits: What have we learned? *Annual Review of Neuroscience*, **16**:531–546.

Shadmehr, R., Brasher-Krug, T., and Mussa-Ivaldi, F. (1995). Interference in learning internal models of inverse dynamics in humans. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems*, volume 7, Cambridge, MA. MIT Press.

Shadmehr, R. and Mussa-Ivaldi, F. (1994). Geometric structure of the adaptive controller of the human arm. Technical Report AI Memo 1437, Massachusetts Institute of Technology, AI Lab.

Tabor, M. (1989). *Chaos in Integrability and Nonlinear Dynamics, An Introduction.* John Wiley and Sons, New York.

Taga, G., Yamaguchi, Y., and Shimizu, H. (1991). Self-organised control of bipedal locomotion by neural oscillators in unpredictable environment. *Biological Cybernetics*, **65**:147–159.

Thrun, S. (1992). Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, Carnegie Mellon University.

Thrun, S. and Moller, K. (1992). Active exploration in dynamic environments. In Moody, J., Hanson, S., and Lippman, R., editors, *Advances in Neural Information Processing Systems*, volume 4, San Mateo, CA. Morgan Kaufmann.

Thrun, S., Moller, K., and Linden, A. (1991). Planning with an adaptive world model. In Touretzky, D. and Lippmann, R., editors, *Advances in Neural Information Processing Systems*, volume 3, San Mateo, CA. Morgan Kaufmann.

Tsung, F.-S. and Cottrell, G. (1995). Phase space learning. In Tesauro, D., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems*, volume 7. MIT Press.

Tsung, F.-S., Cottrell, G., and Selverston, A. (1990). Some experiments on learning stable network oscillations. In *Proc of International Joint Conf on Neural Networks*, volume 1, pages 169–174.

van Soest, A. and Bobbert, M. (1993). The contribution of muscle properties in the control of explosive movements. *Biological Cybernetics*, **69**:195–204.

Verhulst, F. (1989). *Nonlinear Differential Equations and Dynamical Systems.* Springer-Verlag, Berlin.

Watkins, C. (1989). *PhD Thesis: Learning from Delayed Rewards.* University of Edinburgh.

White, D. and Sofge, S. (1992). *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches.* Van Nostrand Reinhold, Florence, Kentucky.

Widrow, B. and Stearns, S. (1985). *Adaptive Signal Processing.* Prentice Hall, Eaglewood Cliffs, NJ.

Williams, R. and Zipser, D. (1989). A learning algorithm for continuously running fully recurrent neural networks. *Neural Computation*, **1**:270–280.

Williams, T. and Sigvardt, K. (1992). How the lamprey swims. *News in Physiological Sciences*, **7**:161–165.

Wolpert, D., Ghahramani, Z., and Jordan, M. (1995). Forwards dynamics models in human motor control: Psychophysical evidence. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems*, volume 7, Cambridge, MA. MIT Press.

Yamada, T. and Yabuta, T. (1993). Dynamic system identification using neural networks. *IEEE Transaction on Systems, Man and Cybernetics*, **23**:204–211.

Yuasa, H. and Ito, M. (1990). Coordination of many oscillators and generation of locomotory patterns. *Biological Cybernetics*, **63**:177–184.