Adaptive optimal-control algorithms
for brainlike networks

by

Lakshminarayan Chinta Venkateswararao

A thesis submitted in conformity with the requirements
For the degree of Doctor of Philosophy
Graduate Department of Physiology
University of Toronto

© Copyright Lakshminarayan Chinta Venkateswararao 2010
Adaptive optimal-control algorithms for brainlike networks

Lakshminarayan Chinta Venkateswararao

Doctor of Philosophy

Graduate department of Physiology
University of Toronto
2010

Abstract

Many neural control systems are at least roughly optimized, but how is optimal control learned in the brain? There are algorithms for this purpose, but in their present forms they aren’t suited for biological neural networks because they rely on a type of communication that isn’t available in the brain, namely weight transport — transmitting the strengths, or “weights”, of individual synapses to other synapses and neurons. Here I show how optimal control can be learned without weight transport.

I explore three complementary approaches. In the first, I show that the control-theory concept of feedback linearization can form the basis for a simple mechanism that learns roughly optimal control, at least in some sensorimotor tasks. Second, I describe a method based on Pontryagin’s Minimum Principle of optimal control, by which a network without weight transport might achieve optimal open-loop control. Third, I describe a mechanism for building optimal feedback controllers, without weight transport, by a method based on generalized Hamilton-Jacobi-Bellman equations. Finally, I argue that the issues raised in these three projects apply quite broadly, i.e. most control algorithms
rely on weight transport in many different ways, but it may be possible to recast them into forms that are free of such transport by the mechanisms I propose.
Acknowledgements

I express my sincere gratitude to my supervisor Dr. Douglas Tweed for his endless guidance, support, patience and encouragement during my doctoral research and the completion of this dissertation.

I thank Dr. F. Ben Amara and Dr. W.D. Hutchison for their efforts as associate advisors on my thesis committee. The support and encouragement of my colleagues and friends: Mohamed Abdelghani, Kristen Fortney, Maryam Fesharaki, Tim Lillicrap, and Danitza Goche Montes and are gratefully acknowledged. Kristen provided the weightless-learning equations in Section 3.3.2. The analysis of the feasibility of learning by weight transport in Section 2.4 was joint work with Mohamed. I would especially like to thank Mohamed for his support and companionship during my doctoral program.

To my family, more than anyone else, I owe this small moment of triumph. I dedicate this dissertation to my grandparents, Shri G. Sadagopan & Smt S. Suseela, my mother, Dr. C. Sridevi and my aunt, Ms S. Vijayalakshmi. I would not be what I am today without them. I would like to thank my wife, Meena for her boundless love and faith in my abilities. Special thanks to my brothers, Deepu and Yogesh for their support.

I would also like to acknowledge the support of Canadian Institutes of Health Research for this work.
# Table of Contents

CHAPTER 1

SCOPE OF THE THESIS 1

1.1 LEARNING OPTIMAL TRAJECTORIES 2

1.2 BIOLOGICAL CONSTRAINTS ON ALGORITHMS 4

1.3 OUTLINE OF THE THESIS 6

CHAPTER 2

GENERAL BACKGROUND 7

2.1 NEURAL COMPUTATION 7

2.2 ADAPTIVE CONTROL 13

2.3 OPTIMAL CONTROL 21

2.4 COMMUNICATION IN BIOLOGICAL NETWORKS 30

2.5 ALGORITHMS FOR SENSORIMOTOR CONTROL 44
<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.2</td>
<td>THE METHOD OF GENERALIZED HAMILTON-JACOBI-BELLMAN EQUATIONS</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>GHJB WITHOUT WEIGHT TRANSPORT</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>5.4</td>
<td>SCOPE OF THE ALGORITHM</td>
<td>128</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GENERAL CONCLUSIONS AND SIGNIFICANCE</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>VARIATIONS</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>ROLE OF THE TRANSPOSE</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>HOW MANY NEURONS DO WE NEED?</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>6.4</td>
<td>CONCLUSION</td>
<td>149</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1: The point neuron 10
Figure 2.2: Network of point neurons 12
Figure 2.3: A biological example of a controlled process or plant: an arm reaching to a target 14
Figure 2.4: A simple example of the need for foresight in control 23
Figure 2.5: Without foresight, learning fails in the simple elbow flexion 24
Figure 2.6: Retrograde cytoplasmic transport in backpropagation algorithm 40
Figure 2.7: Linear-in-the-parameters learning 43
Figure 2.8: Forward internal model 45
Figure 2.9: An inverse model is a type of controller 47
Figure 2.10: Direct inverse model 48
Figure 2.11: Distal teacher learning 50
Figure 2.12: Feedback error learning 53
Figure 2.13: A controller for the vestibulo-ocular reflex, or VOR, learns its task when the sign of its innate estimate of sensitivity derivatives is accurate 56
Figure 2.14: The VOR controller fails to learn its task when the sign of the sensitivity derivative matrix changes midway through the simulation 57
Figure 2.15: Implicit supervision 60
Figure 3.1: Elbow flexion controller trained with Hurwitz error 70
Figure 3.2: Elbow flexion controller trained with nonlinear differential error 75
Figure 3.3: Inverted pendulum task 77
Figure 3.4: Inverted pendulum parameters 79
Figure 3.5: Trained with the energy-based differential error, a controller learns to swing
the pendulum to the upright position 81
Figure 3.6: With weightless learning and the hybrid differential error, a controller quickly
learns to swing a pendulum up to the zenith and hold it there 86
Figure 3.7: With weightless learning and the hybrid differential error, a controller learns
the required number of backswings when the maximum allowable command is altered 88
Figure 4.1: Sweep method implemented in a neural network without weight transport 94
Figure 4.2: The sweep network learns the appropriate biphasic commands for a quick
elevation 98
Figure 5.1: Network implementation of the GHJB algorithm 116
Figure 5.2: Learning the transpose of s synaptic weight matrix 122
Figure 5.3: Bipartite synapses might reduce the number of neurons for the GHJB
algorithm 124
Figure 5.4: GHJB algorithm learns to drive a linear second-order plant to a zero target
minimizing a quadratic loss function 130
Figure 5.5: GHJB algorithm learns roughly time-optimal control for a linear second-order
plant 133
Figure 5.6: GHJB algorithm learns to drive a linear second-order plant to nonzero targets 135
Figure 5.7: GHJB algorithm learns to drive a linear second-order plant to 0 so as to minimize the integral of a time-dependent loss

Figure 5.8: GHJB algorithm learns a task with kinematic redundancy
List of Appendices

APPENDIX A 151

APPENDIX B 155

APPENDIX C 160
List of Nomenclature

$A$ - positive constant

$a$ - coefficients of Hurwitz polynomial; also states of two-planar manipulator

$C$ - cost or performance index

$C_a$ - augmented cost

$c$ - positive constant, coefficients of Hurwitz polynomial, coefficients of nonlinear differential error

$E$ - transposition error

$E^*$ - desired energy

$E_k$ - kinetic energy

$E_n$ - net energy

$E_p$ - potential energy

$e$ - error; performance as opposed to model error

$e_E$ - energy-based differential error

$e_D$ - differential error

$e_m$ - motor error or model error

$f$ - function in the differential equation of a controlled object or plant
\( G \) - control jacobian or sensitivity derivative matrix

\( g \) - gravitational constant

\( H \) - Hamiltonian function

\( J \) - cost-to-go

\( J^* \) - optimal cost-to-go

\( L \) - performance criterion or loss

\( l \) - layer in a neural network

\( l_p \) - length of the rod in an inverted pendulum

\( m \) - mass of the bob

\( N \) - number of examples provided to a learning algorithm

\( n \) - round, run, or iteration of an algorithm

\( p \) - degree of a polynomial

\( o \) - column vector of ones

\( Q \) - state weighting matrix

\( R \) - control weighting matrix

\( s \) - sum of shoulder and elbow angles

\( s^* \) - desired value of the sum of two angles

\( T \) - final time or duration; transpose, e.g. \( v^\top \) is the transpose of a vector \( v \)
$U$ - set of permissible control signals

$u$ - motor command or control signal

$u^*$ - optimal control signal

$u_{fb}$ - command from Kwato’s feedback controller

$u_{ff}$ - command from Kawato’s feedforward controller

$v$ - context vector; signal vector

$v_{\text{max}}$ - positive constant

$W$ - synaptic weight matrix

$W_\theta$ - fixed synaptic weight matrix

$W_A, W_B$ - weight matrices

$w$ - synaptic weight vector

$w_m$ - synaptic weight vector of model

$x$ - states of the controlled object or plant

$x^*$ - optimal state vector

$y$ - firing rate of a neuron; output of a plant

$y^*$ - target or desired value of $y$

$z$ - input vector to a neuron
List of Greek Symbols

\( \varphi \) - hyperbolic tangent; other nonlinear function used as a feature vector

\( \eta, \eta_m \) - learning rate constants

\( \alpha \) - coefficients of Hurwitz polynomial

\( \theta \) - angular position of the pendulum; feature vector

\( \lambda \) - costate vector

\( \lambda^* \) - optimal costate vector

\( \nu \) - scalar constant

\( \tau \) - variable representing time inside some integrals
List of Abbreviations

GHJB - generalized Hamilton-Jacobi-Bellman equation

HJB - Hamilton-Jacobi-Bellman equation

LMS - least mean squares

LTD - long-term depression

LTP - long-term potentiation

NLMS - normalized least mean squares

PMP - Pontryagin's Minimum Principle

VOR - vestibulo-ocular reflex
Chapter 1

Scope of the thesis

Most sensorimotor skills, such as reaching for a visual target, are not inborn but develop over the first few months of life. Then throughout life almost any such skill, from playing the piano to tennis, improves with practice. And after neurological or musculoskeletal injury, the brain's adaptive mechanisms can often restore function to some extent. In the terminology of control theory, the neural circuits that steer our motor behavior are adaptive controllers, but the brain’s capacity for adaptation far exceeds that of any artificial control system. My aim in this thesis is to apply ideas from adaptive control theory to try to understand sensorimotor control and learning in the brain.

A substantial amount of work has been done on this topic, the theory of adaptive sensorimotor control. An early breakthrough in that regard was Donald Hebb's (Hebb 1949) proposal that learning involves changes in synaptic efficacy driven by neural activity. Since his time, other theorists have gone far beyond Hebb in clarifying the
principles of learning in artificial neural networks and engineering control systems. But my work differs from most of my predecessors in two respects: my emphasis on optimal trajectories and on the constraints inherent in neural communication.

### 1.1 LEARNING OPTIMAL TRAJECTORIES

There are many theories of sensorimotor learning, but most of them deal with the problem of learning to steer some object, such as the gaze point or a reaching arm, along a pre-specified trajectory (Kawato and Gomi 1992; Porrill, Dean et al. 2004; Shibata, Tabata et al. 2005; Abdelghani, Lillicrap et al. 2008). There has been much less work on the problem of how the brain learns to form appropriate trajectories — a problem sometimes called *motor planning*.

By far the most fruitful approach to questions of trajectory formation has been the optimization approach. Here, one tries to explain biological trajectories as attempts to optimize certain variables of biological relevance. For instance, one might posit that the vestibulo-ocular reflex, or VOR, which counterrotates the eyes when the head moves, is designed to optimize the stability of the retinal images, or in other words to minimize retinal-image slip velocity. Or arm trajectories may be chosen to optimize speed, or perhaps accuracy. Starting with a putative optimized variable of this type — or in other words a guess — one then uses computer simulations of the system in question — e.g. the VOR or reaches — to reveal the trajectories that would be expected if the system
were in fact optimizing the variable in question. And one compares those predictions to real human movements.

This optimization method has proven enormously powerful. More than any other approach it has yielded surprising, correct predictions about sensorimotor systems and elegantly compact explanations of complex behavior (Harris and Wolpert 1998; Tweed, Haslwanter et al. 1998; Schreiber, Crawford et al. 2001). For example, eye-head gaze control is governed by a 12-D nonlinear differential equation, but most of its behavior follows from the simple fact that it minimizes retinal image slip and time off-target (Tweed 1997; Tweed, Haslwanter et al. 1998).

Optimization theories also hint at a kind of unity, suggesting that diverse neural systems may all arise in essentially the same way, by learning to minimize feedback signals coding their optimized variables, or costs. For example the VOR is shaped by feedback coding retinal slip velocity (Miles and Lisberger 1981; Mandl, Melvill Jones et al. 1981 ), whereas arm controllers may be formed from the same generic process, only with different feedback signals.

But very little work has been done on this fundamental question of how the brain might learn to optimize particular variables. Or in other words, what are the algorithmic principles of adaptive optimal control in the brain? In my thesis I will be concerned with this issue.
There has been some work on this question. A few theorists have looked at how the brain might construct optimal paths (Sutton, McAllester et al. 2000; Shimansky, Kang et al. 2004; Todorov 2004; Chen-Harris, Joiner et al. 2008). And the engineering literature on optimal control is relevant here. But in the engineering literature, the proposed algorithms are of course all designed to run on digital computers, and no thought is give as to whether these algorithms would also be able to run on the neural networks of the brain.

1.2 BIOLOGICAL CONSTRAINTS ON ALGORITHMS

Within neuroscience, different theorists take very different approaches to this question of biological constraints on algorithms. A major problem is that we all want “biologically plausible” theories, but no one is sure what is really feasible in the brain, i.e. what sorts of computations are possible, or easy versus hard, in neural tissue. Some neuroscientists take a very conservative line, rejecting any computational step that hasn’t been demonstrated physiologically, or that doesn’t have some precedent in the theoretical literature. Other neuroscientists, including most of those studying adaptive optimal control, largely ignore biological constraints. The reasons, presumably, are that the constraints are uncertain and the problem is so very difficult, i.e. the problem of designing an algorithm that can match the adaptive power of the brain. In this view, it makes sense to give ourselves as much freedom as possible and only later, once we have devised one or two candidate algorithms, to start bringing in more constraints.
My sympathies are largely with this latter group, because I am impressed by the extreme difficulty of adaptive optimal control, but in this thesis I will steer a middle path, by incorporating the one biological constraint that can be stated precisely, that is almost certain really to hold in the brain, and that has major implications algorithmically.

1.2.1 Nontransmissible variables

A fundamental difference between computers and brains is that brains represent information in (at least) two complementary ways. One way is to code information in patterns of action potentials, which are short-lived, metabolically expensive, and can be transmitted along axons to remote sites. The second way is to store data in the strengths, or weights, of synapses, which are enduring, inexpensive (individually, as a form of data storage), small, and numerous, but not transmissible (i.e. there is no synaptic weight transport), so they influence remote computations only indirectly, via their effects on the firing of their own postsynaptic cells (Grossberg 1987; Mazzoni, Andersen et al. 1991; Levine 2000; Rolls and Deco 2002). This lack of weight transport constrains the algorithms that can run in biological networks; e.g. it is weight transport that makes the learning algorithm called error-backpropagation (Rumelhart and McClelland 1986; Rumelhart, Hinton et al. 1987) unsuitable for the brain, and efforts to render backprop biologically feasible have focused on removing all such transport from it (Grossberg 1987; Stork 1989; Kolen 1994; van Ooyen 2003).
I think this issue of non-transmissible variables is the most important known constraint governing the algorithms that are viable in biological networks — i.e. it the most firmly established finding that distinguishes brains from electronic computers. I will discuss it further in Chapter 2.

1.3 OUTLINE OF THE THESIS

After laying out the general background concepts in Chapter 2, I will devote Chapters 3, 4 and 5 to three complementary approaches to adaptive optimal control within the constraints of information flow in the brain, i.e. without weight transport.

Because optimal control is so computationally complex, in Chapter 3 I will consider a much simpler mechanism, essentially a generalization of the engineering idea of feedback linearization, which can shape movement trajectories but cannot optimize arbitrary cost functions.

In Chapter 4 I will describe a method by which Pontryagin’s Minimum Principle of optimal control might run in a network without weight transport.

In Chapter 5 I will present my best guess as to the basic mechanism of adaptive optimal control in the brain: a method of successive approximation based on generalized Hamilton-Jacobi-Bellman equations.

Finally in Chapter 6 I sum up my main conclusions and consider some possible extensions and open questions.
Chapter 2

General background

This chapter describes the background concepts I will need in order to describe my thesis work on network algorithms for learning optimal control. These concepts divide into four sections: neural computation, adaptive control and learning, optimal control, and biological constraints on neural communication. After covering these topics I will tie them together in a fifth section on current theories of adaptive and optimal control in the brain.

2.1 NEURAL COMPUTATION

This thesis deals with control algorithms, and therefore it centers on computation. I will begin here with a brief account of how, in principle, a network of neurons can represent variables and functions and perform computations with them. In the two sections that follow I will then describe the computational principles of adaptive control and optimal control in general terms, without referring very much to neurons, but it will be
understood that I am ultimately interested in implementing these computations in networks of brain cells. In the final section of this chapter I will return to the issue of network implementation, to emphasize a constraint that is ignored in most studies involving artificial networks but which is important in real, biological neural networks.

2.1.1 The point neuron

Nerve cells, or neurons, communicate rapidly and precisely over long distances by means of action potentials, or spikes, which are brief fluctuations in the electrical potential across the membrane of the cell. These spikes travel like waves down the output line of the neuron, a branching filament called its axon, which makes connections called synapses with many other neurons. Whenever a spike reaches a synapse it releases a chemical — a neurotransmitter — which increases or decreases the rate of spikes generated by the downstream cell.

Synapses come in many grades of strength: the stronger the synapse, the greater the effect of each incoming pulse on the pulse rate of the downstream cell. More precisely, the effect of a synapse is quantified by its weight, which is a number that reflects the sign and strength of the connection. For instance, if a synapse’s weight is positive then a spike on its presynaptic cell will tend to increase the likelihood that the postsynaptic cell will respond with an action potential of its own. But if the weight is negative then a presynaptic spike will tend to suppress action potentials in the postsynaptic neuron. In other words, a synapse with a positive weight is excitatory and one with a negative
weight is inhibitory. The absolute value of the weight reflects the strength of the excitation or inhibition.

To analyze neural processing mathematically, without getting lost in complications, we try to identify the essential properties of nerve cells and eliminate the rest. For these purposes, the most common model is the so-called point neuron, whose output signal $y$ (its rate of action potentials, or in other words its spike rate or firing rate) is a weighted sum of its inputs followed by a nonlinearity (McCulloch and Pitts 1943) (Figure 2.1). That is, the neuron's output is

$$y = \varphi\left(\sum_i w_i z_i\right)$$  \hspace{1cm} (2.1)

where $z_i$ are the inputs to the neuron, $w_i$ its synaptic weights and $\varphi$ a nonlinear function. The most common nonlinearity used in computational studies is the hyperbolic tangent, or tanh, function, which is easy to compute and, like real neurons, yields outputs confined to a finite range. But whatever nonlinearity we choose, the point neuron incorporates several essential features of real nerve cells: numerous inputs, a single output, synapses of unequal weight, and a nonlinear relation between input and output.

For simplicity, our point neurons differ from real neurons in some respects. First, we don't model individual voltage pulses, but simply define a cell's signal as a real number representing its current pulse rate. Second, real neurons can't have negative pulse rates, but we allow signals to be positive or negative. Third, real synapses are, in all likelihood,
**Figure 2.1:** The point neuron is an idealization of a real nerve cell. This neuron receives some large number, $n$, of input signals $z_i$ (which are firing rates of upstream neurons). It then multiplies each of those signals by a synaptic weight $w_i$, sums the products, and sends the sum through a nonlinear function $\phi$, to yield the output of the cell, $y$, also called its signal or firing rate.
much more complex than a single weight. And fourth, it is probable that other parts of a neuron besides its synapses also influence its responses to signals from other cells.

2.1.2 Networks and universal approximation

We can connect point neurons together to form a network. For instance, a two-layer network as shown in Figure 2.2. The network receives an input vector which is sent to every neuron in the hidden layer. Output signals of the hidden-layer neurons form the input vector for the output neuron, whose firing rate we regard as the output of the network. Clearly the output of the network is related to its input vector by some function, so we can regard the network as computing that function. The network function of course depends on all the weights in the network. If we change those weights then we change the function.

What sort of functions can neural networks compute? It is known that networks of sigmoid neurons are capable of universal approximation (Hornik, White et al. 1989). That is, roughly speaking, they can compute any function. More precisely, given any continuous function over a closed and bounded domain, and any desired degree of accuracy, there is a network of this type which approximates the function at least that well, though the network may need a very large number of neurons. And point neurons with sigmoid nonlinearities aren't the only kind that allows universal approximation in this sense: many other nonlinearities work just as well.
Figure 2.2: Network of point neurons. Here two input neurons send their signals, \( z_i \), to four hidden-layer neurons, which in turn send their signals to a single output cell.
The key concept behind neural network learning is that a network can adjust its own weights to change the function it computes. I will discuss this idea later in the next section.

2.2 ADAPTIVE CONTROL

2.2.1 The plant

Control theory begins with the concept of the controlled process, or plant. In sensorimotor control, for instance, the plant might be a rotating eyeball or a reaching arm (Figure 2.3). Normally the plant has some sort of intrinsic dynamics which we cannot change but simply have to work with. For instance an arm must obey Newton’s and Euler’s laws of motion. Usually these dynamics are represented by a differential equation called the plant equation. One very general form of plant equation is

\[ \dot{x} = f(x,u,t) \]  

(2.2)

where \( x \) is the state of the plant (e.g. in the case of the arm, \( x \) might be a vector of the positions and velocities of all the arm joints), \( u \) is the control signal or command (e.g. the firing rates of motoneurons), and \( t \) is time. So the plant equation describes how the plant state evolves as a function of its current state and the commands it is receiving. For many plants, \( t \) can be omitted from the equation. And in many cases the plant equation is affine-in-u, meaning it has this form:

\[ \dot{x} = f(x) + G(x)u \]  

(2.3)
Figure 2.3: A biological example of a controlled process or plant: an arm reaching to a target. Here $e_i$ is the vertical component of the reaching error, i.e. the vector from target to hand, and $u_j$ is some component of the motor command vector $u$ that activates the arm muscles. For effective motor learning, the neural controller must know the plant’s mechanical properties and how the motor commands influence them.
where $f$ and $G$ are (possibly nonlinear) functions, $f$ yielding a vector output and $G$ a matrix. This form of plant equation is very common, and convenient to work with. And even when we are dealing with a plant whose equation doesn’t have this form, usually we can recast it so it becomes affine-in-$u$. For this reason there is usually no loss of generality in assuming that a plant is affine-in-$u$, as I will do in Chapter 5.

2.2.2 Controllers

The structure or process that generates the commands $u$ we call the controller. If the commands are a function of the state — $u = u(x)$ — then we have what is called a closed-loop or feedback controller. The other basic type is the open-loop controller, which generates a sequence of commands as a function of time, $u(t)$. In other words, if we are trying to build an open-loop controller we are looking for the sequence of commands that will best achieve some very specific aim, e.g. moving the hand between two particular locations. But if we are building a closed-loop controller then we are looking for a rule $u(x)$ which says what command is appropriate in any given state $x$, and usually our aims are more general than with an open-loop controller, e.g. we want a control law that will produce efficient hand movements between any two positions. Feedback controllers are of course much more generally useful than open-loop controllers, and unlike open-loop controllers they don’t fail if the plant is unexpectedly perturbed away from its planned course, so feedback control is very likely what the brain needs. In this thesis I will be concerned mainly with feedback controllers, except for a brief look at open-loop control in Chapter 4.
The job of a feedback controller is to put out commands that produce some sort of desired behavior in the plant. For instance, suppose the plant equation has the form of Eq. (2.3) and we want plant velocity to equal some desired value, $\dot{x}^*$. A physiological example is the vestibulo-ocular reflex, whose plant is the eyeball and which tries to keep the rotary velocity of the eye equal and opposite that of the head. If we suppose that the matrix $G(x)$ in Eq. (2.3) is invertible for all $x$ then it is obvious that the ideal controller is

$$u(x) = G(x)^{-1} \left[ \dot{x}^* - f(x) \right]$$

(2.4)

because if we substitute this formula for $u$ in (2.3) and simplify then we get $\dot{x} = \dot{x}^*$.

### 2.2.3 Why the brain needs adaptive control

Finding a good controller is not usually as easy as in my last example, Eq. (2.4). One reason is that we may not know the functions $f$ and $G$ in the plant equation. For one thing, in biological control systems the plant changes throughout life. For instance the mechanics of the eye muscles change with growth, aging, disease, injury and healing. So controllers in the brain need some sort of internal representation of the plant which can be created and updated by learning. Such a representation is called an internal model, or plant model. In short, the brain needs adaptive control in order to build and maintain plant models.
Another factor that makes control difficult is that, even if we have a good plant model, it is not usually as easy as in Eq. (2.4) to solve the plant equation and produce the desired controller. Often a good way to cope with this problem is, again, learning, i.e. employ an algorithm that learns an appropriate controller (Astrom and Wittenmark 1995; Ioannou and Sun 1996).

Next I will give some simple examples of learning algorithms, before showing how these concepts apply to control systems.

### 2.2.4 Principles of learning algorithms

As I have said, biological learning involves modifying synapses (Byrne 1987), strengthening some and weakening others until the whole network computes an appropriate output. A great deal of work has gone into studying the biochemistry of synaptic change, but in this thesis my focus is algorithmic: I am exploring how, in principle, synapses deep inside a neural network could possibly deduce, based on error signals of various sorts, whether they should increase or decrease and by how much.

There are three basic types of learning that differ in the type of error signal that is involved; these are supervised, reinforcement, and unsupervised learning. Supervised learning is guided by feedback signals coding the overall, performance error of the system (Doya, Kimura et al. 2001). For instance in the VOR, performance error might be a vector signal coding the speed and direction of retinal-image slip velocity. In reaching, this error might be the vectorial separation of hand to target. Reinforcement learning uses
a simpler feedback signal: for example a single number coding the distance from hand to target, but with no information about direction; or the feedback may even be a simple binary message like:”getting closer” versus “getting farther away”. So supervised and reinforcement learning differ in terms of errors, i.e., in cases when the controller's performance is below par, supervised learning lets the controller know exactly what it should have done, whereas reinforcement learning informs the controller whether the performance was inappropriate and in most cases how inappropriate that performance was. Supervised learning is believed to be the main process in motor control because sensory feedback signals contain a lot of vectorial information about the magnitudes and directions of the performance errors. And finally, unsupervised learning doesn’t deal with performance error at all, but with some subgoal that may indirectly contribute to performance. For example, suppose we have an adaptive arm controller that is learning to reach for targets. Often it is useful for certain signals within the system to be uncorrelated with each other, because correlation can slow down supervised learning. So it may be useful to include a mechanism that learns to decorrelate those signals. This learning is unsupervised: it may know nothing about the ultimate performance error of the system — e.g. the location of the hand or the target — but is guided instead by signals coding degrees of correlation. Unsupervised learning is probably important in the early stages of sensorimotor processing, setting the stage for efficient supervised learning. But in this thesis I will be concerned with the final, supervised stage of the process.

Learning algorithms are also divided into two groups — batch and online — based on how data are presented to them. In batch methods, the learner receives a lump of data and
goes over it again and again as it computes the required changes to its synaptic weights. *Online* methods work more like real sensorimotor learning: they receive a stream of ever-changing inputs and learn from that. In this thesis I will be concerned with online learning algorithms.

### 2.2.4.1 Learning by gradient descent

In supervised learning algorithms, the output of the learning network is compared to some other signal coding the *desired* output. The difference between them, called the *error* $e$, drives learning in the network. This procedure is also called error-driven learning, because error signals influence the network's synaptic weights to improve its performance.

The aim of error-driven learning is to bring the error vector $e$ close to $0$. But the process is easier to analyze if we think in terms of minimizing rather than zeroing. So we define a non-negative real number called the *performance criterion*, or *loss*, $L$, in such a way that minimizing $L$ is equivalent to bringing the error as close as possible to zero. For instance in many algorithms, $L$ is defined this way:

$$L = \frac{e^T e}{2} \quad (2.5)$$

We would like to make $dL/dt \leq 0$ (Murray, Li et al. 1994), so that $L$ is always decreasing (or holding steady, if it has reached its minimum attainable value). By the chain rule of calculus we have:
\[ \dot{L} = w \frac{\partial L}{\partial w} \]  

(2.6)

Where \( w \) is the row vector of synaptic weights, e.g. if there are 100 synapses in the network then \( w \) is a 100-element vector containing the weights of all the synapses. So this equation says that the rate of change of \( L \) depends upon the partial derivatives of \( L \) with respect to the synaptic weights. It makes sense, then, to adopt the following learning rule to modify the weights:

\[ \dot{w} = -\eta \left[ \frac{\partial L}{\partial w} \right]^T \]  

(2.7)

where \( \eta \) is a positive number called the learning rate. Substituting Eq. (2.7) in Eq. (2.6) leads to

\[ \dot{L} = -\eta \left[ \frac{\partial L}{\partial w} \right]^T \frac{\partial L}{\partial w} \]  

(2.8)

That is, the learning rule in Eq. (2.8) makes \( \dot{L} \) nonpositive, meaning that it drives down the loss \( L \), as we want. There are many different supervised learning algorithms, but most of them come down to adjusting the weights in a direction roughly opposite \( \frac{\partial L}{\partial w} \). Hence, Eq. (2.7) can be considered the basis of most viable learning mechanisms.

It is also the likely basis for learning plant models in the brain: the command \( u \) is sent to the plant and also to the network that is learning to be a plant model. The output of the true plant is \( \dot{x} \), whereas the output of the would-be model is called \( \dot{\hat{x}} \), where the caret
indicates an estimate. We define an error vector $e = \hat{x} - \dot{x}$ and a loss $L = e^Te/2$, and drive down that loss by some gradient-descent learning algorithm.

The same principles apply also to the other main learning task in control systems — deriving an appropriate controller — but in that case the process is more complicated because we have to worry about minimizing not just the current value of the loss but also losses which may happen in the future, as I will discuss in the next section.

2.3 OPTIMAL CONTROL

2.3.1 The need for foresight

Many sensorimotor controllers require foresight, in the sense that they must consider how their commands at any one moment may affect plant behavior later on. For instance, if you want to flex your elbow quickly, you have to activate biceps at first, but then switch it off roughly midway through the movement, because if you kept on activating biceps after the midpoint then you wouldn’t able to brake in time, and your joint would overshoot its target angle a few hundred ms later.

Some very simple controllers can work without foresight in this sense. For example the vestibulo-ocular reflex, or VOR, monitors head rotation and counterrotate the eyeballs in their sockets so as to keep the retinal image stable. So its job is to match instantaneous eye velocity to instantaneous head velocity. And because the eye-muscle plant is well approximated by a first-order linear filter, instantaneous eye velocity is essentially
determined by the current motor command to the muscles; i.e. it doesn't depend on the commands at any earlier time, so the controller, when it computes its commands, doesn't need to worry about later consequences of its current actions. The controller need only learn the command that reduces the current retinal slip, so in this sense retinal-slip is a \textit{temporally local} error signal.

But in more complex tasks like saccadic eye movements or rapid elbow flexion, no temporally local error signal is readily available. There is no sensory signal of which we can say: minimize this quantity at each moment and you will succeed in minimizing your total time in transit. And without this type of temporally-local error signal, learning will be slow or impossible.

Let us consider further this example of elbow control. I have said that fast and accurate elbow flexion requires biphasic commands: first, an accelerating force from one set of muscles, including the biceps, to get the flexion going, and then, at just the right moment, well before the target is reached, a braking force from antagonist muscles, including the triceps, to bring the hand to a halt without overshoot (Figure 2.4).

If the elbow controller is trained simply to drive down instantaneous position error (the directed distance from hand to target), it learns inappropriate commands which lead to overshoot and oscillation (Figure 2.5). To learn the correct biphasic commands, the brain needs a temporally local quantity whose optimization, at each instant, will lead to the desired overall behavior: rapid elbow flexion without overshoot (Figure 2.4, upper plot of
Figure 2.4: A simple example of the need for foresight in control. Here a simplified “elbow” — a hinge that rotates with inertia but no viscosity or stiffness or gravity — bends to a target angle as quickly as possible without overshooting. The required command — i.e. elbow torque — is biphasic: first accelerate and then, at just the right moment, long before the target angle is reached, apply a braking torque.
Figure 2.5: Without foresight, learning fails: here we have trained the simple elbow of Figure 2.4 to move in such a way that it minimizes the current, instantaneous error in the elbow angle, without considering future consequences. As a result, it doesn’t learn to brake appropriately and so creates inefficient, oscillating movements.
In this thesis I will present three mechanisms I have devised whereby neural controllers might attain foresight in this sense. In Chapter 3 I describe a method based on what I call differential error. Using simulations I will show that it works: it can learn to steer an elbow smoothly to its target without overshoot or oscillation. And I will show that the same idea can handle more-challenging tasks such as the inverted pendulum. I think that this differential-error scheme is attractive, owing to its simplicity, but it may not be versatile enough to explain motor learning in the brain. So in Chapters 4 and 5 I will present two other adaptive-control algorithms, more complex but also more powerful, based on concepts from the theory of optimal control. Now I will provide a brief introduction to that theory.

### 2.3.2 The roots of optimal control

The aim of optimal control is to steer a process, for instance an eye movement or reach, so as to minimize some cost that accumulates over time, such as fuel consumption or time away from a target state. The issue of foresight plays a large role here, because one of the hardest things about optimal control is avoiding decisions that seem sensible at the moment but lead to high costs later on.

Optimal control began with the work of Leonhard Euler in the 18th century. Euler considered questions where the aim is to optimize some cost defined over an interval of time, such as minimizing total time in transit, or minimizing total fuel consumption. He
showed how to compute a temporally-local condition — one which must hold at each instant in the trajectory in order that the non-local cost be minimized. His idea has been developed by later mathematicians, including Hamilton and Pontryagin, into the theory of optimal control (Kirk 1970).

Modern engineering approaches to optimal control fall into two broad classes: those based on Pontryagin’s Minimum Principle and those based on Hamilton-Jacobi-Bellman, or HJB, equations (Sontag 1998). I will discuss these in turn.

2.3.3 Pontryagin’s Minimum Principle

Pontryagin’s principle applies to open-loop controllers. Consider a plant with equation

\[
\dot{x} = f(x, u, t)
\]  

(2.9)

We want to control this plant in such a way as to minimize a performance index or cost:

\[
C = \int_{0}^{\infty} L(x, u, \tau) d\tau
\]  

(2.10)

where \( L \) is some non-negative real-valued function called the cost rate or loss. For simplicity, I have assumed infinite time in the cost function. For finite-time problems, my discussion would have to be modified slightly but the essentials would be the same. I will return to this issue in Chapter 5.
Our objective is to find the control sequence $u(t)$ that minimizes $C$. Of course the appropriate sequence depends on the properties of the plant, so we can regard the plant equation as a constraint and write it like this (Bryson and Ho 1975):

$$f(x,u,t) - \dot{x} = 0$$  \hspace{1cm} (2.11)

To solve this constrained optimization problem, we can use Lagrange multipliers $\lambda$, which in the context of optimal control are also called costate variables. To do this we define an augmented performance index $C_a$ by adjoining our original $C$ with the constraint equation to get

$$C_a = \int_0^\infty \left[ L(x(t),u(t),t) + \lambda^T(t)\left(f(x,u,t) - \dot{x}\right) \right]dt \hspace{1cm} (2.12)$$

Next we define the Hamiltonian function (Giaquinta and Hildebrandt 1996; Lewis 1996)

$$H(x,u,t,\lambda) = L(x,u,t) + \lambda^T(t)\left(f(x,u,t)\right)$$  \hspace{1cm} (2.13)

in terms of which (Lewis 1996) we can rewrite the augmented performance criterion as

$$C_a = \int_0^\infty \left[ (H(x,u,t,\lambda) - \lambda^T \dot{x}) \right]dt \hspace{1cm} (2.14)$$

We want to examine the increment in the augmented performance criterion due to increments of all the variables, $x, u, t$ and $\lambda$. According to Lagrange-multiplier theory, this increment in performance equals zero — i.e. $dC_a = 0$ — at the constrained optimum.
Based on this principle, it can be shown that the necessary conditions for the constrained optimal control sequence are

\[
\dot{x} = \frac{\partial H}{\partial \lambda} = f(x, u, t) \tag{2.15}
\]

\[
-\dot{\lambda} = \frac{\partial H}{\partial x} = \left[ \frac{\partial f}{\partial x} \right]^T \lambda + \frac{\partial L}{\partial x} \tag{2.16}
\]

\[
0 = \frac{\partial H}{\partial u} = \frac{\partial L}{\partial u} + \left[ \frac{\partial f}{\partial u} \right]^T \lambda \tag{2.17}
\]

Equation (2.15) is simply the plant or state equation, which describes how the plant state evolves forward through time. Eq. (2.16) is the costate equation, which describes how the costate evolves backward in time. Much of the difficulty of optimal control arises because the costate equation evolves backwards, and this is just another way of saying that foresight matters: we can’t devise an optimal control sequence unless we know the future consequences of present commands.

If the control variable is unconstrained then the control trajectory \( u(t) \) can be computed from Eq. (2.17). If the control variable is constrained (e.g. if we allow no commands outside the range –1 to 1) then we apply Pontryagin's principle (Pontryagin, Boltyanskii et al. 1962), that the optimal admissible \( u \) is the one that minimizes the Hamiltonian. In that case we have to replace Eq. (2.17) by the more general condition,

\[
H(x^*, u^*, t, \lambda^*) \leq H(x^*, u, t, \lambda^*) \tag{2.18}
\]
where * denotes optimality. In Chapter 4, I will present a mechanism for motor learning based on Pontryagin’s principle, implemented without weight transport.

### 2.3.4 The Hamilton-Jacobi-Bellman equation

The other main approach to optimal control uses the Hamilton-Jacobi-Bellman (HJB) equation. This approach centers on the concept of the optimal cost-to-go function $J^*$, which is the solution to a partial differential equation known as the Hamilton-Jacobi-Bellman equation (HJB). This method leads to a closed-loop form of optimal control, where the command $u$ is a function of the current state (and possibly other parameters such as the location of a target) — in contrast to the open-loop control sequences computed using Pontryagin’s Minimum Principle. Again, the closed-loop approach provides a feedback control law whereas the open-loop form is a curve that gives optimal control values at any time instant over the finite time interval of interest.

To explain HJB, I will use the same plant equation as in Eq. (2.9). We want to find a function $u(x)$ — or in other words a feedback controller — that minimizes some cost, as in Eq. (2.10). Given any feedback controller $u$ we define its cost-to-go:

$$J(x,t) = \int_t^\infty L(x(\tau),u(\tau),\tau) d\tau$$

(2.19)

So $J(x,t)$ is the total cost we will accumulate from now on if we are currently (i.e. at time $t$) in state $x$ and we choose our commands using controller $u$. The objective is to find the
continuous optimal control function $u^*$, i.e. the one that minimizes $J(x, t)$. The optimal cost-to-go function is then the cost-to-go of the optimal controller, i.e.

$$ J^*(x, t) = \min_{u(\tau)} \int_{\tau \leq \tau \leq \infty} L(x(\tau), u(\tau))d\tau $$  \hspace{1cm} (2.20) 

Eq. (2.20) is the Hamilton-Jacobi-Bellman equation.

Before we can appreciate how these engineering concepts apply to sensorimotor control, we have to consider a crucial algorithmic difference between electronic computers and biological neural networks. This is the topic of the next section.

### 2.4 COMMUNICATION IN BIOLOGICAL NETWORKS

Mathematicians and engineers have devised many algorithms for adaptive and optimal control, but in their present forms most of these algorithms aren’t suited for biological neural networks because they rely on a principle that holds for computers but not for the brain, namely the principle that any variable represented in the system can take part in any computation. This is true in a computer because any variable can be transported to the central processing unit. That is, it never happens in a computer that we would like to multiply some variables $a$ and $b$ which are both present in memory but you can’t do it because $a$ and $b$ can’t be brought together.

In this regard, a fundamental difference between computers and brains is that brains represent information in (at least) two complementary ways. As we have seen, one way is
to code information in patterns of action potentials, which are short-lived and can be transmitted along axons to remote sites. The second way is to store data in the strengths, or weights, of synapses, which are enduring, small, and numerous — it is estimated there are 100 billion neurons in the human brain and 60 trillion synapses (Shepherd and Koch 1990). So if there are a lot of data to be stored for an extended time, synapses are a compact, high-capacity form of storage.

They are also probably a metabolically cheaper form of data storage than are action potentials: it is true that synapses consume a lot of energy — packaging neurotransmitters, releasing them when an action potential arrives at the terminal bouton, taking them up again from the synaptic cleft after an action potential is over — but still, if there were some scalar piece of information to be represented and stored in the brain for an hour, or a year, it would likely be cheaper to store it as a synaptic weight than in the rate or pattern of firing of action potentials on a neuron, mainly because you would need just one synapse to store that piece of data as a weight, whereas you would need an entire neuron, generating a steady stream of spikes, to store the same data as action potentials. So the advantages of synaptic storage probably include both compactness and energy efficiency.

But synaptic weights are not transmissible, so they influence remote computations only indirectly, via their effects on the firing of their own postsynaptic cells (Grossberg 1987; Mazzoni, Andersen et al. 1991; Levine 2000; Rolls and Deco 2002). We say that the brain has no weight transport.
In the literature, the issue of weight transport has been discussed mainly in the context of the error-backpropagation learning algorithm, or backprop. Like most learning algorithms, backprop works by the principle of gradient descent which I described in Section 2.2. In other words it computes the partial derivative $\frac{\partial L}{\partial w_i}$ for each synapse $w_i$ in the network, and uses that derivative in its learning rule. The central idea of backpropagation is that all these partial derivatives can be computed efficiently by starting with the synapses in the output layer of the learning network and working back layer by layer. I will not describe the algorithm in detail, as I don’t use it in any of my theories because it is well known to be biologically implausible (Zipser and Rumelhart 1988; Hinton 1989; Bengio, Bengio et al. 1991; Crick 1989). But I do want to explain why it is unsuitable for the brain, to help motivate my own alternative learning schemes.

When the backprop algorithm was published in the 1980s (Rumelhart and McClelland 1986; Rumelhart, Hinton et al. 1987) it soon became clear that it presupposed some mechanism by which one neuron could tell other neurons the strengths, or weights, of its synapses (Grossberg 1987). In other words the algorithm presupposed weight transport. It is mainly this fact that makes backpropagation unsuitable as a learning mechanism for the brain, and efforts to render backprop biologically feasible have focused on removing all such transport from it (Stork 1989; Kolen 1994; van Ooyen 2003). Here I will recap some the arguments against weight transport in the brain.

First of all, there is clearly a way that neurons might transmit information about their synaptic weights: if a synapse $w$ sits on a neuron $N$, then $N$'s action potentials carry
information about $w$, which could be sent to other cells. But the information is mixed up with many other variables, such as the weights of all the other synapses onto $N$, and the values of all of $N$'s inputs. In principle one could monitor firing patterns and deduce individual weights (Levine 2000), but the process would call for a lot of computation and a lot of cells, so it would cancel the main advantage of storing information as weights in the first place: the fact that synapses are smaller and less metabolically expensive than neurons. So weight transport is usually taken to mean some other mechanism, besides action potentials on the postsynaptic neuron, for transmitting synaptic weights to other cells. It is widely agreed that there is no weight transport in this sense (Grossberg 1987; Stork 1989; Kolen 1994; van Ooyen 2003). In the next section, I will consider in more depth the physiological mechanisms that might be considered a basis for weight transport and I will explain why they don't work.

2.4.1 Neural modes of communication other than action potentials

Intracellular transport mechanisms involve motor proteins such as kinesins (Hirokawa, Pfister et al. 1989; Yildiz, Tomishige et al. 2004; Asbury 2005) and dyneins (Bloom 1992), which carry their cargoes — vesicles, proteins, and organelles — along the axon through microtubule tracks, either from the cell body toward the axon terminals — *anterograde transport* — or in the opposite direction, from the axon terminals to the cell body — *retrograde transport*.
Anterograde transport operates at three speeds: fast, intermediate and slow. Fast anterograde transport moves organelles, membrane constituents (proteins, glycoproteins, and lipids), and some small molecular materials (amino acids, sugars, and nucleosides) and Ca\(^{2+}\) (Halperin and Lavail 1975; Knull and Wells 1975; Neale and Barker 1975; Iqbal and Ochs 1978). It can reach speeds of about 410 mm/day (Oztas 2003). This type of transport requires a lot of energy, which is supplied locally along the axon. Intermediate anterograde transport achieves velocities of 3-20 mm/day. Slow anterograde transport carries larger amount of proteins than fast anterograde transport (Kristensson 1977). Its cargo consists mainly of cytoplasmic molecules whereas fast transport moves mainly organelles. Slow transport can manage speeds of up to 12 mm/day.

Almost all of the materials transported by fast anterograde transport also are moved the opposite way, by retrograde transport (Frizell and Sjostrand 1974; Price and Griffin 1977). In general, retrograde transport is one-third to one-half as rapid as fast anterograde (Stoeckel, Schwab et al. 1975; Brimijon and Helland 1976; Brimijon and Wiermaa 1978).

Transport velocities in dendrites can reach 70 mm/day. Dendritic transport may play a role in distributing acetylcholinesterase (Kreutzberg and Toth 1974) and in carrying horseradish peroxidase backward from cell to cell (Lynch, Smith et al. 1975).

Besides these intracellular mechanisms, there is evidence that proteins synthesized by glia can be transferred into axons (Lasek, Gainer et al. 1977). This transfer may provide a way of communication between the nerve and glial cells.
2.4.2 What does this mean for learning algorithms?

We have seen that there are several intracellular transport mechanisms, but none of these comes close to what we would need for learning algorithms based on weight transport. The known transport systems are believed to move materials about for synthesis and recycling, and perhaps for simple feedback signaling. For example, M.-M. Poo and colleagues (Li, Lu et al. 2004) have shown that when LDP or LTP is induced in a synapse then changes occur also in upstream synapses, as in back-prop, but their findings do not imply weight transport (Fitzsimonds, Song et al. 1997; Tao, Zhang et al. 2000). There is no evidence that known transporters carry any information about synaptic weights. Nor is it clear how they could obtain that information, or how their cargo — mostly organelles and protein molecules — could serve as a coding system for communicating synaptic weights.

Even if intracellular transporters could deduce and carry information about synaptic weights, there would still be serious problem with timing. With the fastest retrograde transport it would take at least 15 s for information about synapses to reach upstream synapses, assuming that the neurons are packed as densely as neurons in cerebral cortex. And probably the trip would take considerably longer, because fast transport seems to move mainly organelles, whereas to code the changing weights of thousands of synapses we would presumably need a steady flow of smaller signals — some sort of information-bearing molecule — and this may be more a job for slow transport. So the trip might take something closer to 20 minutes.
But even 15 s is almost certainly too long for a feasible learning mechanism. It might seem that long transport delays might still be compatible with learning: we would just have to be content with slow learning. But in fact long delays are incompatible with even slow learning unless we introduce vast amounts of additional memory to preserve crucial information during the transport delay.

The main problem is that in any feasible learning scheme, the information carried by weight transport would have to be brought together rather precisely with certain other signals, and the slower the transport, the harder it is to achieve this precision. For concreteness, consider the equation below. It describes the learning rule for one specific situation — a layered network of hyperbolic-tangent neurons, learning by online backprop — but it illustrates a general point.

\[
\dot{w}(l, j, i) =
-\eta \sum_k \left( \frac{\partial L(l + 1, k, t)}{\partial y} \left[1 - y^2(l + 1, k, t)\right] w(l + 1, k, j) \right) \times
\left[1 - y^2(l, j, t)\right] y(l - 1, i, t)
\]

(2.21)

Here \( w[l, j, i] \) is the synaptic weight from cell \( i \) in layer \( l - 1 \) to cell \( j \) in layer \( l \). The \( y \)'s are neural signals, and \( L \) is the network's loss function. The equation says that the signal \( y[l - 1, i, t] \) — the input to synapse \( w[l, j, i] \) at time step \( t \) — is multiplied by downstream \( y \)'s and \( \partial L/\partial y \)'s for the same timestep \( t \). For a real sensorimotor system operating in continuous time, this really means that \( y[l - 1, i, t] \) is multiplied by all the downstream signals which it affects, all the way to the network's output layer. This is easy to arrange
in a computer-simulated network, but in the brain it takes time for $y[l - 1, i, t]$ to affect the output layer. And it takes more time to transport information from the output layer back to the synapse $w[l, j, i]$. So the synapse has to remember its input signal $y[l - 1, i, t]$ during this whole round trip.

How long do real synapses remember information about individual input spikes? Experiments on spike-timing-dependent plasticity suggest storage over about 250 ms (Chen and Thompson 1995; Song, Miller et al. 2000; Yamamoto, Kobayashi et al. 2002). That is, changes in synaptic weight — whether the synapse becomes stronger or weaker and by how much — depend on the relative timing of spikes in the pre- and postsynaptic cells. For instance if the presynaptic cell spikes before the postsynaptic then the synapse usually strengthens, and the degree of strengthening depends on the time lapse between the spikes — this finding indicates that the synapse “remembers” when it last experienced a presynaptic spike. But if the time lapse exceeds about 250 ms then there is no change in the synapse. So these findings show that a synapse can remember specific inputs for up to about 250 ms, but they provide no evidence of a longer memory. (Of course synaptic changes can endure for 30 minutes or even, perhaps, a lifetime, but this is a different thing from a synapse remembering the specific input signals it received at specific times, which is what we need for learning in Eq.(2.21)).

This duration of 250 ms is too brief to permit learning by weight transport. As we have seen, the fastest known mechanisms of intracellular transport would need about 15 s to carry weight information one step back through a network. So the known storage time of
250 ms is about 60 times shorter than the shortest delay achievable by known mechanisms of intracellular transport.

Of course it may turn out that synapses are capable of longer storage, and so may be able to cope with long transport delays. But the longer the delay, the more information has to be stored in the synapse. If the round-trip transport time, from synapse to network output and back to the synapse, is $T$ seconds, then each adjustable synapse has to store its inputs, $y[l - 1, i, t]$, that long. If its input signal changes significantly on a subsecond timescale, say $R$ times per second, then the synapse has to store $R$ inputs for each second; i.e. it has to store $RT$ inputs in all. So even if $T$ were just 15 and $R$ were just 10, each synapse would have to store 150 past inputs at all times.

And it is not enough that each synapse store all its inputs from the last $T$ seconds. It would also have to multiply each input by the specific downstream $y$'s and $\partial L/\partial y$'s to which it gave rise. Therefore the mechanisms that transported the $y$'s and $\partial L/\partial y$'s up the axons would have to be temporally very consistent, or they would have to attach some sort of time marker to each $y$ and $\partial L/\partial y$. In other words, as the transport delay increases beyond about $1/R$, the complexity and implausibility of the mechanism increase steadily.

Another option might be that the weights are transported not intracellularly but via some other, “weight-transmitter” cells, e.g. by action potentials in other neurons or perhaps by glia. These cells would “read off” synaptic weights and transmit the information by some rapid means. But there is no evidence that any such cells exist, and if they did, they would likely need huge numbers of read-off mechanisms: one for each synapse whose
weight has to be transported. And it is unclear how a cell could read off synaptic weights; 
e.g. how it might measure another neuron's transmembrane potential at a postsynaptic 
region.

Finally, one could try using a mix of intracellular transport and action potentials. 
Referring again to the backprop equation above, suppose we use cytoplasmic transport 
(Figure 2.6) to carry only the information about weights — the $w[l + 1, k, j]$ — and let the 
other information, such as $y[l + 1, k, t]$, be carried by action potentials, which travel 
quickly. This way, only the weight information is slow, and that may be acceptable if 
weights don’t change very quickly; i.e. synapse $w[l, j, i]$ will know the value that $w[l + 1, 
k, j]$ had a few minutes ago rather than its value now, but that doesn’t matter if $w[l + 1, 
k, j]$ has hardly changed in the interim. Assuming that the weights really do change slowly 
 enough, does this idea make learning by weight transport biologically feasible? Not 
likely, because any one neuron projects to many downstream cells. Looking once more at 
the backprop equation, we see that to compute the adjustment to synapse $w[l, j, i]$ on cell $j$ 
in layer $l$, we need to summate, over all $k$ (i.e. over all neurons to which cell $j$ projects) 
the product of $w[l + 1, k, j]$ with a term involving $y^2[l + 1, k, t]$. If the $w[l + 1, k, j]$ and the 
$y[l + 1, k, t]$ travel by separate routes ($y$ on action potentials and $w$ via the cytoplasm) then 
how can cell $j$ pair them up again? How can it know which $w$ goes with which $y$?

In short, the brain does have communication systems other than action potentials, 
including forward and retrograde cytoplasmic transport of various kinds, but there is no
Figure 2.6: In the backprop learning algorithm, each synapse onto each neuron needs to know the weights of all of the neuron’s downstream synapses. This information might perhaps be transmitted by retrograde cytoplasmic transport, but any such transport would be too slow to support learning.
evidence that these transport synaptic weights. And such mechanisms are about 10 million times slower than action potentials (Oztas 2003), so they would be a poor substitute for most purposes.

Of course a mechanism for weight transport may be discovered tomorrow, but it seems doubtful. We know that natural selection has created one system for rapid, precise, long-range communication, namely action potentials, and that this system is elaborate and metabolically expensive. The hypothetical weight-transport system would have to carry much greater loads at comparable speeds (because its whole point is to mobilize the vast information stores in the synapses) and yet have escaped detection for decades.

For these reasons, neuroscientists try to bar weight transport from their models (Kawato and Gomi 1992; Rolls and Deco 2002).

2.4.3 Learning without weight transport

Some reinforcement learning schemes require no weight transport, but these usually yield much slower and less flexible learning than supervised schemes like backprop (O'Reilly 1996).

A more effective way to learn without weight transport is called *linear-in-the-parameters* learning. Here learning is restricted to a single layer of synapses — those onto the output neurons of the network — and all the upstream synaptic weights remain unaltered (so there is no need to transmit learning related information to them; see Figure 2.7. In most
implementations, all the output neurons are made linear to simplify the learning rule; i.e. 
y is no longer a nonlinear function of the neuron's inputs, but is simply

\[ y = \sum_i w_i z_i \]  

(2.22)

The signals \( z_i \), which are the outputs of the second last layer of neurons, are called 
*features*. They are nonlinear functions of the network's inputs, but they are combined 
linearly to yield the network's output \( y \). Linear-in-the parameters learning overcomes the 
main objection to backprop (Kawato 1990; Porrill, Dean et al. 2004) because it avoids the 
need for fast weight transport.

Armed now with these concepts from engineering control theory and neurophysiology, 
we are ready to consider current ideas about adaptive and optimal-control algorithms in 
sensorimotor systems.
Figure 2.7: Linear-in-the-parameters learning. Learning takes place only in the synapses onto the output layer of cells. All upstream synapses are frozen, meaning they can’t change their weights through learning, so they don’t need to know anything about the weights of downstream synapses, i.e. there is no need for weight transport. The output cells of the network are linear, so that the output error is a linear function of the parameters: the adjustable weights.
2.5 ALGORITHMS FOR SENSORIMOTOR CONTROL

2.5.1 Internal models in the brain

I have said that the brain needs adaptive controllers in order to build internal models of its controlled objects, or plants. Many theories of motor learning have been proposed based on this idea (Houk, Buckingham et al. 1996; Kawato 1999; Merfeld and Zupan 2002; Koerding and Wolpert 2004). In all such models, the brain constructs its models by learning: typically it practices certain movements over and over, observing the relation between motor commands $u$ and motor performance $x$ or error $e$, and it uses these data and some sort of learning algorithm to train a neural network to mimic the relation between $u$ and $x$ or $e$. But the details differ greatly from theory to theory.

There are two types of internal models: forward and inverse. Forward models predict the next states of the plant given the present states and motor commands. It is believed that one function of forward models is to provide closed-loop internal feedback to supplement or replace sensory feedback from the plant — see Figure 2.8, where $y^*$ is the desired target vector, $e$ is the error vector, $u$ is the motor command vector, $y$ is the output vector of the plant, and $\hat{y}$ is the output estimate vector. It has been proposed (Ito 1970; Ito 1984) that the cerebellum receives internal feedback from internal forward models of this sort, and that an internal forward model exists in the vestibular system (Merfeld and Zupan 2002) showed evidence of an internal forward model in the vestibular system. Internal feedback of this kind is particularly important when sensory feedback loops are
Figure 2.8: Forward internal model. This is a neural network that receives a copy of the plant’s input $u$ and yields an output that is an estimate of the plant’s output $y$. 
slow, i.e. when sensory information reaches the brain after a long time-delay. Because of sensory delay, fast and accurate goal directed movements cannot rely on sensory feedback. For example, saccadic eye movements last only about 30-80 ms, so they can’t be guided to their targets by visual feedback, which needs about 75-100 ms to get from the retina via visual cortex to the saccadic control system in the brainstem.

An inverse model, on the other hand, acts as a controller: it receives as input some desired state of the plant, and produces as output the motor commands that will take the plant to that state. An inverse model is often referred as the open-loop or feedforward controller Figure 2.9 because it receives no feedback about the actual state of the plant. The scheme is illustrated in Figure 2.9, where \( y^* \) is the desired target vector, \( u \) is the motor command vector, and \( y \) is the actual output vector of the plant.

2.5.1.1 Direct inverse model

How might the brain construct inverse models? One approach is called direct inverse modeling (Widrow and Stearns 1985). In this scheme, illustrated in Figure 2.10, a learning network receives as input the current output of the plant, \( y \), and the output of this learning network is compared with the input to the plant, \( u \), to create an error signal \( e \). The network is trained by supervised learning, to zero that error.

But there are drawbacks to this approach. In many cases the plant may not be strictly invertible, e.g. there may be many different inputs that yield the same plant output \( y \), so there is no way, given \( y \), to compute which input is responsible. The best you can expect
**Figure 2.9:** An inverse model is a type of controller. It receives as input a signal coding some desired state or output of the plant, $y^*$, and produces a command $u$ which the plant then converts into an output $y$. If the inverse model is truly the inverse of the plant then $y$ will equal $y^*$. 
**Figure 2.10**: Direct inverse model. This is a scheme for building inverse models. The network labeled “inverse model” receives plant outputs and puts out estimates of the command that gave rise to that plant output. These estimates of $u$ are then compared to the true commands $u$ and the differences, called the model errors $e_m$, are used to improve the inverse model.
is that the inverse model computes one of the inputs that might have caused y. But if the plant is nonlinear, the direct inverse modeling scheme may fail even to find even one such input, and may instead compute an input that yields some plant output not equal to y, i.e. the method simply fails. Another drawback is that the learning is not goal-directed (Jordan and Rumelhart 1992), i.e., learning acts on the input quantity while the controller acts on the output quantity, for example steering the position of the arm to the desired target. It is more likely that learning in the brain involves acting on the same quantities that the controller is trying to minimize.

2.5.1.2 Distal teacher

Another way of learning an inverse model is to use a distal supervisor or distal teacher (Jordan and Rumelhart 1992). This scheme, illustrated in Figure 2.11, involves two interacting processes: one where a forward model is learned and another where that forward model is used to learn the inverse model.

In this scheme, we must distinguish between different error signals that train the two models: forward and inverse. The forward model is trained using what is called the model error, \( e_m \) — the difference between the actual output of the plant and the estimates put out by the model, i.e.

\[
e_m = \hat{y} - y
\]  

(2.23)

And the synapses of the forward model change according to the following learning rule,
Figure 2.11: Distal teacher learning. The network labeled “model” learns to be a forward model of the plant. When it has done so, it can then process the performance error $e$ to produce a signal (gray line) that can be used to train the controller, but this process requires that the signal $e$ pass backward through the model network in a way that is similar to the flow of information in the backprop learning algorithm.
\[ w_m = -\eta_m e_m \frac{\partial e_m}{\partial \hat{y}_m} = -\eta_m e_m \frac{\partial \hat{y}_m}{\partial w_m} \] (2.24)

The inverse model, on the other hand, is a controller, so its job is to make the plant's outputs equal some desired values, so it is trained using the performance error,

\[ e = y - y^* \] (2.25)

and the controller weights change according to

\[ \dot{w} = -\eta e \frac{\partial e}{\partial w} = -\eta e \frac{\partial e}{\partial u} \frac{\partial u}{\partial w} = -\eta e \frac{\partial f}{\partial u} \frac{\partial u}{\partial w} \] (2.26)

Notice, in this equation, that the learning rule requires knowledge of the partial derivative \( \partial e/\partial u \), or equivalently \( \partial f/\partial u \). In control theory, this quantity (which is a matrix) is called the control jacobian or the matrix of sensitivity derivatives. It plays a crucial role in any algorithm for adaptive control. The reason is simply that any adaptive controller needs to know how its motor command \( u \) affects its performance error \( e \), and this information is in the sensitivity derivatives. The problem, though, is that these derivatives depend on the properties of the physical plant — its muscles, tendons etc. So how can the controller obtain the information it needs to compute \( \partial e/\partial u \), or equivalently \( \partial f/\partial u \)?

The plant model embodies the plant equation \( \dot{x} = f(x, u) \), so it contains complete information about \( \partial f/\partial u \), i.e., the model contains the missing piece of information we need to train the controller. But there is still the problem that, in Jordan and Rumelhart’s...
scheme, the information about $\partial f / \partial u$ is not contained in the model's output — i.e. in the firing rates of its neurons — but is distributed among its synaptic weights.

Jordan and Rumelhart (1992) therefore use weight transport (and specifically the error-backpropagation algorithm) to transport this information from the model to the controller. But how could this work in the brain? In Section 2.4.2 I argued that known cytoplastic mechanisms are very unlikely to be able to measure or encode information about synaptic weights, and that in any case they are too slow to provide the kind of weight transport that is needed for learning.

Later in this chapter I will describe a mechanism called implicit supervision that can learn and transmit sensitivity derivatives as required in the distal teacher theory, but without weight transport. In the meantime, though, I will follow the story chronologically, looking at how this difficulty in Jordan and Rumelhart’s scheme led theorists to search for alternative schemes where $\partial e / \partial u$ didn’t have to be transported from model to controller at all, because it was built into the controller from birth.

2.5.1.3 Feedback-error learning

Kawato and colleagues have proposed what they called a feedback-error learning model as a possible mechanism to learn inverse models (Kawato, Furukawa et al. 1987). Their mechanism comprises an inverse model (or in other words a feedforward controller) and a feedback controller (Figure 2.12). According to Kawato and colleagues, the feedback controller processes the performance error $e$ and generates a motor error signal that is used as the training signal for the inverse model. The motor error also serves as a motor
Figure 2.12: Feedback error learning. The network labeled “controller” is an innate, feedback controller which produces a command $u_{fb}$. The input to the plant is the sum of $u_{fb}$ and another command, $u_{ff}$, from an adaptive feedforward controller, labeled “inverse” in the diagram. As the adaptive controller improves, there is less and less error $e$ and so the feedback controller becomes less and less active, until its function has been taken over entirely by the adaptive feedforward controller.
command, $u_{fb}$ that is eventually replaced by the feedforward motor command, $u_{ff}$ after an inverse model is learnt. The key idea is that as the inverse model improves through learning, the performance error gets smaller and smaller and therefore the feedback controller contributes less and less to the motor command. Its contribution vanishes when the inverse model has finished learning its job.

Unlike the direct inverse method, Kawato’s feedback-error mechanism always learns a correct inverse, but learning still involves using an input quantity instead of an output quantity (Kawato and Gomi 1992).

Kawato’s feedback-error learning scheme avoids the main difficulty in Jordan and Rumelhart’s distal teacher theory — the need to transport sensitivity derivatives — but it does so by assuming that information about those derivatives is built into the feedback controller from birth. And in the feedback-learning theory, this feedback controller is not adaptive.

In other words, the feedback-error learning theory assumes that the controller's knowledge of $\frac{\partial e}{\partial u}$ is innate rather than learned. But if that were so, then if the sign of $\frac{\partial e}{\partial u}$ reversed — for example by transposing muscles or by having a person wear reversing spectacles — the true value and the innate estimate would have opposite signs, and learning would run in the wrong direction, strengthening synapses that should be weakened and vice versa.
This point is illustrated by the simulations in Figure 2.13 and Figure 2.14, showing the VOR (vestibulo-ocular reflex) learning by the feedback-error algorithm. Recall that the aim of the VOR is to make eye velocity equal and opposite to head velocity, so as to stabilize the retinal image. Figure 2.13 shows that the feedback-error learning scheme manages to do this if the feedback controller’s innate estimate of $\frac{\partial e}{\partial u}$ has the correct sign. We see that after learning, the controller makes eye velocity equal $-1$ times head velocity. But if the innate estimate of $\frac{\partial e}{\partial u}$ has the wrong sign then the learning fails. Figure 2.14 simulates the sign change of $\frac{\partial e}{\partial u}$ by reversing the sign of $u$ in the plant equation (as if the left- and right-pulling eye muscles had been transposed). As the plot shows, the controller fails to adapt to this change.

But there is extensive evidence that real neural adaptation doesn't collapse in this way when people wear reversing goggles or when nerves or muscles are transposed, which implies that the brain is able to update its estimates of sensitivity derivatives when the plant changes (Stratton 1897; Ewert 1930; Sugita 1996). These findings tell us that any viable theory of motor learning needs a mechanism like Jordan and Rumelhart’s distal teacher for learning sensitivity derivatives, but without weight transport. A suitable mechanism has recently been found (Abdelghani, Lillicrap et al. 2008).

2.5.1.4 Sensitivity derivatives without weight transport

Abdelghani et al. (2008) have shown that the sensitivity derivatives can be learned without weight transport by a mechanism they call implicit supervision, where the sensitivity derivatives are coded in firing rates, and are therefore transmissible, unlike
Figure 2.13: A controller for the vestibulo-ocular reflex, or VOR, learns its task when the sign of its innate estimate of sensitivity derivatives is accurate.
Figure 2.14: The VOR controller fails to learn its task when the sign of the sensitivity derivative matrix changes midway through the simulation, because the controller’s innate estimate of that derivative has the wrong sign, and so guides learning in the wrong direction.
Jordan and Rumelhart's distal-supervisor theory where the information was distributed in the model's synaptic weights.

**In the implicit supervision method**

Figure 2.15 we define a vector $z = (u, v)$ consisting of all the elements of the command $u$ and the context vector $v$, where $v$ is defined to be everything, other than $u$, that we need to determine $e$. For example in the VOR, $v$ would be the velocity of the head and the position of the eye, because if we know these things and $u$ and the plant equation, we can compute $e$. So with $z$ defined this way, we know that $e$ is determined by $z$ and therefore, by the chain rule of calculus, we can write

$$\dot{e} = \frac{de}{dz} \dot{z}$$

(2.27)

This equation tells us that we can learn $de/dz$ by producing an estimate, $<de/dz>$, multiplying that estimate by $\dot{z}$, and comparing the product, which we call $<\dot{e}>$, to the actual $\dot{e}$. If these differ, we have a *model error* vector $e_m = <\dot{e}> - \dot{e}$. So we can define a corresponding model loss and minimize it by gradient-descent learning. Figure 2.15 illustrates a neural circuit for this job. The matrix $de/dz$ contains all the elements of $\partial e/\partial u$ as a submatrix, so by learning $de/dz$, the circuit in Figure 2.15 also learns the sensitivity derivatives. And it codes them in firing rates rather than in synaptic weights, so it can transmit them to the controller where they are needed. Simulations show that this scheme can re-establish control even after the sensitivity derivatives switch sign. The method is fast, robust, and applies to linear and nonlinear systems of any dimension or order.
Implicit supervision is currently the only known mechanism for learning sensitivity derivatives and using them for adaptive control without weight transport, but of course other possibilities may be found, and it may turn out that the brain uses some quite different method. Nevertheless, implicit supervision shows that there is no fundamental conceptual difficulty in understanding how biological controllers might learn sensitivity derivatives. So in my thesis I have not worried about $\partial e/\partial u$, but have simply assumed that that information is available, and I have concentrated on the problem of learning controllers. Or in other words, Abdelghani et al. (2008) have proposed a solution to the weight-transport problem for plant-model learning, showing how this can be done with all the transmissible variables represented in neural firing rather than synaptic weights. My aim is to do something analogous for the problem of constructing roughly optimal controllers without weight transport.
Figure 2.15: In implicit supervision, a forward plant model receives the same command $u$ as the plant, and also other contextual variables $v$, and learns to compute an estimate of the sensitivity derivatives of the control system by a method involving the time-derivative of the vector $z = (v, u)$ as described in Abdelghani et al. (2008). The sensitivity derivatives are represented in the firing rates of the model’s neurons, so they can be transmitted to the controller without the weight transport that was presupposed in Jordan and Rumelhart’s distal teacher scheme.
2.5.2 Optimal sensorimotor control

The sensorimotor theories I have described so far all assume that a movement trajectory has already been specified. All the controller has to do is track that desired trajectory at each moment, so the issue of foresight doesn’t arise. For instance in the VOR the task is simply to match current eye velocity to the current motion of the head. In this section I consider ideas about how the brain might specify its movement trajectories so as to minimize certain costs, so now we are in the province of optimal control.

Until recently, most optimal sensorimotor control models have focused on open-loop optimal control. As I have said, open-loop optimal control is based on Pontryagin`s Minimum Principle in which the control sequence or control trajectory is selected so as to optimize some cost function. Biological theories centering on optimization include minimum jerk (Flash and Hogan 1985), minimum torque change (Uno, Kawato et al. 1989; Shibata, Tabata et al. 2005), minimum end point variance (Harris and Wolpert 1998). These papers explore the consequences of minimizing specific cost functions to model observed behaviors but don't explain how the brain might learn to minimize those costs, which is the aim of my thesis.

Todorov has emphasized the importance of optimal closed-loop or feedback control in motor systems (Todorov 2004). In closed-loop optimal control the feedback controller's gains are optimized to produce an optimal control law. The optimal control law is an optimal mapping of the states to the motor commands. In closed-loop optimal control, the
desired trajectories do not need to be planned but they simply fall out from the control law.

Todorov has considered how optimal control might be learned in his iLQG method (Todorov 2005). Generally, if the plant dynamics are linear in $x$ and $u$, the cost is quadratic and noise is Gaussian, then the optimal control can be formulated as Linear Quadratic Gaussian (LQG), which can be solved analytically. However, if the plant dynamics are nonlinear and the cost is non-quadratic, LQG does not hold anymore. According to Todorov’s theory, one can make use of time-varying linear approximations of the plant and cost and apply a similar formalism to that of LQG to iteratively improve the control law, until at least a local minimum of the cost function is achieved.

But Todorov has so far ignored the issue of biologically plausible information flow; that is, he has made no attempt to cast his algorithm in a form that could be executed by networks of neurons without weight transport. For instance he employs numerous matrix inversions, but he doesn't specify whether the matrix elements are stored in the form of firing rates of neurons, or in the form of synaptic weights.

Shimansky and colleagues have proposed a mechanism that acquires knowledge and skill necessary for optimal control of a multisegmental limb dynamics knowing only the dimensionality of the plant’s state space, but their solution relies on error-backpropagation and therefore weight transport (Shimansky, Kang et al. 2004).
So current theories of adaptive optimal control presuppose a communication form that is almost certainly not available in the brain. The aim of my thesis is to provide the first algorithm for adaptive optimal control in brainlike networks, without weight transport.
Chapter 3

Trajectory learning through differential error signals

3.1 INTRODUCTION

In Chapter 2, I pointed out that many sensorimotor controllers require foresight in the sense that they must consider how their commands at any one moment may affect plant behavior later on. Perhaps the simplest example is fast elbow flexion, where biceps must be activated at first, but then it must be switched off again, and its antagonist muscles switched on, long before the movement is finished, so as to avoid overshooting the target later on. I said that the main branch of control theory concerned with this problem is optimal control. In later chapters I will consider mechanisms of sensorimotor learning based on the theory of optimal control, and I will argue that some mechanism of this sort does operate in the brain, but my work in those chapters will also underline that optimal-
control algorithm are computationally complex, so it makes sense to consider other, simpler principles that might also achieve foresight.

These simpler mechanisms do not achieve optimal control but still they are interesting. One reason is that some brain systems may not require strict optimality. Another reason is that the learning mechanism I consider most plausible — the one I present in Chapter 5 — cannot start from scratch but requires a pre-existing controller that is capable of driving the plant to its target. That pre-existing controller needn’t be anywhere near optimal, but it does have to be competent enough to take the plant to its goal. The mechanism I discuss in this chapter provides a simple way to create initial controllers for this purpose.

3.1.1 Hurwitz error

My work in this chapter builds on a recent suggestion (Lillicrap, Abdelghani et al. 2005), which was in turn based on the notion of Hurwitz polynomials (Nise 2004) from linear control theory. A Hurwitz polynomial represents a differential equation, with a variable $s$ representing differentiation with respect to time. That is, $s$ is the rate change of some variable, $s^2$ is the second derivative, or acceleration, of that variable, and so on. So the polynomial

$$a_n s^n + a_{n-1} s^{n-1} + \ldots + a_1 s + a_0$$

(3.1)

Represents the differential equation
Why is this representation useful? Hurwitz and other showed that if the roots of the polynomial (3.1) are all real and negative then the variable $x$ in Eq. (3.2) will evolve toward 0 and stay there. In control theory, this means that if we can make our system obey an equation like Eq. (3.2), with the variable $x$ replaced by the performance error $e$, and if we choose the coefficients $a_i$ so that the corresponding polynomial, Eq. (3.1), has all negative roots, then we know that our error will go to zero. This is the key idea behind the control technique of feedback linearization.

Lillicrap et al. suggested that a formula like Eq. (3.2) could be used as a “Hurwitz error” signal to train an adaptive controller. For instance in a second-order task such as elbow flexion control, we define $e$ to be the position error — the difference between the actual angle of the elbow joint and its desired value — and we define the Hurwitz error

$$e_D = a_2 \ddot{e} + a_1 \dot{e} + a_0 e$$  \hspace{1cm} (3.3)$$

It is important to distinguish the various $e$’s in this formula: $e$ is position error (the angle between actual and desired elbow position), $\dot{e}$ is the first time-derivative of position error, $\ddot{e}$ is the second derivative, and the $a$’s are constant coefficients. On the left-hand side of the equation I have called this linear combination of errors and derivatives $e_D$, which stands for differential error rather than Hurwitz error, because I will soon be generalizing the idea to cases where $e_D$ is no longer a linear function of the $e$-derivatives,
and Hurwitz’s formula no longer applies. For the present, though, we choose the coefficients to satisfy the Hurwitz property, i.e. to make all the roots of the polynomial $a_2 s^2 + a_1 s + a_0$ real and negative.

How does this differential error help us? Suppose the aim of the control system is to bring the position error $e$ near 0. In Chapter 2, I showed that if we train a controller simply to zero $e$ then it produces trajectories with large oscillations. So now instead we will train the controller to zero $e_D$. Or equivalently, we will train it to minimize a loss function defined this way:

$$L = \frac{e_D^2}{2}$$

(3.4)

As usual, our aim is to make $\dot{L} \leq 0$ (Murray, Li et al. 1994; Nise 2004), so that $L$ is always decreasing (or holding steady, if it has reached its minimum attainable value). We apply the general argument I outlined in Chapter 2 to arrive at this learning rule:

$$\dot{w} = -\eta \frac{\partial L}{\partial w} = -\eta \frac{\partial L}{\partial e_D} \frac{\partial e_D}{\partial u} \frac{\partial u}{\partial w} = -\eta e_D \frac{\partial e_D}{\partial u} \frac{\partial u}{\partial w}$$

(3.5)

In Eq. (3.5), the matrix $\partial e_D/\partial u$ is the sensitivity derivative — I explained in Chapter 2 how it can be learned and delivered to the controller without weight transport by implicit supervision, so in what follows I will simply assume that this matrix is known. Similarly, the quantity $\partial u/\partial w$ depends on the structure of the controller itself, so there is no puzzle
as to how it could be known. For instance, suppose the command $u$ is given by the formula

$$u = w\theta$$

where $w$ is the row vector of adjustable synaptic weights and $\theta$ is a column vector of features (recall that features were defined in Section 2.4.3, just below Equation (2.22), though there I called the feature vector $z$ rather than $\theta$). Then

$$\frac{\partial u}{\partial w} = \theta$$

So there is no problem making all the terms on the right-hand side of this learning rule, Eq.(3.5), available to the controller. And with this rule, the controller can adjust its weights to make $e_D = 0$ (or at least to bring it close to 0).

So after learning we have, at least approximately,

$$e_D = a_2\ddot{e} + a_1\dot{e} + a_0e = 0$$

Or in other words

$$\ddot{e} = -\frac{\alpha_1}{\alpha_2}\dot{e} - \frac{\alpha_0}{\alpha_2}e$$

That is, the trained controller causes the position error $e$ to obey this differential equation. Because its coefficients fulfill Hurwitz's criterion, the equation guarantees that $e$ will
evolve smoothly toward zero, which is what we wanted originally. So with this linear differential error, the controller learns the biphasic commands that drive the elbow smoothly to its target.

The simulation in Figure 3.1 demonstrates that the idea works, in the sense that the system reaches the target along a sensible-looking path. In fact this scheme, so far, is simply the engineering technique of feedback linearization (Khalil 2002), except that where engineers solve an equation to find a controller that zeroes $e_D$, Lillicrap et al. instead use learning to create a controller that (perhaps only roughly) zeroes $e_D$. This latter approach is slightly more flexible, because it can still deliver good results in cases where $e_D$ can’t be zeroed exactly, and it is more biologically feasible because in a network it is easier to implement the learning rule Eq. (3.5) than the symbolic algebra needed to solve the equation.

In the simulation, the state trajectories look reasonable, but they are very unlikely to minimize any cost function we might have in mind. For instance, if we were interested in driving the system to its goal as quickly as possible, then our controller would be disappointing, because it is far from time-optimal.

Figure 3.1 show that the motions don’t have the bell-shaped velocity profiles of time-optimal movements. In the next section I describe my own work on generalizing the method of Hurwitz errors so that it can yield something approximating time-optimal control.
Figure 3.1: Elbow flexion controller trained with Hurwitz error.
3.2 NONLINEAR DIFFERENTIAL ERROR

I have pointed out that a controller trained with linear differential error is far from time-optimal. Here, I will present a mechanism, based on a nonlinear generalization of Hurwitz polynomials, and drawing on other ideas (Slotine and Li 1991; Krstic, Kanellakopoulos et al. 1995; Lyshevski 1998), to define a nonlinear $e_D$ that enforces roughly time-optimal control.

The key idea is as follows. If we choose a linear differential error then the system learns linear dynamics, which are far from time-optimal because the velocity declines in direct proportion to the remaining error. For instance, in the simplest, first-order case, the system would learn the dynamics

$$a_1 \dot{e} + a_0 e = 0,$$

i.e.

$$\dot{e} = -\frac{a_0}{a_1} e = -ce$$  \hspace{1cm} (3.10)

where $c$ is a constant. It follows that, as the system approaches its target, and position error $e$ therefore approaches 0, the rate of change of that error will also decline to 0, so the system will move toward its target more and more slowly.

For time-optimal control, we don’t want the velocity to decline to zero in this drawn-out, gradual way. Instead, we want to maintain a high velocity until the system is very close to its target — i.e. until $e$ is very near 0 — and then reduce velocity quite suddenly. In other words, we want $\dot{e}$ to be a nonlinear function of $e$, maintaining a large value as $e$ declines, and then dropping to 0 abruptly when $e$ is very close to 0. This is the idea behind so-
called bang-bang control. With a pure bang-bang controller (and a first-order system) the nonlinear function relating $\dot{e}$ and $e$ is the step function:

$$\dot{e} = -v_{\text{max}} \, \text{sgn}(e) \quad (3.11)$$

Here $v_{\text{max}}$ is a positive constant representing the maximum achievable rate of change of $e$, and sgn is the signum function, which yields an output of 1 for any positive inputs, $-1$ for negative inputs, and 0 for an input of 0. So with this function $e$ approaches 0 with at its maximum rate until $e$ reaches 0, at which point $\dot{e}$ suddenly drops to zero, and so $e$ stays fixed at 0 as well.

This bang-bang scheme drives $e$ to zero in minimum time, but it has some practical disadvantages: in the presence of noise or feedback delays it can lead to instability. So for many applications it is better to adopt a slightly less aggressive strategy, replacing the sharp-edged step function with a more rounded sigmoid approximation to it, for instance a hyperbolic tangent, i.e.

$$\dot{e} = -v_{\text{max}} \, \tanh(e) \quad (3.12)$$

With this approach we no longer have strictly time-optimal control, but we have something reasonably close to it and more robust.
Equation (3.12) is first-order, but we can generalize it to achieve roughly time-optimal behavior for higher-order systems. For example, in second-order elbow flexion control, we can use

$$\ddot{e} = -c_2 \varphi_2 (\dot{e} + c_1 \varphi_1 (e)) \quad (3.13)$$

where the $c_i$ are constants and the $\varphi_i$ are sigmoid nonlinearities.

It follows that we should be able to achieve roughly time-optimal control by training a controller to zero the dynamic error

$$e_D = \dot{e} + c_2 \varphi_2 (\dot{e} + c_1 \varphi_1 (e)) \quad (3.14)$$

The simulation in Figure 3.2 shows that this method works. Trained with a nonlinear differential error mechanism, the controller learns to produce movements with the appropriate biphasic commands and bell-shaped velocity profiles. Comparison with the true, strictly time-optimal solution shows that my adaptive controller performs movements about 97% as fast as time-optimal ones.

A major advantage of this differential error mechanism is that it doesn't have to store the control or state trajectory, but rather works based on instantaneous state information. This makes the mechanism much simpler than most algorithms for learning optimal control, and easier to implement in a neural network.
In this section I have shown how a controller trained with nonlinear differential error can learn roughly time-optimal control. But it remains to be seen whether the same technique can handle more difficult control tasks. The most challenging tasks for this method are those where the optimal path to the target is highly indirect. That is, the method of differential errors is based on defining some simple dynamics in which the system is drawn in to its target as smoothly and directly as possible. But what if no direct path is available? In the next section, I will extend the method of differential errors to cope with cases of this type.
Figure 3.2: Elbow flexion controller trained with nonlinear differential error.
3.3 LEARNING MORE-COMPLEX MOTIONS

The inverted pendulum has been a test bed for different control theories (Astrom and Furuta 2000; Bugeja 2003) for the last 50 years. The task is to swing a pendulum from its downward position to its upright position (Figure 3.3). If the command $u$ is powerful enough, it may be possible to swing the pendulum up directly to the desired position. But in the more interesting case, the command is too weak for a direct approach, and so the controller must instead learn to swing the pendulum part way up in one direction, then let it fall and build up momentum, before driving it up to the zenith on the other side (like a child pumping a playground swing). Evidently this task requires a controller intelligent enough to learn to create the initial backswing.

This problem is relevant to sensorimotor control because the brain faces similar tasks requiring actions analogous to a backswing. In throwing, for instance, you draw your arm back before moving it forward. And in jumping up, you first crouch down. So our neural controllers show evidence of foresight and intelligence in learning these kinds of challenging motor tasks.

3.3.1 Energy-based differential error

I suggest that one solution is to define a differential error in terms of energy rather than position. The advantage is that when we think in terms of energy then it is possible to take a direct route to the target in this sense: you simply drive the energy ever upward.
Figure 3.3: Inverted pendulum task: swing up from the downward to the upright position.
Consider a pendulum (Figure 3.4) where \( m \) is the mass of the bob, \( l \) is the length of the rod, \( \theta \) is the angle between the rod and the vertical (when \( \theta = 0 \), the pendulum is hanging straight down). This apparatus can be modeled as a second-order nonlinear system as shown in Eq. (3.15), where the state variables \( x_1 \) and \( x_2 \) are the pendulum’s angular position and velocity respectively and \( u \) is the control signal or command.

\[
\begin{align*}
\dot{x}_1 &= x_2 \\
\frac{\dot{x}_2}{l} &= -\frac{g}{l} \sin x_1 + \frac{u}{ml^2}
\end{align*}
\]  

The net energy of the pendulum is the sum of its potential and kinetic energies:

\[
E_n = E_p + E_k = mgh + 0.5mlx_2^2
\]

where \( h \) is the height of the bob above its lowest possible position. In order to swing the pendulum from the downward position to the zenith, its energy must be increased.

At the zenith, the potential energy is

\[
E^* = 2mgl
\]

So if we could get the net energy to equal \( 2mgl \) then the pendulum (being an energy-conserving system when \( u = 0 \)) would swing up to the zenith and then stop, because all
Figure 3.4: Inverted pendulum parameters.
its energy would then be potential and none of it kinetic (though the stopping is not essential to the current discussion, whose only aim is to swing the pendulum up to the zenith; I will consider the harder problem of swinging and holding in the next section).

How can we drive the energy to its desired value? If we differentiate the net energy, Equation (3.16), and simplify, we find that its time derivative is

$$\dot{E} = x_2 u$$  \hspace{1cm} (3.18)

From Equation (3.18) we see that the net energy will increase if $x_2 u > 0$ and decrease if $x_2 u < 0$. So if our aim is to drive the pendulum to the zenith, then a candidate energy-based differential error to train the controller is

$$e_E = \dot{E} - c$$  \hspace{1cm} (3.19)

where $c$ is a positive constant and the subscript $E$ in $e_E$ is meant to remind us that this differential error is based on energy.

Computer simulations show that a controller trained by this energy-based differential error learns to make a backswing and then drive the bob to its target position, as shown in Figure 3.5. In that figure, the system resets after each success to show that it can repeat the task consistently.

In this section I have aimed only to get the controller, trained with differential energy-based error, to swing the bob to its zenith, not to hold it there. With this goal
Figure 3.5: Trained with the energy-based differential error of (3.19), a controller learns to swing the pendulum to the upright position.
accomplished, in the next section I will describe a concept of hybrid differential error that can be used to train a controller to swing up and hold the pendulum at the zenith.

### 3.3.2 Hybrid differential error

In the engineering literature, the job of swinging and stabilizing the pendulum at the zenith is managed with two controllers: one to swing up and then another for stabilization close to the zenith. I instead trained a single controller with a hybrid differential error vector, containing an energy component and a position component:

\[
e_p = \alpha_2 \ddot{\theta} + \alpha_1 \dot{\theta} + \alpha_0 e
\]

\[
e_E = \dot{E} + \varphi(E - E^*)
\]

In my first simulations of this method, I used hand-picked features; that is, I choose the vector of functions called \( \theta \) in Eq. (3.6) so that they would be particularly well suited for this control task. Given these hand-picked features, the algorithm worked very well.

But in neuroscience one might regard hand-picked features as cheating. After all, one might ask, how could the brain know to choose those specific features?

Actually, this isn’t all that implausible. It is possible that the brain might use some learning algorithm to construct especially useful features. It is also possible, and even likely, that natural selection would create a good set of features and store them in the genome so they were available to the learning organism from birth, in the same way that
natural selection (presumably) creates the feedback controller in Kawato’s feedback-error learning scheme from Chapter 2, or the way that natural selection discovers the learning rules used in any theory of sensorimotor learning. So it isn’t necessarily cheating to use hand-picked features, not is it at all rare in the neuroscience literature.

Nevertheless, an algorithm is more impressive if it can do its job with random, non-hand-picked features, so my next step was to try that. I found that the method required 70 features (not an unreasonable number) but learned very slowly, requiring about 500,000 time steps, when I trained it with the simple least mean square algorithm (Haykin 1998) (LMS). I was able to speed it up by optimizing a quantity called its learning rate constant, but if I were to propose this method as an option for the brain then I would have to propose a neural mechanism to deduce optimal rate constants. Another option is to use an algorithm called normalized least mean square (Haykin 1998) (NLMS) which learns much more quickly using not a rate constant but an easily computed variable learning rate factor.

But the fastest-learning method, and the one I have used for all my pendulum simulations in this thesis, is based on the idea of a weightless controller (Fortney and Tweed 2006). In this scheme there are no features, so the issue of choosing random versus hand-picked ones doesn't arise. The weightless controller worked by setting up the loss function \( L \) based on a differential error. For example, in the case of my hybrid differential error vector, the loss might be
\[ L = \frac{1}{2} e_D^T e_D = \frac{1}{2} e_p^2 + \frac{1}{2} e_E^2 \]  

(3.21)

We want \( L \) to go to zero, and preferably quickly. So we might, for instance, like to have

\[ \dot{L} = -\eta L \]  

(3.22)

because then \( L \) would evolve to zero exponentially. Or for greater speed we might prefer

\[ \dot{L} = -\eta L^{1/2} \]  

(3.23)

because this formula implies that

\[ \dot{e}_D = -\eta \frac{e_D}{2 \|e_D\|} \]  

(3.24)

which is a vectorial version of bang-bang control. More generally, let us simply suppose that we have *some* desired function in mind,

\[ \dot{L} = \Psi(L) \]  

(3.25)

Next I will show how we can use a weightless controller to force \( L \) to obey these dynamics.

To begin with, we define a *context vector* \( y \) consisting of everything, other than \( u \), that we need to compute \( e_D \). Then by the chain rule of calculus we can write

\[ \dot{L} = \frac{\partial L}{\partial y} \dot{y} + \frac{\partial L}{\partial u} \dot{u} \]  

(3.26)
Next we solve for $\dot{u}$

$$\dot{u} = \left( \dot{L} - \frac{\partial L}{\partial y} \right) \left[ \frac{\partial L}{\partial u} \right]^+$$  \hspace{1cm} (3.27)

where the superscript + represents the Moore-Penrose inverse. And finally, we replace $\dot{L}$ in Eq. (3.27) by its desired value, to yield

$$\dot{u} = \left( \Psi(L) - \frac{\partial L}{\partial y} \right) \left[ \frac{\partial L}{\partial u} \right]^+$$  \hspace{1cm} (3.28)

So the time-integral of this quantity is a command $u$ that drives $L$ to zero according to the dynamics defined by the function $\Psi$ (for this to work, the weightless controller of course needs to know the derivative matrices $\frac{\partial L}{\partial y}$ and $\frac{\partial L}{\partial u}$ which appear in Eq. (3.28) — it learns these by a mechanism that is beyond the scope of my thesis but that resembles implicit supervision, which I discussed in Chapter 2).

When I define $L$ as in Eq. (3.21), based on my hybrid differential error vector, and I feed this $L$ into the weightless-controller formula Eq. (3.28), with a $\Psi$ chosen for soft bang-bang dynamics, then the controller learns to swing the pendulum up and hold it at its zenith, as desired (Figure 3.6), and it performs the movement about 95% as quickly as a strictly time-optimal controller.

To test for controller adaptation, I altered the maximum allowable command. Doing so of course changes the required number of back-swings to reach the zenith: if I allow very strong commands then the controller has the power to drive the pendulum up to its goal
Figure 3.6: With weightless learning and the hybrid differential error (3.20), a controller quickly learns to swing a pendulum up to the zenith and hold it there.
position directly, with no swinging required; if I allow only very weak commands then
the controller has to learn to make a whole series of backward and forward swings before
it can try for the summit. Simulations confirm that the controller learns to adapt to these
changes and makes the required number of swings (Figure 3.7).

3.4 PROS AND CONS OF THE DIFFERENTIAL-ERROR APPROACH

In this chapter, I have devised a nonlinear differential error and hybrid differential error
for roughly time-optimal control. The method is far simpler than any optimal-control
algorithm, but it does have one serious limitation, at least in its present form, and that is
that it doesn’t specify how to choose the coefficients $c_i$ or the nonlinearity $\varphi$ in the
differential error formula Eq. (3.14). So I have to guess. If I choose very small $c_i$’s then
the movements generated by the controller, after it has finished learning, will be very
slow, and therefore very far from time-optimal. We can see this by inspecting (3.14): for
instance if $e_D = 0$ and $c_1$ is small then $\dot{e}$ will be small, i.e. the error will be decreasing
slowly. On the other hand if I choose $c_i$’s very large then I get an $e_D$ that the controller
simply cannot zero, even approximately, and control collapses. So the key question is,
how could a sensorimotor network choose the right coefficients and nonlinearities for its
differential error? To put this another way, how can a controller know what dynamics it
should aim for — what dynamics are the best from among those that are actually
achievable?
Figure 3.7: With weightless learning and the hybrid differential error (3.20), a controller quickly learns to swing a pendulum up to the zenith and hold it there. And it learns the required number of backswings when the maximum allowable command is altered.
One way this might work would be to have these coefficients, and perhaps certain parameters defining the nonlinearity, under feedback control. In the simplest version, we might propose that the $c_i$’s have an inherent tendency to rise — i.e. to push toward faster and faster trajectories — but their ascent is checked when they go too far, i.e. when the adaptive controller is no longer able to drive the resulting $e_D$ close to zero.

I think this plan has some promise, but I won’t pursue it any further in this thesis, because the next chapters, and Chapter 5 in particular, will show that optimal-control theory provides very flexible and powerful methods for solving this problem of identifying optimal dynamics, though these methods also entail greater computational complexity.
Chapter 4

Open-loop optimal control without weight transport

4.1 INTRODUCTION

In the previous chapter I showed how the method of differential errors can produce controllers with some measure of foresight, able to drive a plant to a target without avoidable oscillations or overshoot. But that method did not create optimal trajectories, i.e. given some prespecified cost, the method was unable to produce a controller that minimizes that cost. In this chapter I will show how optimal trajectories might be learned in a brainlike network by applying Pontryagin’s Minimum Principle, which is based in turn on the work of Leonhard Euler in the 18th century.
This chapter will be brief, because I have come to think that Pontryagin’s principle is less promising for sensorimotor applications than methods based on the Hamilton-Jacobi-Bellman approach to optimal control, which I will apply in the next chapter. As I discussed in Chapter 2, Pontryagin’s method yields the optimal trajectory for one specific movement, whereas Hamilton-Jacobi-Bellman yields something much more general: an optimal control law that generates optimal trajectories for a whole range of movements. It seems very likely that the brain needs control laws, but still I think the Pontryagin method is worth considering, because it is computationally simpler and so might provide a relatively easy method in cases where the brain really does want to optimize one highly stereotyped motion.

As I discussed in Chapter 2, the hard thing about optimal control is knowing what to do right now, because a decision that seems sensible now may lead to bad consequences later. For instance, in a rapid elbow flexion it is not at all obvious when exactly to apply the brakes, i.e. when to relax biceps and activate triceps. So the controller needs somehow to deduce, from its stream of feedback, what it should be doing at this moment, or in other words it needs to compute some quantity which, when optimized at each moment, leads to an overall trajectory that minimizes the cost. This crucial quantity has been identified, for a wide range of control problems, through the work of a long series of mathematicians, and it is called the Hamiltonian, $H$.

Recall from Chapter 2 that Hamiltonian is defined like this:

$$H(x, u, t, \lambda) = L(x(t), u(t), t) + \lambda^T(t)[f(x, u, t)]$$  \hspace{1cm} (4.1)
where $x$ is the plant state, $u$ is the command, and $\lambda$ is the costate. We can assume that information about $x$ and $u$ is readily available, so the challenge is calculating $\lambda$. It turns out that we can calculate $\lambda$, $H$, and the optimal control trajectory $u(t)$ all together by solving, simultaneously, these three differential equations:

$$
\dot{x} = \frac{\partial H}{\partial \lambda} = f(x,u,t)
$$

$$
\dot{\lambda} = -\frac{\partial H}{\partial x} = -\left(\frac{\partial f}{\partial x}\right)^T \lambda - \frac{\partial L}{\partial x} \tag{4.2}
$$

$$
0 = \frac{\partial H}{\partial u} = \frac{\partial L}{\partial u} + \left(\frac{\partial f}{\partial u}\right)^T \lambda
$$

The third equation assumes that the command $u$ is unconstrained, or in other words that it can take on any value at any time. If instead the command is constrained (e.g. no commands outside the range –1 to 1) then we must replace the third equation by a more general condition, known as the Minimum Principle, derived by Pontryagin and colleagues:

$$
\forall u \in U \quad H(x^*,u^*,t,\lambda^*) \leq H(x^*,u,t,\lambda^*) \tag{4.3}
$$

Here * denotes optimality and $U$ is the set of admissible commands. So the Principle is saying the optimal command $u^*$ at any moment is the one that makes the Hamiltonian smaller than any other admissible $u$ could go (given that $x(t) = x^*(t)$, i.e. the current state $x(t)$ is exactly what it optimally should be at this moment $t$).
Whether we use version Eq. (4.2) or Eq. (4.3), there is a very general, and relatively simple, way to solve these equations, the sweep method (Bryson and Ho 1975). Here I show how that method can be implemented in a network without weight transport.

4.2 THE SWEEP METHOD WITHOUT WEIGHT TRANSPORT

The scheme is illustrated in the form of a flow diagram in Figure 4.1. For simplicity I will assume that the controller receives $N$ input axons, carrying $N$ scalar inputs $z_0, \ldots, z_{N-1}$, which are phased unit pulses, i.e. input $z_0$ has value 1 during time step 0 and value 0 at all other times; $z_1$ is 1 during time step 1 and is otherwise 0, and so on.

This pulsed arrangement may seem a little artificial, and it is not strictly necessary in order to make the scheme work, but on the other hand there is no apparent reason why neurons couldn’t fire in roughly unit pulses, and this arrangement does simplify the computations and the exposition. In effect, it allows the brain to code variables the same way that a computer would do if it were implementing the sweep method: it would divide up time into small steps and assign a different value of each relevant variable to each time step. In sensorimotor control it is not obvious how large those time steps should be, but intuitively we might expect that about 50 steps per movement might yield enough temporal resolution for, say, a reaching movement.
Figure 4.1: Sweep method implemented in a neural network without weight transport.

The inputs $z_i$ are phased pulses of unit height. The command is a weighted sum of these pulses, so in time interval $i$, the current command $u_i = w_i z_i$. This $u_i$ passes through a forward plant model $f$ to yield an estimate of the plant state in the next time step, $x_{i+1}$. Both $u_i$ and $x_{i+1}$ are gated, by multiplication with the pulse $z_i$, so that in each time step $i$, $u_i$ and $x_{i+1}$ are stored in memory locus $A_{i+1}$ (to avoid clutter, this gating is drawn only for $A_1$). Once all the $u_i$ and $x_{i+1}$ for the whole movement have been stored in memory loci $A_1$ through $A_N$, then $A_N$ can compute $\lambda_N$, pass it to $A_{N-1}$ for use in computing $\lambda_{N-1}$, and so on back up the chain. Once all the $\lambda_i$ have been found, a processor $H$ can compute $\frac{\partial H}{\partial u_i}$ for each time step $i$, and use these gradients to adjust the weight vector $w$. 
Each input $z_i$ is multiplied by a synaptic weight $w_i$ and the products are summed to yield the controller's command $u$, i.e. $u_i = w_i$ during time step $i$. So with this arrangement, the synapses $w_i$ reflect the control trajectory in a very straightforward way.

Learning begins with an initial guess at the appropriate weights, which determine the initial control sequence $u_0, \ldots u_{N-1}$ for the motion in question. The flow diagram shows the motor command $u$ being multiplied by $z_0$ to yield the signal $u_0$, the command at time step 0, which we need to store and use later, during the backward integration which we will use to compute the costate $\lambda$. So the idea is that by multiplying $u$ and $z_0$ we get an output which equals zero at all times except during time step 0, because that is the only time that $z_0$ is not zero. In the flow diagram, the information about $u_0$ is stored in the memory location $A_1$.

In a similar way, the network also stores the state trajectory $x_i$. The motor command $u$ and the initial state $x_0$ are fed into a plant model, called $f$ in the flow diagram, to yield the state $x_1$ at the next time instant (though the current-$x$ input to $f$ is not shown in the diagram, to avoid clutter). This $x_1$ is then gated, by multiplication with by $z_0$, for storage in $A_1$. And similarly, the network stores $u_{i-1}$ and $x_i$ in each $A_i$ down the chain to $A_N$. In the flow diagram, to ease the clutter, I have drawn the input lines and integrators only for $A_1$ and $A_N$, and the inputs to those integrators only for $A_1$, but in reality each memory element $A_i$ would be fed by the same machinery.

So at the end of the movement we have a record of its state and command trajectories stored in the chain of memory elements $A_i$. Given these, it is then possible to compute the
costate trajectory $\lambda_N$, ..., $\lambda_1$ by integrating back up the chain. It is easy to compute $\lambda_N$ (Bryson and Ho 1975). This is then fed to $A_{N-1}$ for use in computing $\lambda_{N-1}$, and so on to $A_1$ and $\lambda_1$. In the end, all these trajectories — control, state and costate — are stored in the $A_i$, coded in neural firing rather than weights, so as to be transmissible. This form of storage is feasible because the data need only be retained for about the duration of a single movement, say about a second, and this is well within the storage duration achievable by reverberating loops of neural activity (Cannon and Robinson 1987). Between movements, the contents of the $A_i$ could be reset to zero (there is a precedent for resettable circuits like this in theories of saccadic control (Scudder 1988)).

With $u_{i-1}$, $x_i$ and $\lambda_i$ all present in $A_i$, we can send these quantities to the operator $H$ to compute the Hamiltonian, or more precisely the partial derivative of the Hamiltonian at time step $i-1$ with respect to $u_{i-1}$, $\partial H_{i-1}/\partial u_{i-1}$, which is the quantity we want to zero, according to Eq. (4.2), to yield the optimal control sequence. We manage this zeroing by sending $\partial H_{i-1}/\partial u_{i-1}$, as an error signal to the weights $w_i$, adjusting them by gradient descent so as to reduce their error (if we instead adopt Eq. (4.3) then the details are slightly different but the principle is the same).

Once the synaptic weighs have been adjusted, we return to time step 1 and begin a new attempt at the same movement, but now performed by the slightly adjusted controller. Over many rounds, the system evolves toward the solutions of Eq. (4.2), and so the $u_i$ converge to the optimal control sequence. Simulations show that this network learns to compute the command for roughly time-optimal elbow flexion: the command is a little
rough near the end of the movement but it shows clearly the squared-off, biphasic pattern required for a fast, accurate movement (Figure 4.2).

4.3 SUMMARY

I have presented a neural network that can learn nearly time-optimal elbow flexions without weight transport based on the Pontryagin’s Minimum Principle. The method works but has its limitations. A major limitation is that it is an open-loop strategy, so it works for just one specific movement and it can’t correct when it is perturbed from its optimal path. But I believe we shouldn’t completely rule out open-loop approaches, because the brain may employ a mix of open and closed-loop control. And there may be advantages to switching between the two strategies: for example, new skills might be learned using a closed-loop approach, but then particular movements may be fine-tuned by an open-loop strategy such as the sweep method.

But I have not pursued this path very far. I have tested the sweep algorithm only on simple sensorimotor tasks like elbow flexions. One future direction might be to test the mechanism on more challenging tasks such as the inverted pendulum (described in
Figure 4.2: The sweep network learns the appropriate biphasic commands for a quick elbow flexion.
Chapter 3) or multijoint arm movements, to see how much time and how many memory elements are required. It also remains to be seen whether the mechanism works with other features besides the unit pulses described in the current version.

The sweep method is my first attempt at devising a learning mechanism to explain how brainlike networks might learn optimal control tasks involving arbitrary performance criteria. In the next chapter, I will present a more versatile mechanism, based on the Hamilton-Jacobi-Bellman approach to optimal control, which seems better suited for biological neural networks.
Chapter 5

Adaptive optimal feedback control without weight transport

5.1 INTRODUCTION

The aim of optimal control is to steer a process, for instance an eye movement or reach, so as to minimize some cost that accumulates over time, such as fuel consumption or time away from a target state. Mathematicians and engineers have devised methods of doing this, as I described in Chapter 2, and they have implemented them in computers. But a fundamental difference between computers and brains is that brains represent information in (at least) two complementary ways: in patterns of action potentials, which are short-lived, metabolically expensive, and transmissible to remote sites; and in synaptic weights, which are enduring, inexpensive, small, and numerous, but not transmissible (i.e. there is no synaptic weight transport), so they influence remote computations only indirectly, via
their effects on the firing of their own postsynaptic cells (Grossberg 1987; Mazzoni, Andersen et al. 1991; Levine 2000; Rolls and Deco 2002). This lack of weight transport constrains the algorithms that can run in biological networks; e.g. it is weight transport that makes the learning algorithm called error-backpropagation unsuitable for the brain — see Chapter 2. Here I describe a mechanism of adaptive optimal control that runs without weight transport.

Surprisingly little work has been done on this problem. Many papers explore the consequences of minimizing specific cost functions but don’t explain how the brain might learn to minimize those costs (Harris and Wolpert 1998; Tweed, Haslwanter et al. 1998; Schreiber, Crawford et al. 2001). Conversely, there are many theories of sensorimotor learning, but most of them don’t construct the trajectory that minimizes a given cost; instead they assume that a desired trajectory has already been specified (Kawato 1990; Porrill, Dean et al. 2004; Shibata, Tabata et al. 2005; Abdelghani, Lillicrap et al. 2008). Others (Sutton, McAllester et al. 2000; Shimansky, Kang et al. 2004; Todorov 2004; Chen-Harris, Joiner et al. 2008) construct optimal paths but make no attempt to cast their algorithms in forms that run without weight transport. The closest approach may be the theory of (Rivest, Bengio et al. 2005), who do learn optimal control without weight transport, but they assume the controller has only a very small number of possible actions, whereas real motor control involves huge repertoires.

So no one knows how the brain creates optimal controllers, but some approaches seem more likely than others. As I have said, the methods used in engineering fall into two
broad classes: those based on Pontryagin’s Minimum Principle and those based on Hamilton-Jacobi-Bellman, or HJB, equations (Sontag 1998), but it is the HJB methods that are better suited for delivering feedback controllers, which are probably what the brain needs (Sontag 1998; Todorov and Jordan 2003). HJB equations can be expressed in discrete or continuous form, but motor systems deal with such finely graded inputs, outputs, and timing that we should focus on the continuous versions. Continuous HJB equations can be solved by grid methods, but this approach can call for astronomical numbers of grid points. So we are left with methods based on sequences of approximations: the policy iteration and policy gradient algorithms (Sutton and Barto 1998; Sutton, McAllester et al. 2000). Of the continuous-time and -state versions of these, perhaps the most efficient, in terms of speed and memory, is the method of \textit{generalized HJB} (GHJB) equations (Saridis and Lee 1979; Beard, Saridis et al. 1997; Lyshevski 1998; Abu-Khalaf and Lewis 2004). I will take it as my starting point.

\section*{5.2 THE METHOD OF GENERALIZED HAMILTON-JACOBI-BELLMAN EQUATIONS}

\subsection*{5.2.1 History}

Saridis and Lee (1979) showed that one can converge on an optimal controller by solving a sequence of linear partial differential equations, the generalized Hamilton-Jacobi-Bellman equations, as I will describe shortly. But their method found little use because the GHJB equations for most control tasks aren’t soluble. In 1997 Beard \textit{et al.} showed how
to find approximate solutions using Galerkin methods. Starting in 2004, Abu-Khalaf and Lewis adapted the method to handle control constraints, using a novel loss function defined for that purpose by Lyshevski in 1998. But none of these methods was suited for the brain, as I will show here.

5.2.2 Cost-to-go

The GHJB method, like most approaches to optimal control, depends on the idea of cost-to-go functions or some variant such as action-value or Bellman functions (Sutton and Barto 1998; Todorov 2004; Abu-Khalaf and Lewis 2005; Todorov 2005). To define the cost-to-go, we begin with the plant equation describing the process we want to control,

\[ \dot{x} = f(x) + G(x)u(x) \]  

which I will usually write as \( \dot{x} = f + Gu \). Here \( x \) is the state of the controlled process, or plant, and \( u \) is the control signal or command, which is computed from \( x \) by a function called a control law, or controller. The aim of optimal control is to find the controller that minimizes some cost

\[ C = \int_{0}^{\infty} L(x,u)dt \]  

where \( L \) is a nonnegative scalar function called the cost rate or loss, e.g. \( L \) might be the distance from the gaze point or hand to a target. Given any controller we define its cost-to-go:
\[ J(x) = \int_{t}^{\infty} L(x, u) dt = \int_{t}^{\infty} -L(x, u) dt \] (5.3)

So \( J(x) \) is the total cost we will accumulate from now on if we are currently (at time \( t \)) in state \( x \) and we choose our commands using controller \( u \).

### 5.2.3 Importance of the cost-to-go

The cost-to-go function measures the quality of a controller: the more nearly optimal the controller, the lower its cost-to-go. More than this, it can be shown that if we know the cost-to-go function \( J^{(n)} \) of one controller \( u^{(n)} \) then we can define a better controller \( u^{(n+1)} \).

Specifically, we define \( u^{(n+1)} \) to be the minimizer of \( \frac{dJ^{(n)}}{dx} \dot{x} + L \), i.e.

\[
\forall x \ u^{(n+1)}(x) = \arg\min_{u \in U} \frac{dJ^{(n)}}{dx} \left[ f(x) + G(x)u^{(n)}(x) \right] + L(x, u^{(n)}(x))
\] (5.4)

where \( U \) is the set of permissible control signals, e.g. if \( u \) is a number between –1 and 1 then \( U = [-1, 1] \). (Note that \( dJ^{(n)}/dx \), being the gradient of a scalar function, is a row vector, and therefore the multiplication in (5.4) makes sense dimensionally.)

We can find the minimizer in Eq. (5.4) by differentiating with respect to \( u^{(n)} \) and solving for the \( u \) that zeroes this derivative (or if \( u \) is constrained we can similarly differentiate the right-hand side and use gradient projection (Kirk 1970). For example, if \( L = x^{T}Qx + \)
\( u^T u / 2 \), for some matrix \( Q \), then the derivative with respect to \( u^{(n)} \) of the right-hand side of Eq.(5.4) is \((d f^{(n)}/dx) G + u^T\), and so the minimizer is

\[
\begin{align*}
    u^{(n+1)}(x) = - \left[ \frac{d f^{(n)}}{dx} G \right]^T
\end{align*}
\]

(5.5)

We can repeat the process, using \( f^{(n+1)} \) to find a still-better controller \( u^{(n+2)} \), and so on until we converge to the optimal controller \( u^* \) (see Section 5.2.5). To get started we need an initial controller \( u^{(1)} \), but it needn’t be anywhere near optimal, so creating it is relatively easy. The hard part is finding the cost-to-go.

5.2.4 Approximating the cost-to-go

There are a handful of basic ways to do this (Bryson and Ho 1975; Sontag 1998). In GHJB, we use the fact that, while the cost-to-go itself may be unknown, its time derivative is readily available: differentiating Eq.(5.3), we get \( \dot{J} = -L \). And of course this relation holds in every round \( n \) of the algorithm, i.e.

\[
\dot{J}^{(n)} = -L
\]

(5.6)

So assuming we know the loss \( L \), we can use it to train an approximator of \( \dot{J}^{(n)} \) by supervised learning. For instance we can compute a feature vector \( \theta(x) \) — a vector of nonlinear functions of the state \( x \) — and then differentiate it with respect to time to get \( \dot{\theta} \),
and find the row vector of weights $w$ that minimizes the sum of squared approximation error over some data sample of $N$ examples:

$$SSE = \sum_{i=1}^{N} \left( w\theta(x_i) - \hat{J}^{(n)}(x_i) \right)^2 = \sum_{i=1}^{N} \left( w\theta(x_i) + L(x_i, u(x_i)) \right)^2 \quad (5.7)$$

This is easily done by least-squares fitting. The resulting optimal $w$ — the weight vector that yields the best approximation to $\hat{J}^{(n)}$ — I will call $w^{(n)}$.

Once $w^{(n)}$ is found, we have of course an approximation to $\hat{J}^{(n)}$, namely $w^{(n)}\theta$, but we have also an approximation to $J^{(n)}$ itself, namely $w^{(n)}\theta$, and most importantly we have an approximation to $dJ^{(n)}/dx$,

$$\frac{d\hat{J}^{(n)}}{dx} = w^{(n)} \frac{d\theta}{dx} \quad (5.8)$$

This estimate is crucial because, as we see from Eq.(5.5), we need $dJ^{(n)}/dx$ to create the new, improved controller $u^{(n+1)}$. And while Eq. (5.5) applies to just one specific loss function, it is easy to see from Eq. (5.4) that, whatever the loss, the formula for the minimizer will include $dJ^{(n)}/dx$. So the GHJB method learns $\hat{J}^{(n)}$, derives an estimate of $dJ^{(n)}/dx$, and uses that to create $u^{(n+1)}$.

A few examples will reveal a pattern which has important biological consequences. Suppose our feature vector $\theta$ is based on the log-cosh function (the antiderivative of the
hyperbolic tangent, tanh) — i.e. \( \theta(x) = \log(\cosh(W_0 x)) \), where \( W_0 \) is a matrix. Then Eq. (5.5) becomes

\[
\begin{align*}
\mathbf{u}^{(n+1)}(x) &= -\left[ \frac{d\hat{f}^{(n)}}{dx} G \right]^T \\
&= -\left[ w^{(n)} \frac{d\theta}{dx} G \right]^T \\
&= -\left[ w^{(n)} \text{diag}(\tanh(W_0 x)) W_0 G \right]^T \\
&= -G^T W_0^T \text{diag}(\tanh(W_0 x)) w^{(n)^T}
\end{align*}
\]

where \( \text{diag}(\tanh(W_0 x)) \) is a square matrix with the vector \( \tanh(W_0 x) \) on its main diagonal and zeros elsewhere. Now suppose instead that \( \theta \) is based on polynomial kernels (Schölkopf and Smola 2002) — i.e. \( \theta(x) = (W_0 x + o)^p \), where \( o \) is a column vector of \( n_\theta \) ones (\( n_\theta \) being the number of elements in \( \theta \)), \( p \) is the degree of the polynomials, and the exponentiation is done element-wise. Then Eq. (5.5) becomes

\[
\begin{align*}
\mathbf{u}^{(n+1)} &= -G^T W_0^T \text{diag}(p [W_0 x + o]^{p-1}) w^{(n)^T}
\end{align*}
\]

Or again, if \( \theta \) is based on Gaussian kernels (Schölkopf and Smola 2002) — i.e. \( \theta_i(x) = \exp(\nu \Sigma (W_{0ij} - x_j)^2) \), where \( \nu \) is a scalar constant — then Eq. (5.5) becomes
\[ u^{(n+1)}(x) = -G^T(xo^T - W_\theta^T)\text{diag}(\theta)w^{(n)T} \]  

(5.11)

The thing to notice is that in each of these examples, owing simply to the chain rule, the vector \( w^{(n)T} \) and the matrix \( W_\theta^T \) appear in the formula for \( u^{(n+1)} \). I will trace the biological implications shortly, in the section GHJB without weight transport.

5.2.5 Why the GHJB algorithm converges to the optimal controller

The GHJB method creates a sequence of controllers whose limit is the optimal controller \( u^* \). Given a stabilizing controller \( u^{(n)} \) we generate \( u^{(n+1)} \) in two steps. First we find the cost-to-go function \( J^{(n)} \) associated with \( u^{(n)} \) by solving the GHJB equation for \( J^{(n)} \):

\[ \frac{dJ^{(n)}}{dx} [f + Gu^{(n)}] + L = 0 \]  

(5.12)

Then we define \( u^{(n+1)} \) to be the controller that minimizes the quantity

\[ \frac{dJ^{(n)}}{dx} [f + Gu^{(n+1)}] + L = \dot{J}^{(n)} + L \]  

(5.13)

at each moment. Because \( u^{(n+1)} \) minimizes \( \dot{J}^{(n)} + L \) rather than just zeroing it as did \( u^{(n)} \), we can see that \( u^{(n+1)} \) is a better controller than \( u^{(n)} \): it yields an equal or smaller loss \( L \) for any given \( \dot{J}^{(n)} \) than did \( u^{(n)} \). In other words \( u^{(n+1)} \) drives \( x \) through the contours of \( J^{(n)} \) with less loss than did \( u^{(n)} \). It follows, then, that the cost-to-go function of \( u^{(n+1)} \) is
everywhere less than or equal to that of $u^{(n)}$:

$$\forall n \in N \quad \forall x \in X \quad J^{(n+1)}(x) \leq J^{(n)}(x) \quad (5.14)$$

Clearly no controller has a cost-to-go lower than that of the optimal controller, $J^*$, so for each $x$, the sequence $J^{(n)}(x)$ is bounded below by $J^*(x)$, and so $J^{(n)}$ converges to a limit, $J^{(\infty)}$. Furthermore, it is clear that this limit is $J^*$: recall that we generate the sequence $J^{(n)}$ by solving, alternately, Eqs. (5.12) and (5.13), so the limiting controller $u^{(\infty)}$ satisfies both equations, i.e. $J^{(\infty)}$ is the cost-to-go of $u^{(\infty)}$ but also $u^{(\infty)}$ minimizes

$$\frac{dJ^{(\infty)}}{dx} \left[ f + Gu^{(\infty)} \right] + L = J^{(\infty)} + L \quad (5.15)$$

That is, no other controller can drive $x$ down through the contours of $J^{(\infty)}$ with less loss than does $u^{(\infty)}$, so no controller has a lower cost-to-go, and so we must have $u^{(\infty)} = u^*$ and $J^{(\infty)} = J^*$. A more detailed proof is given in (Abu-Khalaf and Lewis 2005).

### 5.3 GHJB WITHOUT WEIGHT TRANSPORT

My aim is to devise a method of adaptive optimal control that runs entirely in neural networks without weight transport. It is a challenge because any candidate algorithm, if it
has any flexibility, will involve large arrays of parameters for creating features and for adjustment during learning. In the brain, we want to store these arrays as synaptic weights rather than as firing rates — because otherwise we would need vastly more space and energy and huge numbers of neurons. The stored values of course have to enter into calculations, meaning they have to interact with other variables, so the representations of these variables have to be brought together physically. Given almost any algorithm, if we implement it straightforwardly we find ourselves transporting synaptic weights from place to place in the network. But in the brain, those weights can’t be transported; they can influence remote calculations only via their effects on cell firing — see Chapter 2. My aim is to recast the algorithm so it respects this constraint.

The GHJB algorithm, implemented straightforwardly, calls for weight transport at three stages of the calculation. In this section I consider all three in turn and show how we can remove the need for weight transport.

### 5.3.1 Learning the cost-to-go

The synaptic weight vector \( \mathbf{w} \) in Eqs. (5.7) and (5.8) must be adjusted, by some sort of learning algorithm, to yield approximations to \( \hat{J}^{(n)} \) (i.e. \( \mathbf{w} \) must be driven toward \( \mathbf{w}^{(n)} \)). In (Abu-Khalaf and Lewis 2005) \( \mathbf{w} \) is adjusted by batch learning, which involves the rapid inversion of a very large matrix. In the brain, that matrix would have to be represented as an array of synapses, and the inversion would call for weight transport.
We can solve this problem simply by switching to an on-line learning algorithm. I propose that the weight vector \( w \) multiplies the time-derivative of our feature vector \( \theta \) to yield an estimate of \( \hat{J}^{(n)} \):

\[
\hat{J}^{(n)} = w \dot{\theta}
\]  

(5.16)

Anatomically, this means that a set of axons, \( n_\theta \) in number, carries the signal \( \dot{\theta} \) to a neuron with \( n_\theta \) synapses, whose weights are the \( n_\theta \) elements of \( w \). Each component \( \theta_i \) is multiplied by its corresponding synaptic weight \( w_i \), and the products are summed in the neuron to yield the output firing rate, which represents our estimate \( \hat{J} \). This estimate is compared with the true value of \( \dot{J} \), which we know equals \(-L\), to yield an error signal:

\[
e = \hat{J} - \dot{J} = w \dot{\theta} + L
\]  

(5.17)

And \( e \) is used to adjust \( w \), by some appropriate online learning rule, so as to improve the estimate of \( \hat{J} \).

The simplest such learning rule would be the least mean square formula, or LMS (Haykin 2002), which drives \( w \) down the gradient of the squared error:
where $\eta$ is a positive number called the rate constant, and the third equality follows from Eq. (5.17). The fourth equality follows because the partial derivative $\partial L / \partial w$ is 0: $w$ affects only the estimate $\hat{J}$, not the controller or the plant, though after $w$ has converged to $w^{(n)}$ it will be copied over to the new controller for the next round — see Section 5.3.2. Over time, this rule is guaranteed to drive $w$ to its optimal value, $w^{(n)}$, so long as the neuron in Eq. (5.16) receives a varied stream of input vectors $\Theta$. In Eq. (5.18) two terms on the right-hand side, $e$ and $\dot{\Theta}$, are functions of $x$, which means that the synaptic change $\dot{w}$ depends on the state, as you would expect. Equation (5.18) implies that the learning rate depends in a very simple way on the rate constant: the larger the $\eta$, the faster the learning. But in reality, noise and delays mean that an overly large $\eta$ can cause instability, so there is an optimal value of $\eta$, not too large and not too small, which depends on the statistical properties of the input vector $\dot{\Theta}$ (Werfel, Xie et al. 2005). It is possible that the brain monitors those statistics and selects a suitable $\eta$.

But a better plan may be to let $\eta$ vary depending on the input, as in the learning rule called normalized least mean square, or NLMS (Haykin 2002), which is just like LMS except that $\eta = 0$ when $\|\dot{\Theta}\|$ is small and otherwise
This rule is scarcely more complex than LMS itself, but it converges faster without having to worry so much about input statistics.

For even faster learning we could use the rule called recursive least squares, or RLS (Haykin 2002), which is a simple form of Kalman filter. It is more complicated than LMS or NLMS, but still it appears biologically feasible: Kawato and colleagues have argued in its favor (Shibata, Tabata et al. 2005), and it needs no weight transport.

In what follows, I will call the network in Eq. (5.16) the model, i.e. the model is the structure that adjusts its synapses \( \mathbf{w} \), by whatever learning rule, to approximate \( \hat{J} \). And with this terminology, we are ready to consider a second and less easily removable instance of weight transport in GHJB.

### 5.3.2 Creating the new controller

How does the model communicate what it has learned to the controller? The synapses \( \mathbf{w} \) in the model Eq. (5.16) evolve by learning until they reach, or at least approach, their optimal values \( w^{(n)} \). Shortly thereafter, \( w^{(n)} \) appears in the formula for the new improved controller \( u^{(n+1)} \), as shown in the example Eqs. (5.9), (5.10) and (5.11). Clearly model and controller are distinct structures: the model receives \( \hat{\theta} \) and puts out \( \hat{J} \); the controller’s \( w^{(n)} \) acts on different inputs, as in Eqs. (5.9), (5.10) and (5.11), and puts out commands. And model and controller have to coexist, doing their separate jobs at the
same time: during each round of the GHJB algorithm, the current controller $u^{(n)}$ is emitting commands based on the weight vector $\mathbf{w}^{(n-1)}$ that was learned in the previous round, while the model is driving its $\mathbf{w}$ toward the value $\mathbf{w}^{(n)}$ that it will use to create the next controller.

So how could $\mathbf{w}^{(n)}$ make the trip from model to controller at the end of each round? In a computer we can simply store a copy of $\mathbf{w}^{(n)}$ and use it to calculate commands. But in the brain there is no known way to copy the values of one set of synapses, in the model, onto another set, in the controller.

A possible solution is shown in Figure 5.1. For concreteness I will explain the mechanism using example Eq. (5.9), but the story is similar for Eqs. (5.10) and (5.11) or any other choice of features or loss. The key idea is that $\mathbf{w}^{(n)}$ needn’t be transported anywhere if the model network that learns it can become the controller. But to make this possible, we have to reorganize model and controller so they have the same inputs and structure. To begin with, observe that if we define

$$\varphi = \tanh(W_\theta x)$$  \hspace{1cm} (5.20)

then we can rewrite the controller Eq. (5.9) in the following form:

$$u^{(n+1)}(\mathbf{x}) = -G^T W_\theta^T \text{diag}(\varphi) \mathbf{w}^{(n)T}$$

$$= -G^T W_\theta^T \text{diag}(\mathbf{w}^{(n)}) \varphi$$  \hspace{1cm} (5.21)
Figure 5.1: Network implementation of the GHJB algorithm. (A) The upper branch is a controller. The lower loop learns $dJ^{(n)}/d\mathbf{x}$, the gradient of the current controller's cost-to-go, by adjusting synaptic weights $\mathbf{w}'$; the caret over $dJ^{(n)}/d\mathbf{x}$ indicates an estimate. When the learning is done, the values of $\mathbf{w}'$ are copied over to the vector $\mathbf{w}$ in the upper branch to create the new, improved controller. But in the brain, how can one set of synaptic weights be copied onto another? And how can the weight matrices $\mathbf{W}_\theta^\top$ be set equal to the transpose of the anatomically separate weight matrix $\mathbf{W}_\theta$? I suggest that the transposes can be constructed by learning, and that $\mathbf{w}'$ needn't be copied to $\mathbf{w}$ if their output signals are redirected instead. (B) At first, $\mathbf{w}$ feeds the controller while $\mathbf{w}'$ learns. Small black disks are switches; dotted lines are suppressed pathways. (C) Once $\mathbf{w}'$ has converged to $\mathbf{w}'^{(n)}$, we begin round $n + 1$: the switches flip, so now $\mathbf{w}'$ becomes part of the controller $\mathbf{u}^{(n + 1)}$ while $\mathbf{w}$ learns. There is no need to copy weights from one set of synapses to another; instead the synapses swap roles. And the alternation continues, the controller improving with each round.
Further, we can rewrite the model Eq. (5.16)

\[
\hat{j}^{(n)} = w \hat{\theta}
\]

\[
= [w \hat{\theta}]^T
\]

\[
= \left[ w \frac{d\theta}{dx} \hat{x} \right]^T
\]

\[
= [w \text{diag}(\varphi) W_\theta \hat{x}]^T
\]

\[
= [\varphi^T \text{diag}(w) W_\theta \hat{x}]^T
\]

\[
= \hat{x}^T W_\theta^T \text{diag}(w) \varphi
\]

Recast this way, the controller, in Eq. (5.21), and the model, in Eq. (5.22), now receive the same input vector, \( \varphi \). And in both networks the initial processing is similar: each element of \( \varphi \) is multiplied by the corresponding element of \( w^{(n)} \) or \( w \), and the resulting vector passes through the matrix \( W_\theta^T \). From Eqs. (5.9) and (5.21) we know that

\[
W_\theta^T \text{diag}(w^{(n)}) \varphi = \left[ \frac{d\hat{j}^{(n)}(x)}{dx} \right]^T
\]

so we can write
\[ u^{(n+1)}(x) = -G^T \left[ \frac{d\hat{J}^{(n)}}{dx} \right]^T \] 

(5.24)

and

\[ \hat{J}^{(n)} = x^T \left[ \frac{d\hat{J}^{(n)}}{dx} \right] \] 

(5.25)

keeping in mind that in Eq. (5.24), the estimate of \( d\hat{J}^{(n)} / dx \) is the optimal one, based on the optimal weight vector \( w^{(n)} \), whereas in Eq. (5.25), \( \hat{J}^{(n)} \) and \( d\hat{J}^{(n)} / dx \) are estimates-in-training, based on the vector \( w \) which is still evolving toward \( w^{(n)} \).

Now that controller and model share the same structure, it is a simple matter to let them swap roles, as shown in Figure 5.1. Here we have a branching network, each branch containing a \( w \)-sized array of synapses; call one array \( w \) and the other \( w' \). In round \( n \) of the algorithm, one of the two arrays learns \( w^{(n)} \). The other array, its learning switched off, stores the weight vector \( w^{(n-1)} \) that it learned in the previous round, for use in this round’s controller. At the finish of the round, the arrays swap roles: learning is switched off in one (e.g. by setting its error signal \( e \) to 0) and switched on in the other. The switching means that in round \( n \), a set of synapses evolves toward \( w^{(n)} \) starting from \( w^{(n-2)} \), the values it learned two rounds back, but these starting values don’t matter because all of my proposed learning rules — LMS, NLMS, and RLS — are linear in the parameters, so
they converge to their optimal weights regardless of where they start (Haykin 2002; Farrell and Polycarpou 2006).

The chief novelty here is the role-swapping. In most adaptive-control schemes in engineering or neuroscience there is one structure that serves as a model and another that generates commands. But in Figure 5.1, two branches alternate between modeling and control, so at each round the new \( \mathbf{w}^{(n)} \) appears in the new controller with no need for weight transport. And unlike weight transport, switching can work with mechanisms that are known to exist in the brain: it just means suppressing signals along certain pathways and disinhibiting others, e.g. by presynaptic inhibition or by inhibiting relay neurons. Known biological examples are the way omnipause neurons switch the saccadic system on and off (Nakao, Curthoys et al. 1980), the way vestibulo-ocular pathways are rapidly switched off and then on again during eye-head gaze shifts (Laurutis and Robinson 1986; Guitton and Volle 1987; Tweed, Haslwanter et al. 1998), and possibly the action of subthalamic nucleus neurons in switching between different kinds of eye movement (Isoda and Hikosaka 2008) and the role of posterior parietal cortex in switching between verbal tasks (Gurd, Amunts et al. 2002).

### 5.3.3 Computing \( d\theta/dx \)

There remains one final instance of weight transport to purge from the algorithm. At the core of the GHJB method is the fact that the quantity \( \hat{\mathbf{j}}^{(n)} \) is readily available to the model, by Eq. (5.6), and so it is possible to learn an approximation \( \hat{\mathbf{j}}^{(n)} = \mathbf{w}^{(n)} \hat{\theta} \) and then derive
an estimate of $dJ^{(n)}/dx$: $d\hat{J}^{(n)}/dx = w^{(n)}d\theta/dx$. To compute $d\theta/dx$, though, we need the matrix $W_\theta^T$, as shown in example Eqs. (5.9), (5.10) and (5.11). So we need $W_\theta$ to compute $\theta$ (or $\phi$), and we need its transpose, $W_\theta^T$, to compute $d\theta/dx$. Clearly these are two different arrays of synapses: for $W_\theta$, we need $n_x$ synapses on each of $n_\theta$ neurons; for $W_\theta^T$, $n_\theta$ synapses on each of $n_x$ cells. In the brain, how could one array of synapses be made equal to the transpose of another?

It could be done by learning. One of the arrays might be filled in with random weights (or by some more sophisticated method) and the second array could learn to be its transpose. Putting the problem in general terms, suppose $W_A$ is any $n$-by-$m$ array of weights, $W_B$ is $m$-by-$n$, $x_A$ and $x_B$ are their input vectors, $y_A$ and $y_B$ are the corresponding outputs. Then $W_B$ can learn to equal $W_A^T$, without weight transport, by the learning rule

$$\dot{W}_{bij} = -\eta (y_{bi}x_{ai} - x_{bj}y_{aj}) x_{ai}x_{bj}$$

where $\eta$ is a positive constant). See Figure 5.2. For more on the role of the transpose in optimal control, see Chapter 6.
**Figure 5.2:** Learning the transpose of s synaptic weight matrix. (A) $W_A$ is an array of 6 synapses: 2 on each of 3 neurons; $W_B$ is 3 synapses on each of 2 cells; $x_A$ and $x_B$ are firing rates on the input axons to these arrays; $y_A$ and $y_B$ are their outputs. To illustrate the learning rule of Eq. (5.26), I at first assume the $x$’s are cardinal, unit vectors, i.e. one element equals 1 and the rest are 0’s. If $x_A$ has 1 (filled circle) as its first element and 0 (open circle) as its second then $y_A$ equals the first column of $W_A$, e.g. its second element $y_{A2} = W_{A21}$. Similarly if $x_B = (0, 1, 0)^T$ then $y_{B1} = W_{B12}$. So we can drive $W_{B12}$ toward $W_{A21}$ by defining an error $E = y_{B1} - y_{A2}$ and setting $\dot{W}_{b12} = -\eta Ex_{b2}$. This rule becomes Eq. (5.26) if we add extra terms that allow the $x$’s to be arbitrary vectors rather than unit cardinals. (B) This learning rule doesn’t require that the inputs be cardinal vectors, with all but one element equal to 0, but it converges faster if they are. For this simulation, $W_A$ is 5-by-3 and $W_B$ is 3-by-5. On the ordinate, transposition error $e_T$ is the sum of the squared elements of $W_B - W_A^T$. The plot shows that $e_T$, averaged over 100 trials of $10^4$ timesteps each, declines faster when the input vectors are cardinal (thin black line) than otherwise (thick gray).
**Figure 5.3:** Bipartite synapses might reduce the number of neurons for the GHJB algorithm. (A) shows how a network could use $n_\theta$ neurons (open circles) to multiply the signal $\phi$ by the adjustable weights $w$ and convey the product $\text{diag}(w)\phi$ to the weight matrix $W_\theta^T$ for use in Eq. (5.21). But with bipartite synapses it doesn't need those cells. The elements storing $w$ could lie upstream from where the axon branches, as in (B), or downstream as in (C), or even further downstream, with $w$ and $W_\theta^T$ both on the postsynaptic cell. These latter schemes somewhat resemble cerebellar cortex. For $W_\theta$ we need very many neurons (represented by the small black cells in the figure), each with relatively few inputs — like cerebellar granule cells. For $w$ and $W_\theta^T$ we need fewer neurons (the large gray cells), each with many inputs — like Purkinje cells. In one implementation of the algorithm, $W_\theta^T$ could be laid down in the granule cells and transposed by Eq. (5.26) to postsynaptic elements on Purkinje cells. Those same cells would then adjust their upstream weights, $w$, learning to compute $dJ/dx$ for transmission to controllers.
5.3.4 Overview of the algorithm

By the methods in the last three subsections, I have devised a mechanism for adaptive optimal control that works without weight transport. Here I gather together the equations for the algorithm, assuming a log-cosh feature vector $\theta$ and quadratic loss $L = x^T Q x + u^T R u$. Modifications for other features and losses are straightforward.

\[
\nu = W_o x \tag{5.27}
\]

\[
\theta = \log(\cosh(\nu)) \tag{5.28}
\]

\[
\phi = \text{vec}\left(\frac{d\theta}{dv}\right) = \tanh(\nu) \tag{5.29}
\]

\[
\left[ \frac{d\hat{j}^{(n)}}{dx} \right]^T = W_o^T \text{diag}(w) \phi \tag{5.30}
\]

\[
e = \frac{d\hat{j}^{(n)}}{dx} \dot{x} + L \tag{5.31}
\]

\[
\dot{\theta}_i = -\eta e \dot{\phi}_i = -\eta e \phi_i \frac{\phi_i}{1 - \phi_i^2} \tag{5.32}
\]
\[ u^{(n+1)} = - \left( \frac{1}{2} R^{-1} G^T \left[ \frac{d\hat{f}^{(n)}}{dx} \right] \right) \]  

(5.33)

And we need something like Eq.(5.26) to set up the transpose weight matrix \( W_\theta^T \).

In Eq. (5.29), \( d\theta/d\nu \) is a diagonal matrix and \( \text{vec}(d\theta/d\nu) \) is the vector composed of the elements on its main diagonal, so the dimensions of \( \text{vec}(d\theta/d\nu) \) agree with those of \( \varphi \) and \( \tanh(\nu) \).

In Eq. (5.32) I give three versions of the LMS learning rule, Eq. (5.18). All three are equivalent, as a brief calculation shows, but with the first version, on the left, the network has to compute the signal \( \dot{\theta} \) and convey it to the adjustable synapses \( w \) on a set of \( n_\theta \) axons; with the second version the network needn’t ever compute \( \theta \) or \( \dot{\theta} \) (so Eq. (5.28) drops out of the algorithm) but it does need to compute \( \dot{\nu} \) and carry that in a set of \( n_\theta \) axons; the third version, on the right, is more efficient: it needn’t compute \( \theta \), \( \dot{\theta} \), or \( \dot{\nu} \), because the adjustable synapses exploit the fact that all the information they need for their learning is present in just two inputs, \( e \) and \( \varphi \). In this third scheme, \( \nu \) is needed only to compute \( \varphi \), so Eqs. (5.27) and (5.29) can be carried out by a single set of \( n_\theta \) cells, saving more neurons. In version three the learning rule is more complicated, but only slightly, and it needn’t be exact: \( w \) will converge to \( w^{(n)} \) eventually even if the learning rule is as simple as \( \dot{w}_i = -\eta e \varphi_i \hat{\varphi}_i \).
Two of my learning rules — Eq. (5.26) and the second version of Eq. (5.32) — make synaptic change depend on three or four variables. In the literature, two-signal rules are more common, but three- or four-factor rules have also been proposed many times, and call for nothing fundamentally new (Mazzoni, Andersen et al. 1991; Wolpert and Kawato 1998; Abdelghani, Lillicrap et al. 2008).

One more point about implementation: notice that in Eq. (5.30) the matrix $W_{\theta}^T$ appears right next to $\text{diag}(w^{(n)})$ or $\text{diag}(w)$. Anatomically, this could mean that the $w$ (or $w^{(n)}$) synapses sit on one layer of neurons and the $W_{\theta}^T$ on the next downstream layer. But we could eliminate one whole layer of cells if we packaged the elements of $W_{\theta}^T$ and $w$ together in single synapses, i.e. if we supposed that the overall strength or “weight” of a synapse could be a product of two different factors, independently adjustable. These factors might be pre- and postsynaptic structures, or either one of them might lie outside the synapse, in the voltage-transmission properties of the upstream axon or downstream dendrite, so there are plenty of possibilities. This notion of bipartite synapses is speculative, but it would be extremely useful, allowing computations to proceed with fewer neurons (see Chapter 6). And it is likely that synapses, dendrites and the sites of learning are more complex than single, scalar weights (Lisberger and Sejnowski 1992; Spruston and Kath 2004; Fusi, Drew et al. 2005). Figure 5.3 shows how bipartite synapses might reduce the number of neurons needed for the GHJB algorithm.
SCOPE OF THE ALGORITHM

My scheme applies to linear and nonlinear plants, so long as they are affine in $u$, and it can be generalized to the non-affine case by the usual method of treating $\dot{u}$ as the command and considering $u$ itself as part of the plant state. The controller needs to know $f$ and $G$ from the plant Eq. (5.1), but as I described in Chapter 2, this knowledge can be obtained without weight transport (Abdelghani, Lillicrap et al. 2008). In this section I provide some simulations to show that the method is more widely applicable than it may appear at first.

Figure 5.4 shows a linear, second-order plant with one degree of freedom learning to minimize the time-integral of the loss function $x^T x + u^T u$. Matlab code for this simulation is included in Appendix C. In all three panels, line styles indicate stages of the algorithm: dotted lines are traces created by the initial, far-from-optimal controller; dashed lines are created by the second, improved controller; and solid lines are due to the final controller. Figure 5.4A plots plant state versus time: position in black and velocity in gray. The dotted lines show that the initial controller drives the plant toward its target state of 0, but doesn’t get very close within the 800-ms time window plotted here. Figure 5.4B shows that the accumulated cost for this controller, at the 800-ms mark, is about 115. Figure 5.4C shows that the controller emits only very small commands, which is why it takes so long to approach its goal. The dashed lines in all three panels show that the second controller is much better: it carries the plant promptly to its goal, and lowers the cost to about 80. Solid lines show that the final controller, after convergence, brings the cost
**Figure 5.4:** GHJB algorithm learns to drive a linear second-order plant to a zero target minimizing a quadratic loss function. Rows A–C show state, cost, and command $u$ versus time for one sample movement; in row A, black lines (of all styles — dotted, dashed, solid) represent position; gray lines, velocity. Here and in subsequent rows, dotted lines represent trajectories created by the initial controller $u^{(1)}$, dashed lines those of the second controller $u^{(2)}$, and solid lines those of the final controller in the run, $u^{(5)}$. Learning takes about five rounds of NLMS with 10 log-cosh features and 100 teaching examples per round. Matlab code is provided in the *Appendix*. 


down to about 30. The algorithm converges in 5 rounds and needs just 10 log-cosh features.

The simulations in Figure 5.5 are similar, but the loss function is now \( \text{tanh}(Qx^T x) \), \( Q = 100 \), which is useful for creating roughly time-optimal — i.e. quick — movements (Abu-Khalaf and Lewis 2004). Here I have constrained the command \( u \) to lie between –1 and 1 to show that the algorithm can handle control constraints (and because rapid motion isn’t challenging unless the commands are bounded). In this case the controller achieves essentially optimal performance in just one round. The cost has not fallen as dramatically as in Figure 5.4, but the solid trajectory is markedly faster than the initial, dotted one, and the commands show the biphasic profile needed for quick movements. Owing to a more-complex cost-to-go, this learning task requires more features than the one in the previous figure — about 100.

In Eq. (5.3) I defined the cost-to-go \( J \) as a function of the plant state \( x \) alone, apparently ruling out cases where the cost depends not only on the state but also on some sort of target, as in goal-directed eye or hand movements. But Figure 5.6 shows that we can handle those cases simply by redefining the plant state, augmenting it by adding extra components to represent the goal. These simulations show a movement from an initial position of -0.5 to a target at 0.5, with the initial and target velocities equal to 0. A feeble initial controller moves the system toward the target but doesn't reach it in the plotted time window. The second controller reaches the target promptly, achieving a much lower cost, and the final controller is slightly better still.
Figure 5.5: GHJB algorithm learns roughly time-optimal control for a linear second-order plant, with $L = \tanh(100x^\top x)$ and $|u| \leq 1$. Learning takes one round of RLS with 100 log-cosh features and 1000 examples per round.
Figure 5.6: GHJB algorithm learns to drive a linear second-order plant to nonzero targets

The augmented state is \((x, x_1^*)\) where \(x_1^*\) is the target position; \(L = (x_1 - x_1^*)^2 + u^2\).

Learning takes about five rounds of RLS with 50 log-cosh features and 500 examples per round. (A) Black dotted, dashed, and solid lines represent position as it evolves under the first, second, and final controllers, respectively. Gray lines represent velocity. (B) Costs accrued by the three plotted controllers. (C) Commands delivered by the three controllers.
Similarly, Figure 5.7 shows that we can incorporate time into an augmented plant state, to deal with problems where the plant dynamics or the losses depend not only on the (unaugmented) state but also on some sort of clock. Here the loss function is \( \exp(-A[T-t]^2)Qx^T x + Ru^T u \), where \( A, Q \) and \( R \) are positive constants, \( t \) is time, \( T \) is the desired duration of the movement, and \( x \) is the unaugmented state. So the idea is that we want to drive \( x \) to zero, and we want to conserve fuel (hence the \( Ru^T u \) term), but we don’t care about our distance from the goal state of \( 0 \) until \( t \) approaches the desired movement duration \( T \). In this simulation, \( T \) is 0.5 s. The initial controller expends very little fuel, and so makes only slow progress to its goal. Early in the movement this strategy seems to be paying off — the cost stays very low. But as \( t \) approaches \( T \), the loss function begins to worry about the long distance remaining to the target, and so the cost mounts rapidly. By the second round, the controller has realized that it has to move quickly to the target, burning fuel early on to avoid positional costs later. The optimal controller fine-tunes the timing, arriving a little later to save fuel and achieve a slightly lower cost overall.

Finally, Figure 5.8 shows that the algorithm can handle tasks involving kinematic redundancy, meaning ones where the plant has more degrees of freedom than are strictly needed for the job. Here the plant is the planar two-link manipulator, a nonlinear fourth-order system with two degrees of freedom, like a two-joint arm with shoulder and elbow bending in a plane (Lewis, Jagannathan et al. 1999). As in Figure 5.6 we can incorporate desired joint coordinates in an augmented state vector and so teach this arm to reach for targets. But here we instead provide a loss function that specifies only the sum, \( s \), of the
two angles, not their individual values, i.e. $s^*$ is the desired value of $a_1 + a_2$, where the $a$’s are the shoulder and elbow angles. This task, which amounts to controlling the angle of the forearm with respect to the torso, is redundant, because two joints are used to set one angle. In these simulations the desired angle-sum $s^*$ is 0.25 radians. To handle this task, we define an augmented state, $(a_1, a_2, \dot{a}_1, \dot{a}_2, s^*)$; i.e. the augmented state comprises the two joint angles, their rates of change, and the desired sum of the angles.

The loss $L = y^T Q y + u^T u$, where and $Q = 10$ and $y = (a_1 + a_2 - s^*, \dot{a}_1, \dot{a}_2)$; this $y$ appears in the loss because it is the vector that will equal 0 when the arm reaches its target state.

Figure 5.8A shows that the first two controllers don’t manage to drive $s$ to $s^*$ in the plotted time, but the optimal controller (the seventh in this run) does. For this final controller (and only for it, to avoid clutter) we plot also the individual joint angles in dark and light gray — but again, the controller doesn’t care about these, only about their sum.

In Figure 5.8C the commands to shoulder and elbow are plotted in black and gray.
Figure 5.7: GHJB algorithm learns to drive a linear second-order plant to 0 so as to minimize the integral of a time-dependent loss $L = \exp(-A[T - t]^2)Qx^T x + Ru^T u$, where $A = 100$, $Q = 10$, and $R = 1$. The augmented state is $(x, t, T)$ where $t$ is time and $T$ is the desired duration of the movement. Learning takes about five rounds of RLS with 100 log-cosh features and 1000 examples per round.
Figure 5.8: GHJB algorithm learns a task with kinematic redundancy, driving a nonlinear fourth-order plant, the planar two-link manipulator, so that the sum, $s$, of its two joint angles $a_1$ and $a_2$ equals some desired value $s^*$. The augmented state is $(a_1, a_2, \dot{a}_1, \dot{a}_2, s^*)$ and $L = y^TQy + u^Tu$, where $y = (a_1 + a_2 - s^*, \dot{a}_1, \dot{a}_2)$ and $Q = 10$. Learning takes five to ten rounds of RLS with 300 log-cosh features and 1200 examples per round. (A) Black dotted, dashed and solid lines represent the angle sum $s = a_1 + a_2$ as controlled by the first, second and final controllers, respectively. Gray lines are individual joint angles, shoulder $a_1$ in dark gray, elbow $a_2$ in light gray, plotted for the final controller only. (B) Costs accrued by the three plotted controllers. (C) Commands signals delivered by the three controllers: shoulder commands in black, elbow commands in gray.
Chapter 6

General conclusions and significance

I have presented the first mechanism for optimal sensorimotor control that runs without weight transport. To illustrate the issues I have described a specific algorithm involving generalized Hamilton-Jacobi-Bellman equations, but my message applies quite broadly, i.e. most control algorithms rely on weight transport in many different ways, but they can be recast into forms that are free of such transport by the mechanisms I have described.

6.1 VARIATIONS

I have argued that GHJB is a promising method for biological adaptive control, but there are other possibilities. Here I sketch four other approaches that may have some advantages.

One possibility is to learn $J$ directly, rather than $\dot{J}$: starting from some initial state $x_0$, perform a movement to completion (i.e. until your loss falls to near 0), integrating the
loss as you go, so that at the end you have computed $J(x_0)$. Do this from many different starting points $x_0$ and use the resulting $J(x_0)$’s to train an estimator, $\hat{J} = w\theta$. Then derive an estimate of $dJ/dx$. This method seems to work about as well as GHJB. In the form I have just described, it learns more slowly, because it has to perform a complete movement to obtain a single supervisor signal, $J(x_0)$, but there may be ways around that problem. With this type of algorithm, the weight-transport issues are much the same as with GHJB.

Another option is to drop the idea behind Eqs. (5.3) and (5.4), which is that the new controller $u^{(n+1)}$ should be the one that descends though the contours of $J^{(n)}$ with the lowest cost. Instead we could opt for a less “greedy” algorithm, where $u^{(n+1)}$ is only a modest improvement on $u^{(n)}$, obtained by moving a short distance away from $u^{(n)}$ down the gradient of $e = \left( \frac{d\hat{J}^{(n)}}{dx} \right) \dot{x} + L$ with respect to the controller parameters. This change transforms the algorithm from policy iteration to a kind of policy-gradient learning. Again, it doesn’t affect anything related to weight transport.

We can dispense with the learning rule of Eq. (5.26) if we suppose that transposition is handled by the genome. For instance, the synaptic weights making up $W_\theta$ could be set by a recipe encoded in our DNA, and the same group of genes could guide the construction of the transpose $W_\theta^T$. In other words, the existence of transposes might conceivably be part of our genetically programmed microanatomy, like the wiring patterns of the cerebral and cerebellar cortices.
A more radical departure might be to abandon the whole notion of successive approximation. Here the idea is that a single round of GHJB is often enough to yield a decent controller. This is particularly clear in Figures 5.5 and 5.6. It is less impressive in Figures 5.4 and 5.8, where the final controller is much better than $u^{(2)}$, but even here, $u^{(2)}$ is a substantial improvement over $u^{(1)}$. For tasks where strict optimality isn’t needed, or isn’t worth the effort, the brain might settle for $u^{(2)}$ and greatly simplify its motor learning. In this scheme, we would be born with an estimate of the cost-to-go gradient $df^{(1)}/dx$ for some less-than-optimal controller $u^{(1)}$ (this notion of innate gradients is less familiar than that of innate controllers, but no different in principle). There would be no need to learn any $J$’s, no role-swapping between models and controllers, and no transposition. Motor learning would reduce to computing a minimizer, as in Eqs. (5.3) and (5.4). Essentially it would boil down to learning the matrix $G(x)$ from the plant equation Eq. (5.1). This matrix is called the control jacobian, or the matrix of sensitivity derivatives, and as I have said, there are ways of learning and applying it without weight transport (Abdelghani, Lillicrap et al. 2008). This scheme retains some of the flavor of optimal control, in that it centers on a loss function and devises a controller to reduce the time-integral of that loss, but of course it doesn’t usually reduce the integral to its minimum.
6.2 ROLE OF THE TRANSPOSE

One aspect of the theory that may need more explanation is the central role of the transpose matrix $W_\theta^T$ in the GHJB algorithm. In this section I will clarify its role by showing what happens if we try to modify the algorithm to run without it.

It first glance it may seem easy to avoid the transpose. Comparing Eqs. (5.21) and (5.24), it is clear that we are approximating $[df^{(n)}/dx]^T$ by adjusting an $n_\theta$-by-$n_\theta$ diagonal matrix $\text{diag}(w)$ and left-multiplying it by the pre-existing $n_x$-by-$n_\theta$ matrix $W_\theta^T$ to yield an overall weighting $W_\theta^T\text{diag}(w)$ — an $n_x$-by-$n_\theta$ matrix that transforms $\phi$ into an estimate of $[df^{(n)}/dx]^T$. So why can’t we simply define an $n_x$-by-$n_\theta$ overall weighting matrix $W$ and adjust it by learning until, we hope, it comes to equal $W_\theta^T\text{diag}(w)$? That is, we define an estimate based on $W$,

$$
\left[ \frac{df^{(n)}}{dx} \right]^T = W\phi \tag{6.1}
$$

Then we compute an estimation error based on an available teaching signal such as $L$, e.g.

$$
e = \hat{x}^T \left[ \frac{df^{(n)}}{dx} \right]^T + L \tag{6.2}
$$

And we use the error to adjust $W$ by some learning rule such as RLS, NLMS, or LMS, e.g.
\[ \dot{W} = -\eta e \left( \frac{\partial e}{\partial W} \right)^T = -\eta \hat{x} \Phi^T \] (6.3)

With this plan, we dispense with the factors diag(\(w\)) and \(W_\theta^T\), so we don’t have to worry about transposing weight matrices.

The plan fails because it discards crucial information. Our learning rule is based on the error signal \(e\) in Eq. (5.17), obtained by comparing \(\hat{J}\) and \(\hat{\hat{J}}\). That is, \(e\) is the difference between \(-L\) and \([dJ^{(n)}/dx] \hat{x}\), where the latter is a scalar quantity reflecting the component of the vector \(dJ^{(n)}/dx\) along \(\hat{x}\). Now \(dJ^{(n)}/dx\) varies as a function of the state \(x\), but for any one state the probe vector \(\hat{x}\) is always the same. So for any one state, we only ever probe one dimension of \(dJ^{(n)}/dx\) — the one parallel with \(\hat{x}\). This one-dimensional datum is not enough to reconstruct the full vector \(dJ^{(n)}/dx\) unless we exploit a further constraint.

The crucial constraint is that \(dJ^{(n)}/dx\) is not just any vector function of \(x\), but is the gradient of a scalar function \(J^{(n)}\) of \(x\). Given this fact, \(W\) must have the form \(W_\theta \text{diag}(w)\), where \(W_\theta\) is the matrix that (together with some nonlinearity) converts \(x\) to \(\Phi\) — see Eq. (5.20) and the definitions of \(\Theta\) preceding Eqs. (5.9), (5.10) and (5.11). In my algorithm I impose the correct form on \(W\) by computing \(W_\theta^T\) and leaving only the \(n_\theta\) elements of \(w\) to be learned. So the transposed matrix \(W_\theta^T\) plays a vital role in GJHB. And it is equally
important in algorithms, such as the first variant described above, that derive $d\hat{J}^{(n)}/dx$ from $\hat{J}$ rather than $\hat{J}$.

**6.3 HOW MANY NEURONS DO WE NEED?**

If we implement the method of Abu-Khalaf and Lewis (2004; 2005) by representing their invertible matrix in neural firing rather than synaptic weights then we need fully $n_\theta^2$ cells.

If we eliminate the matrix inversion by using an online learning method such as LMS, we need much smaller numbers: on the order of $n_\theta n_x$ to represent $W_\theta$, but still this product $n_\theta n_x$ could be very large. If instead we represent $w$ and $W_\theta$ using synaptic weights, as I have described in Chapter 5, then we get by with a cell count of about $2n_\theta$ per control system (plus relatively negligible numbers of neurons, on the order of $n_x$, to carry $x$ and $d\hat{J}/dx$). And if we avail ourselves of bipartite synapses then we need only about $n_\theta$ cells for all the control systems that share the feature vector $\phi$. All these cell counts ignore the redundancy in the brain, so they don’t yield absolute numbers of neurons, but they do indicate the relative numbers required by different algorithms.
6.4 OPEN QUESTIONS

My GHJB mechanism leaves plenty of room for extension and improvement. In its current form it neglects noise and delays, and it requires state feedback. It is also far from foolproof. It is guaranteed to find the optimal controller if it gets exact estimates of each $J^{(n)}$ (Abu-Khalaf and Lewis 2005) — I sketched the proof in Chapter 5 — and of course those $\hat{J}^{(n)}$ can be made arbitrarily close to exact by choosing enough features, but in practice all estimates are approximate, and if we choose too few features then the method fails. A subtler pitfall is that a set of features may suffice to fit $J^{(1)}$, in the first round of the algorithm, but then fail to fit $J^{(2)}$ because this second cost-to-go function has a more complicated shape than the first. One solution is to use a lot of features, but a more sophisticated variant might adjust the features from round to round. Another problem is that we may get a good fit to $J^{(n)}$ but nevertheless a poor controller $u^{(n+1)}$ owing to range effects. That is, we learn $J^{(n)}$ over some region of state space but then $u^{(n+1)}$ carries us outside that range, e.g. in time-optimal tasks, the new controller may generate higher velocities than the algorithm has seen so far. So the new controller drives itself into unmapped territory where it hasn’t learned what to do. An obvious solution is to train over large regions of state space, but in general, the larger the region the more features we need. One simple device that helps with all these convergence issues is to test each new controller before discarding the old one: try a few motions, and if the new controller $u^{(n+1)}$ isn’t really an improvement then revert to $u^{(n)}$, and either rest content with it or try
again to make a better $u^{(n+1)}$, for instance by relearning $dJ^{(n)}/dx$ with more training examples, or a wider range of examples, or perhaps more or different features.

6.4 CONCLUSION

I have presented the first mechanism that achieves adaptive optimal, or near-optimal, control without weight transport in cases where the system has a very large number of possible motor commands at each moment.

My scheme suggests several predictions, e.g. it says that some cells in neural control systems should code estimates of $dJ/dx$ in their firing, and that these estimates should feed controllers. In its most efficient form, the scheme calls for bipartite synapses. My model-controller role-swapping mechanism implies that synapses should alternate between learning and stasis, evolving toward a new $w^{(n)}$ or storing an old $w^{(n-1)}$, as the optimization proceeds. And the algorithm works even if it doesn’t actually perform the movements that supply the teaching signal $L$ in Eq. (5.17) but merely simulates them in an internal model, so the theory says motor optimization should be possible with mental practice or in sleep.

Details of the scheme depend on the choice of loss and features, and on whether we use pure GHJB or some variant, but its common core is the following four principles. First, we solve a hard optimization problem by breaking it into a series of simpler ones. Second, my mechanism is novel in the way it creates and uses knowledge of its own network.
Generally, learning algorithms are more effective when they take better advantage of self-knowledge, e.g. the algorithms called weight perturbation and node perturbation (Werfel, Xie et al. 2005) are similar, but the latter learns much faster because it exploits the basic form of the input-output function of its neurons. In my scheme the learning rule of Eq. (5.18) exploits the neuronal properties and architecture of the whole network, and that architecture is enforced in part by the algorithm itself, i.e. another learning rule, Eq. (5.26), lays down a special network structure (with a copy of $W^T_\theta$ on the cells coding $dJ/dx$) which enables Eq. (5.18) to create an optimal controller. Third, the theory calls for a neural representation of $dJ/dx$ — a variable central to almost any approach to optimal control. And fourth, I have shown that the lack of weight transport is an obstacle that crops up over and over but can be offset by three mechanisms: the role-swapping of model and controller I use to store $w^{(n)}$, the copying-by-learning I use to transpose $W_\theta$, and, more speculatively, bipartite synapses. These devices may be useful for other neural algorithms, even beyond optimal control, because they perform a basic role, compensating for the absence of weight transport in the brain.
Appendix A

Pendulum swing and hold method code

%Initialize
nx = 2; nz = 3; ne = 2; endT = 20000;
gt0 = endT; if gt0 < 0, gt0 = 0; end, gtStep = ceil((gt0)/1000); n = 0;
x(1) = 0; x(2) = -0.001; u = 0; umax = 2; L =0; dL_du =0;
m = 0.16*1e-6; %bob mass
r = 2.5*1e6; %rod length
g = 9.81; %gravity
nrg(1) = m*r*(g*(1 - cos(x(1))) + 0.5*r*x(2)*x(2)); %total energy
nrgRef = 2*m*g*r; %desired energy ~ potential energy at zenith
eta = 1; alpha = sqrt(1.5); beta = sqrt(2); gamma = 5*umax; delta = 7;
e(1, 1) = alpha*( x(2)*u + gamma*tanh(0.05*(nrg(1) - nrgRef)) );
e(2, 1) = beta *( 2*x(2) + (x(1) - pi) );
ed(1) = e(1,1) + e(2,1);
for i = 1:ne, for j= 1:nz, De_Dz(i, j) = 2*(rand - 0.5); end; end;
for i = 1:nx, z(i, 1) = x(i); z(nz, 1) = u; end;
reset = 0; ts = 0;

%Time loop
for t= 2:endT

% Initialize
if x(1) >= pi, reset= reset + 1; end
if (x(1) >= pi) && (reset <= 1)
\[ x(1) = 0; \quad x(2) = -0.001 \times 0.5; \quad u = 0; \quad ts = 0; \]
end;

\textbf{Plant}
\[
DDx = \frac{u}{m \times r} - \sin(x(1)) \times g / r;
\]
\[
x(2) = x(2) + DDx; \quad x(1) = x(1) + x(2);
\]

\textbf{Error}
\[
nrg(t) = m \times r \times (g \times (1 - \cos(x(1))) + 0.5 \times r \times x(2) \times x(2)); \quad \% \text{total energy}
\]
\[
Dnrg = nrg(t) - nrg(t - 1);
\]
\[
e(1, t) = \alpha \times (Dnrg + \gamma \times \tanh(0.05 \times (nrg(t) - nrgRef)));
\]
\[
e(2, t) = \beta \times (DDx + 2 \times x(2) + (x(1) - \pi));
\]
\[
\% e(2, t) = \beta \times (DDx + \delta \tanh(0.05 \times (x(2) + (x(1) - \pi))));
\]
\[
L = 0;
\]
for i = 1:ne, L = L + 0.5 * e(i, t) * e(i, t); De(i) = e(i, t) - e(i, t - 1); end;

\textbf{Plant model (doesn't need nrg as an input)}
for i = 1:nx, z(i, t) = x(i); z(nz, t) = u; end
Dzip = 0;
for j = 1:nz, Dz(j) = z(j, t) - z(j, t - 1); Dzip = Dzip + Dz(j) * Dz(j); end;
if Dzip > 0
for i = 1:ne
De_(i) = 0; for j = 1:nz, De_(i) = De_(i) + De_Dz(i, j) * Dz(j); end;
em(i) = De_(i) - De(i);
for j = 1:nz, De_Dz(i, j) = De_Dz(i, j) - em(i) * Dz(j) / Dzip; end;
end;
end;

\textbf{Controller}
dL_du = 0; for i= 1:ne, dL_du= dL_du + e(i, t)*De_Dz(i, nz); end;

LdLip = dL_du*dL_du;

if dLip > 0, LyDy = 0;

for i= 1:ne
    for j= 1:nz - 1
        LyDy= LyDy + e(i, t)*De_Dz(i, j)*Dz(j);
    end
end

Du_bc = - (2*L + LyDy)*dL_du/dLip;

u_minus = u;   u = u_minus +Du_bc;

if abs(u) >= umax, u = umax*sign(u); end;

u = u + 0.05*(rand - 0.5); Du = u- u_minus;

end

% Time to reach zenith
if (x(1)<=0.99*pi) && (reset<=1), ts =  ts + 1; end;

% Record variables for plotting
n = n + 1;
rec(1, n) = x(1)';
rec(2, n) = 100*x(2)';
rec(3, n) = u';
rec(4, n) = L';
end

% Time to reach zenith
if (x(1)<=0.99*pi) && (reset<=1), ts =  ts + 1; end;

% Record variables for plotting
n = n + 1;
rec(1, n) = x(1)';
rec(2, n) = 100*x(2)';
rec(3, n) = u';
rec(4, n) = L';

end

time=1:length(rec(1,:));

% Plotting
figure(1)
plot(time, rec(1,:))
xlabel('Time (ms)')
ylabel('Position')
Appendix B

Sweep method code

//Learns the biphasic command for a 2nd-order system by repeating a single motion,
//sweeping back to compute costates. To implement this algorithm in a network,
//use a set of features, endT in number, each a unit pulse. That way, the control
//trajectory is stored in synapses. The state trajectory could be stored by
//integrating products of x with the pulse features.

const
nx = 2;  // # state variables
nit = 1000;  // # iterations
endT = 100; // duration of each iteration

var
  // Graphics variables
  clGold: TColor;
  gxZero, gyZero, gt0: integer;
  gxScale, gyScale, gxMax, gyMax, gx, gy: double;
  // Simulation variables
  i, j, k, t, it, nextPlot: integer;
  umax, cost, prevCost, lowCost, dhduip: double;
  u, L, H, dH_du: array[0..endT] of double;
  x, lam: array[1..nx, 0..endT] of double;

begin
  // Set up graphics
  clGold := $0000A3DA;
Canvas.Brush.Color := clWhite;
Canvas.Font.Color := clBlack;
Canvas.FillRect(Rect(0, 0, ClientWidth, ClientHeight));
gt0 := 0; //start time for plots
gxMax := endT - gt0; gyMax := 1;
gxZero := 50; gxScale := (ClientWidth - 2*gxZero)/gxMax;
gyZero := ClientHeight div 2; gyScale := -(ClientHeight - 2*gxZero)/(2*gyMax);

//Draw axes
Canvas.Pen.Color := clGray;
Canvas.MoveTo(gxZero, gyZero);
Canvas.LineTo(round(gxZero + gxScale*gxMax), gyZero);
Canvas.MoveTo(gxZero, round(gyZero - gyScale*gyMax));
Canvas.LineTo(round(gxZero + gxScale*gxMax*j/10), gyZero + 2);
Canvas.LineTo(round(gxZero + gxScale*gxMax*j/10), gyZero);
Canvas.MoveTo(gxZero + 2, round(gyZero + gyScale*gyMax*j/10));
Canvas.LineTo(gxZero, round(gyZero + gyScale*gyMax*j/10));
Canvas.MoveTo(gxZero + 2, round(gyZero - gyScale*gyMax*j/10));
Canvas.MoveTo(gxZero, round(gyZero - gyScale*gyMax*j/10));
end;

//Initialize
umax := 0.001; //0.018;
for t := 0 to endT - 1 do u[t] := umax*(random - 0.5);
Canvas.Pen.Color := clRed;
Canvas.Pen.Width := 1;
nextPlot := 0;
cost := 100; lowCost := 3;

//Iterate
for it := 1 to nit do begin

//Forward time loop
x[1, 0] := 1; x[2, 0] := 0; L[0] := 0.5*x[1, 0]*x[1, 0];
prevCost := cost; cost := L[0];
for t := 1 to endT do begin
  x[1, t] := x[1, t - 1] + x[2, t - 1];
  x[2, t] := x[2, t - 1] + u[t - 1];
  L[t] := 0.5*x[1, t]*x[1, t];
  cost := cost + L[t];
end;
if cost >= prevCost then lowCost := prevCost else lowCost := 0.99*lowCost;

//Backward time loop
lam[1, endT] := 0; lam[2, endT] := 0; dhduip := 0;
for t := endT - 1 downto 0 do begin
  lam[1, t] := x[1, t] + lam[1, t + 1]; //remember to use transposes
  lam[2, t] := lam[1, t + 1] + lam[2, t + 1];
  H[t] := L[t] + lam[1, t + 1]*x[1, t + 1] + lam[2, t + 1]*x[2, t + 1];
  dH_du[t] := lam[2, t + 1];
  dhduip := dhduip + dH_du[t]*dH_du[t];
  //u[t] := u[t] - 0.001*dH_du[t];
  //if abs(u[t]) > umax then u[t] := sgn(u[t])*umax;
end;

//Adjust control sequence
for t := endT - 1 downto 0 do begin
    u[t] := u[t] - dH_du[t]/dhduip;
    if abs(u[t]) > umax then u[t] := (u[t]/abs(u[t]))*umax;
end;

//Plot
{  if log2(it) >= nextPlot then begin
    for t := 0 to endT - 1 do begin
        gy := 100*u[t]; //0.1*H[t]; //0.1*lam[2, t];
        if t = 0 then Canvas.MoveTo(round(gxZero + gxScale*t), round(gyZero +
            gyScale*gy))
        else Canvas.LineTo(round(gxZero + gxScale*t), round(gyZero + gyScale*gy));
    end;
    nextPlot := nextPlot + 1;
    Canvas.Pen.Color := clBlue;
end;
}  //it

Canvas.Pen.Color := clGold;
Canvas.Pen.Width := 3;
for t := 0 to endT - 20 do begin
    gy := 100*u[t]; //0.1*H[t]; //0.1*lam[2, t];
    if t = 0 then Canvas.MoveTo(round(gxZero + gxScale*t), round(gyZero +
        gyScale*gy))
    else Canvas.LineTo(round(gxZero + gxScale*t), round(gyZero + gyScale*gy));
end;

Canvas.Pen.Width := 1;
Canvas.Pen.Color := clGreen;
for t := 1 to endT - 2 do begin
  //gy := 0.5*(H[t] - H[t - 1]);
  //gy := 10*(L[t] - L[t - 1]);
  if t = 1 then Canvas.MoveTo(round(gxZero + gxScale*t), round(gyZero +
                             gyScale*gy))
    else Canvas.LineTo(round(gxZero + gxScale*t), round(gyZero + gyScale*gy));
end;

Canvas.Pen.Color := clFuchsia;

if t = 1 then Canvas.MoveTo(round(gxZero + gxScale*t), round(gyZero + gyScale*gy))
  else Canvas.LineTo(round(gxZero + gxScale*t), round(gyZero + gyScale*gy));
end;
(Mandl, Melvill Jones et al. 1981 )

Canvas.Pen.Color := clLtGray;
for t := 0 to endT - 20 do
  if t = 0 then Canvas.MoveTo(round(gxZero + gxScale*t), round(gyZero +
                               gyScale*x[1, t]))
    else Canvas.LineTo(round(gxZero + gxScale*t), round(gyZero + gyScale*x[1, t]));
end;
Appendix C

GHJB algorithm code

% Initialize
nr = 10; nx = 2; nf = 10;  % # rounds; dim'ns of state; features
nm = 10*nf; nm_test = 10;  % # training examples/round; test examples
wait = 800;  % duration of mov't in ms
UCF = 0.01;  % unit-conversion factor for integration
kappa = 0.025; rho = 0.5; mu = 0.1;  % plant coefficients
F = UCF*[0 1; -kappa/mu -rho/mu]; G = UCF*[0; 1/mu];  % plant matrices
W = 0.5*(rand(nf, nx) - 0.5);  % fixed weights used to compute features
w = zeros(1, nf);  % adjustable synapses
X_TEST = 0.8*(2*rand(nx, nm_test) - 1);  % columns are test states
X_TEST(:, nm_test) = [0.5; 0];

Plant = @(x, u) F*x + G*u;
Loss = @(x, u) x'*x + u'*u;  % the cost rate
Controller = @(x) -0.02*sum(x);  % initial controller u(1)
Feature = @(x) tanh(W*x);  % the vector phi

% Compute avg cost of u(1) over nm_test mov'ts
C_sum = 0;
for m = 1:nm_test
    REC = Con_ode(Plant, Loss, Controller, [0; wait], X_TEST(:, m));
    C = REC(size(REC, 1), nx + 2);  % cost
    C_sum = C_sum + C;
end  % m
C_avg(1) = C_sum/nm_test
subplot(3, 1, 1), plot(REC(:, 1), REC(:, 2:3), ':'), hold on
for r = 1:nr

% Learn dJ/dt by GHJB
% Takes random samples of state space; could also work by performing
% continuous mov'ts and sampling along the paths.
% P = 1e8*eye(nf); % soft initializat'n for RLS
for i = 1:nm
    x = (2*rand(nx, 1) - 1); u = Controller(x); L = Loss(x, u);
    Dx = Plant(x, u);
    phi = Feature(x); z = phi.*(W*Dx); zip = z'*z;
    e = w*z + L;
    w = w - e*z'/zip; % NLMS
    %Pz = P*z; q = Pz/(1 + z'*Pz); P = P - q*Pz'; w = w - e*q'; % RLS
end

% Compute costs of new controller
Controller = @(x) -0.5*G'*W'*w'.*Feature(x);
C_sum = 0;
for m = 1:nm_test
    REC = Con_ode(Plant, Loss, Controller, [0; wait], X_TEST(:, m));
    C = REC(size(REC, 1), nx + 2); % cost
    C_sum = C_sum + C;
end % m
C_avg(r + 1) = C_sum/nm_test
if r == 1
    subplot(3, 1, 1), plot(REC(:, 1), REC(:, 2:3), '--')
function REC = Con_ode(Plant, Loss, Controller, T, xi)
% T in ms

x = xi; C = 0; t_span = T(2) - T(1);
for t = 1:t_span
    u = feval(Controller, x); L = feval(Loss, x, u); Dx = feval(Plant, x, u);
    x2 = x + Dx;
    REC(t, :) = [T(1) + t - 1, x', C, u']; % each row of REC holds data for 1 time step
    u2 = feval(Controller, x2); L2 = feval(Loss, x2, u2); Dx2 = feval(Plant, x2, u2);
    x3 = x + (Dx + Dx2)/4;
    u3 = feval(Controller, x3); L3 = feval(Loss, x3, u3); Dx3 = feval(Plant, x3, u3);
    x = x + (Dx + 4*Dx3 + Dx2)/6;
    C = C + (L + 4*L3 + L2)/6;
end  % t
References


