Self-Modeling Neural Systems

Gregory D. Wayne

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

under the Executive Committee

of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2013

ABSTRACT

Self-Modeling Neural Systems

Gregory D. Wayne

Goal-directedness is a fundamental property of all living things, but it is perhaps most easily identified in the movement patterns of animals. Ethologists have divided the basic forms of animal behavior into three categories: reproductive, defensive, and ingestive, all of which depend on the complex orchestration of motor control. In this dissertation, we use the framework of optimal control theory to model goal-directed behavior and repurpose it in new ways. We demonstrate a method for creating a hierarchical control network in which higher levels of the control hierarchy deal with tasks of increased abstractness. In a two-level system, the lower-level deals with short time-scale, low-dimensional motor control, and the higher-level is charged with longer time-scale, higher-dimensional planning. Central to our approach to joining the levels is the construction of a forward model of the behavior of the lower-level by the higher-level. Thus, we extend ideas of optimal control theory from controlling a "plant" to controlling a controller. We apply our method to the example problem of guiding a semi-truck in reverse around a field of obstacles. The lower-level controller drives the truck, and the higher-level detects obstacles and plans routes around them. In other work, we consider whether it is possible for a neural system that obeys certain biological constraints to solve optimal control problems. We exhibit a simple method to train a different kind of internal model, a neural network model of the Jacobian of the plant, and we integrate the internal model in a forward-in-time computation that produces an optimal feedback controller. We apply our method to two well-known model problems in optimal control, the torque-limited pendulum and cart-pole swing-up problems.

Contents

List of Figures

Acknowledgements

There's an expression in Washington D.C., known as a "Washington read," which refers to flipping through a book or article only to find out if the author mentions your name. If you've come here first, shame on you!

I first want to thank my family. My parents, Ellen and Peter, are also two of my closest friends. I still catch myself bragging about you. I have pride in your morals and your minds, and I would do well in life to try to inhabit the world as you do. My siblings, Teddy, Elizabeth, and Geoffrey, you are also role models for me. It's been exceptionally satisfying as I've grown up to see what kinds of adults you've become, to see how we are refracted in one another and how not. I wish I could see you more. My grandma, Bess! You'll soon be 100 years old. How the world has changed since 1913! Do you remember, when babysitting me as a child, telling me about a radio program you had heard on mnemonists and the method of loci? We practiced for hours; this was probably the first awakening in me of an interest in mentality.

My good friends, Ross, Jess, Karan, Jared, Alex, Ryan, Kiel, Rachael, Antonio, Jonathan, Anna .... I am just going to add extra names, so no one has to feel left out. You've been the source of so many good feelings and good times. You keep the meanness in the news at bay. You share your fascinations and let me share mine with you.

My good friends at work, you are also good friends ... period. Saul, Clay, Drew, Armen,

Zev, David, Yashar, Baktash, Carl, Tim, Irene, Ann, Wujie, Mattia, Burcin, Misha, Brian, Patrick, Merav, and Claudia, I am sorry for starting collaborations with so many of you that I had nary a moment to work on (especially you, Mattia and Claudia). Clay and Drew, we finished ours! How fun was that?

To the program administrators, Alla and Cecil, there is no one I would rather be yelled at by than you.

To the program directors, Carol, Ken, and Darcy, you have been entirely supportive and understanding throughout my graduate career. I want to redouble my thanks to you, Carol. You are my favorite person to see on the street in front of P&S.

To my committee, a word for each:

Nate, I would like to thank you for sharing your interests in physiology. I loved it when you would pull me aside to tell me about a paper you'd just read about adaptive filters in the cerebellum or internal models postulated in the VOR. (Incidentally, I hope we can finish this project soon.)

Mark, I want to thank you for turning my committee meetings into deep discussions. You have a way of politely segueing from a simple clarifying question to the nuts and bolts of an idea.

Ken, you are showing up here twice in two different guises. I still remember my first visit to the theory center was to see a lecture by you about your work with Brendan Murphy. I felt such a flush of enthusiasm. Throughout my time in the center, I've admired your passion for science and your ability to make a critical scientific point in a persistent but well-meaning way.

Yann, you've practically been my advisor *in absentia*. I am permanently grateful to you for the time you let me spend in your lab and the lessons I squeaked in at your white board.

You taught me to look for common, simple structure amidst the menagerie of algorithms and methods used in machine learning.

Larry, I want to apologize for all of the times I walked into your office with nothing much to report scientifically, just because I wanted to say hi. It has been extremely fun to be your student. It's been said that style comes from failed imitation. I have tried and failed to imitate you; I can only hope that failing leads to style of its own.

Chapter 2 Acknowledgements

We are grateful to Yann LeCun, Sara Solla, John Krakauer, and Peter Dayan for ideas, support, and criticism. We thank Yashar Ahmadian, Loïc Matthey, Mattia Rigotti, Ann Kennedy, Darcy Wayne, and Saul Kato for helpful discussions. Research supported by the Gatsby, Swartz, Kavli, and National Science Foundations and by NIH grant MH093338.

Dedicated to those who have taught me and to those who teach others.

Prologue

It comes as a shock to learn that our conscious experience is the result of a physical structure. The seismograph barely settles down before we learn that this physical structure is made of microscopic cells called neurons. A mystery announces itself: What are these neurons doing, and, from their perspective, how do they know what to do? The miracle of the brain is that not only do these neurons know what to do, they act in such a concerted manner that our introspective experience is independent of their operation. We feel whole, unitary, not the product of 100 billion autonomous cells. We act with purpose.

And so must they. Neurons are not the loosely connected stars in a constellation; they are arranged in circuits, fulfilling specialized roles.

Just as the computer has parts – random access memory, buses, the central processor, input/output ports – so too do neural systems have parts that interact in prescribed ways. In this thesis we explore what some of those parts might be.

Chapter 1

Introduction

All stable processes we shall predict. All unstable processes we shall control. – John von Neumann

1.1 Introduction

Norbert Wiener, in his seminal book *Cybernetics* (Wiener, 1961), set a grand theoretical agenda for research in machine intelligence and neuroscience. He argued that we would understand intelligence when we could connect the sciences of information and control theory. Wiener was on to something, and his book was initially received to great acclaim, followed, though, by petering enthusiasm. In 1948, there simply was not enough *there* there.[1] Claude Shannon himself scoffed at "bandwagonism" – the attempted and usually misinformed explanation of everything under the sun by the principles of information theory (Shannon, 1956). And yet, these ideas percolated; staid old biology gradually transformed into molecular biology and genetics when it was realized that the very process of reproduction was bound by efficient coding and error correction. In neuroscience, Attneave (Attneave, 1954) and Barlow (Barlow, 1961) applied information theory, perhaps to less resounding effect, to understand sensory processing based on metaphors of redundancy reduction. In the present day, information theory has become one of the central tools of analysis for the theoretical neuroscientist. Not bad for a mathematical field originating in telegraphy.

Yet control theory, whose development was furthered not by names like Shannon and Wiener but by names like Bellman, Kalman, and Pontryagin, lay inside the penumbra for much longer. Perhaps not until the 1970s did the corresponding cross-fertilization begin between control theory and neuroscience. It is not clear if this lag was just a historical accident or a function of practical difficulty.[2] Even now, one is more likely to hear "predictive coding" escape the mouth of a neuroscientist than "model-predictive optimal control," even

[1] to paraphrase Gertrude Stein

[2] It is hard to record from brains inside moving animals, and control theory computations demand modern computers.

though the case could be made that the two concepts are equally central to their respective disciplines, and the two disciplines in turn are equally central to explaining intelligence.[3]

As of 2013, the theory of optimal control has become the dominant framework for explaining motor behavior. The seeds of this development were sewn by researchers like Flash and Hogan (Flash and Hogan, 1985), who showed that the computations of optimal control theory could describe the qualitative behavior of human arm movements, for example, the smoothness of human reaches. In 2002, Emanuel Todorov and Michael I. Jordan argued that optimal feedback control could account more sufficiently for movements responsive to perturbations (Todorov and Jordan, 2002). In optimal control theory, a movement trajectory is planned before action occurs, and it is assumed that the intended movement unrolls as a predetermined script on movement initiation. This is known as "open-loop" control. In optimal feedback control, the result of optimization is instead a control "policy" or controller – a function or dynamical system that takes in sensory or state information and produces motor commands as outputs. Policies are more flexible and can respond reactively to disturbances. Optimal feedback control is known as a "closed-loop" paradigm.

Yet optimal control and optimal feedback control are solely normative models. That is, they describe a set of computations relevant to planning and executing movements, but they do not strongly prescribe the neural mechanisms that can actually achieve those movements. This is both a strength and a weakness. They can make predictions about real motor behavior without concern for the complicated internal nature of these computations. Still, ultimately, we will need to know how neural systems actually achieve such optimal or nearly optimal performance.

Why optimal performance? Isn't it possible that the motor system is just "good enough" to

[3]This is no more an endorsement of model-predictive optimal control than it is of predictive coding. The sole point is that most neuroscientists have heard of the latter but not of the former.

solve its tasks? This is, of course, true. The motor system is probably just "good enough," but researchers with experience solving control problems can say with confidence that the distance between competence and optimality is often narrower than one might suppose. It is difficult to control a 7 degree-of-freedom arm with variable loads and applied disturbances in simplistic ways. Merely to achieve "good enough" demands substantial computation or innate structure that is already exceedingly superlative. Moreover, human motor behavior is extremely flexible. Even in novel tasks, human subjects exhibit great skill, skill that must be the product of sophisticated strategies of improvement or optimization. Human behavior is not just good enough, nor is it optimal, it is markedly better and more interesting than extant frameworks of optimal control, which can only characterize very specialized forms of optimality.[4]

1.1.1 Motor Flexibility

Consider the following intuitive examples of human behavior. A chess grandmaster sits motionless, his blood pressure as high as that of a male baboon in a lethal fight (Sapolsky, 2004), for several minutes before barely lifting a finger to place a move; a classroom student raises her hand, obeying a learned instruction from her teacher; an amputee rehearses the control of a prosthetic arm. These examples provide incontrovertible evidence for human motor flexibility of three different kinds.

1. The first example demonstrates that behavioral "planning" is largely independent of motor action.

2. The first and second demonstrate that human motor goals can be instructed or varied based on contextual signals that are quite remote in nature from the movement control

[4] I find it extremely intellectually liberating to note that, no matter how brilliant and persuasive the proponents of various theories are, we are all, at some level, radically confused.

problem itself.

3. The third demonstrates that motor control can be learned despite little *a priori* knowledge of the motor apparatus or "plant"[5] to be controlled.

Our capacity to modulate movement subject to internal and external context is extraordinary, and the understanding of this capacity connects to conceptual difficulties that reach far beyond the conventional boundaries of the field of motor control; it is fair to say that it will probably resist full-scale modeling efforts indefinitely. Nevertheless, researchers in theoretical neuroscience, motor control, and machine learning have begun to develop models that possess these three features qualitatively.

The space of possible strategies for designing control systems is large, but several important classificatory divisions exist, although individual models sometimes blur these distinctions. We have already alluded to the difference between open-loop and closed-loop control. Another prominent distinction is between control systems that invoke models of the plant and those that do not, known as "model-free" methods. Reinforcement learning methods tend to fall inside this latter category (Sutton and Barto, 1998). In these systems, one of which we will derive in this chapter, the activity of some intrinsic behavioral perturbation generator is correlated with the variation of a performance function. Perturbations correlated with reward drive adaptive change in the system. A third important distinction is between so-called "global" and "local" methods. In global controllers, the optimal motor command is pre-computed for every conceivable state. These methods are extremely powerful but typically suffer from the curse of dimensionality, also known as the "infinitude of the possible." It is staggeringly difficult to compute the right response to every possible scenario that will ever be encountered. On the other hand, local methods compute the desired motor

[5] a weird archaism from the time when chemical and factory plants were the systems controlled

response only in the vicinity of the current state. They have more modest pretensions but can often attain satisfactory performance quickly.

By a process of evolutionary selection, certain kinds of controllers are currying favor among control theorists and roboticists. Fast, dynamic, optimal control is most easily achieved when the plant equations are known directly (Tassa et al., 2011) (Tedrake, 2009). This is a much stronger kind of knowledge than possessing a model. Such a method depends on knowing the exact model and hence can only be applied within a simulation. When the exact plant equations are not given, the case relevant to animal motor behavior, identifying a model of the plant typically allows for better control solutions than can be produced by model-free methods. Identified models are known as "forward models" because they can approximate the dynamics of the system forward-in-time or forward from motor command to sensory consequence. Model-free reinforcement learning techniques require large numbers of trials to learn control policies; so far, their use has either been limited to the optimization of low-dimensional control policy parameters (with careful parameterization and built-in structure, sometimes to very impressive effect (Theodorou et al., 2010)), or they must be augmented by other tools such as "experience replay" in which data are stored in computer memory and then re-used later several times offline for learning (Lin, 1992) (Wawrzyński, 2009). Techniques similar to reinforcement learning but based on simultaneous, parallel evaluation of many forward models (whereas reinforcement learning is applied to serial evaluations of the actual system dynamics) can produce state-of-the-art results (Wang et al., 2009). It is much easier to optimize complex single trial motions than to optimize a generically capable feedback controller (Mordatch et al., 2012); i.e., it is much easier to learn local controllers than global ones.[6] Recurrent neural networks can

[6]The Mordatch paper is actually a much slower-than-real-time, open-loop computation, performed in the absence of noise, using the exact plant model. Still, it stands as the most astonishing use of optimal control I have seen.

usefully exploit hidden state to compensate for plant disturbances such as changes in the friction of joints (Sutskever, 2013) or to switch behavior based on contextual signals (Huh and Todorov, 2009).

While stimulating, the travails and successes of engineers are unlikely to yield definitive guidance to neuroscientists about what mechanisms to search for as the interplay between optimization and pre-determined structure is intricate. A "simple" method such as reinforcement learning can perform well when the right heuristics, controller structures, and problem formulations are combined. Complex model-based optimization can fail when the system model is wrong or the parameters to optimize are chosen naïvely. This trade-off between pre-determined structure and optimization parallels the eternal debate between nature and nurture.

From the camps of experimentalists, there is a long and proud tradition that points toward the existence of internal models of the motor apparatus that can adapt to disturbances. In the study of human visuo-motor coordination, there have been numerous studies of inversion and prism adaptation in which subjects don goggles that either mirror-reverse the visual field or refract it by a constant angle (Stratton, 1896) (Sugita, 1996). After a period of disorientation, subjects return to high levels of performance on various tasks. In animal studies, several surgical "rewiring" experiments have been carried out, the most dramatic of which, for our purposes, was a transposition of the ulnar and radial nerves controlling the grasping thumb of macaque monkeys (Brinkman et al., 1983). After healing, the monkeys exhibited no decline in grasping performance, despite the total reversal of the relationship between the central motor command and muscle contraction. Standard model-free methods cannot account for such dramatic changes to the motor apparatus since they conflate the learning of optimal decisions with the dynamics of the plant. If the plant changes, the rug is pulled out from under them. Other studies of motor adaptation to a force field or

the redirection of reaching angle in cursor pointing studies suggest that subjects' ability to compensate for the disturbance using verbally-instructed advice (a cognitive strategy) is distinct from slower, unconscious adaptation; in fact, the two compensatory mechanisms can interfere with each other (Mazzoni and Krakauer, 2006). In all of these studies, we see glimmers of Lashley's principle of motor equivalence (Lashley, 1930): not only can a single goal be achieved in multiple ways, but the procedural learning of a goal can also be disassociated from the details of the motor plant used to achieve it.

These experimental studies have therefore primed the pump for researchers in human motor control to take more seriously possible roles for internal models of the motor apparatus in the optimization or generation of goal-directed movement (Wolpert et al., 2011). We are sanguine about this move; at the same time, we acknowledge that the link between the methods used by engineers and the experimental studies of adaptation is still very tenuous and should be looked on with great skepticism. Others share the same perspective (Krakauer, 2013).

1.1.2 Conceptual Foundations of Optimal Control

The road to optimal control passes primarily through two theories. One of these theories poses the control problem globally: find the best control command for every state. The other of these theories poses the control problem locally: find the best sequence of commands that can generate an optimal trajectory from the present state. There are a few other ways to solve optimal control problems, which we will also touch on briefly.

Bellman's Approach

The global approach to optimal control was initiated by Bellman. It is based on his principle of optimality:

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. (Bellman, 1957)

In Figure 1.1, we diagram the principle of optimality. The vertices are states and are labeled by letters a, b, c, d, and e. The paths between states are labeled with numbers, indicating the cost of traversing them. In this discrete world, our actions, or controls, are to move along the paths marked by arrows. We begin in state a and want to get to the goal state e. For didactic purposes, we allow multiple paths between the same two connected states. We can see that there are two ways of exiting state b to get to e and also two from d to e. Suppose we are in state b. The minimal cost to get to the goal is to move along the straight path of cost 1. So, if we are in state b, and we act optimally, we expect to incur a total of 1 unit cost from then on. Now, consider state c. The minimal cost to the goal, also known as the "cost-to-go" is 5 because there is only one exit edge. Similarly, from state d, the cost-to-go is 2. The lowest cost path from a to e is to go first to state b, then to go from b to e along the straight-line path, incurring a total of 3 units of cost. We denote the cost of the best trajectory from state a, the cost-to-go, as $V(a) = 3$. Interestingly, the lowest cost path from state a takes a route through b that incurs relatively high immediate cost. The lowest cost action from a is to move to c, but once in c one would suffer a path cost of 5 to the goal, so this entire trajectory is actually suboptimal.

Suppose $f(\text{state}, \text{control})$ is a function that takes in the state (say a) and the control command u and gives us the consequent state. This is a forward model. Further, let $C(a, u)$ be the cost of executing control command u while in state a. One way of stating the principle of optimality formally is that $V(a) = \min_u \Big(C(a, u) + V(f(a, u)) \Big)$. That is, the cost-to-go from the present state is the minimum of the cost of choosing a path from the state plus the cost-to-go from the state at which we arrive. Thus, we have a recursive definition of the

cost-to-go. Evidently, to figure out what the cost-to-go is from state a, $V(a)$, we need to know the cost-to-go of the states that come after a. Therefore, one algorithm for computing the cost-to-go of each state is to start at e, noting that $V(e) = 0$. Then work backward to calculate that $V(b) = 1$, $V(c) = 5$, and $V(d) = 2$. Now, head back to state a and compute the cost-to-go from here. In this way, Bellman's principle gives us a method to compute the optimal trajectory by working backward from the goal state to the initial state. We and others sometimes say that the trajectory is computed backward-in-time from goal states. This may sound mysterious, but we mean nothing more by it than what we have just shown. Such a method of computing the optimal control is known as dynamic programming. We can see that to find the best path from the initial state, we also need to find the best path from every other state. Consequently, this method is a global method.

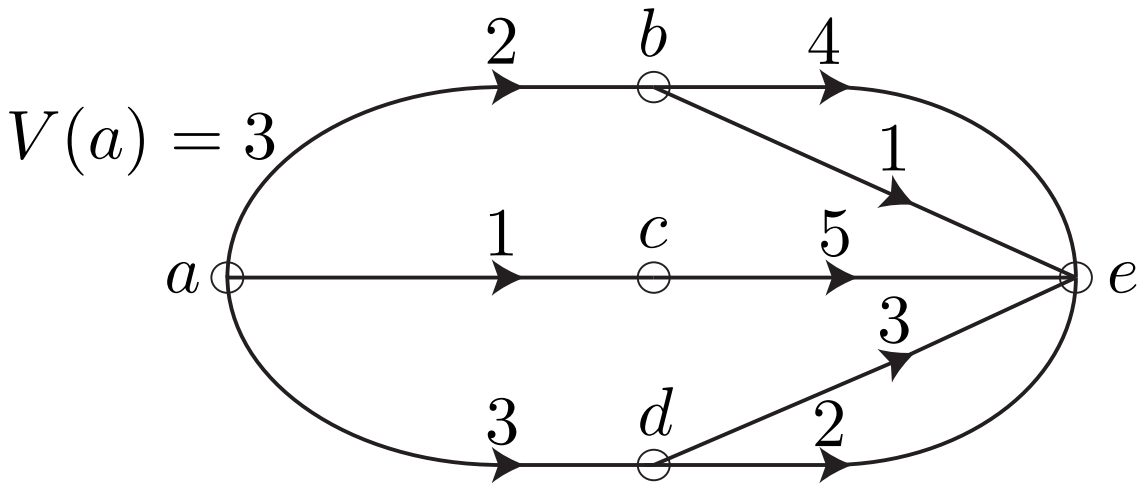

Figure 1.1: Dynamic Programming.
The minimum cost route from state a to state e incurs high immediate cost but low total cost.

Pontryagin's Approach

The other approach to optimality is due to Pontryagin and his colleagues (Stengel, 1994), though the basic idea follows naturally from the work of Euler and Lagrange in the 18th century. It characterizes the optimal trajectory to the goal using calculus (variational calculus). Consider the supposedly optimal trajectory, colored blue in Figure 1.2 from a to c. Nowhere along this path can any distortion of the path, even infinitesimal, lower the cost of the entire path. This is by assumption. Suppose we transit through b' instead of b. If the trajectory in blue is optimal, we know that this excursion can only increase the trajectory cost. If this excursion is infinitesimal, just as in calculus ($df/dx = 0$ for optima of the function $f(x)$), the change in cost is 0 to first order. Pontryagin's minimum principle therefore states that infinitesimal changes to the solution trajectory should not change its cost (again, to first order).[7] Unfortunately, calculus characterizes extrema, maxima or minima, by the same first-order condition. If we use the first-order condition alone, we do not know whether this trajectory is truly the lowest cost trajectory. It could be one of many low cost trajectories. Or it could actually be the highest cost trajectory in its neighborhood of trajectories. If one is deriving a solution analytically, which we will only do for comparison purposes once in this thesis, one must vigilantly check to see whether one has found the lowest cost or highest cost trajectory. First-order conditions in calculus are known as *necessary* conditions. The truly optimal trajectory will satisfy the minimum principle, but so will the truly pessimal. By contrast, Bellman's optimality principle is a *sufficient* condition. If we have found a trajectory satisfying Bellman's optimality principle, we know for certain that there is none better. Bellman's principle is strictly stronger than Pontryagin's. A consequence of this that we will examine later is that one can derive Pontryagin's principle from Bellman's but not conversely.

[7] The minimum principle is actually subtler than this, but we never need all the subtlety, so we'll call this the minimum principle, even if wonks would object.

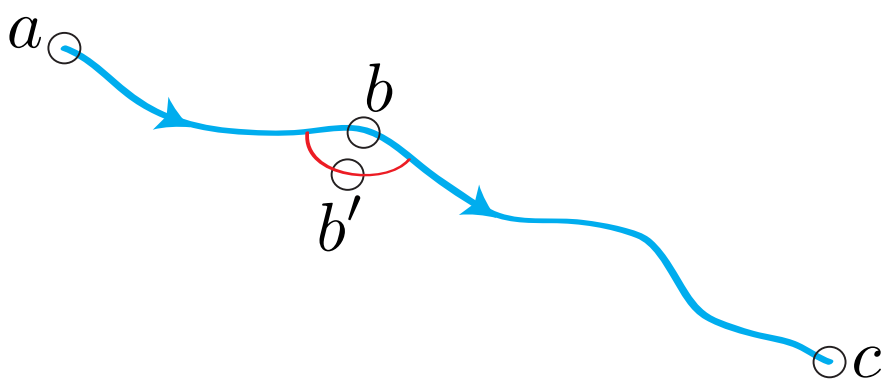

Figure 1.2: The Variational Approach to Optimization.
Any trajectory that is infinitesimally close to the optimal trajectory has the same cost to first order. We compare here a trajectory from a to c that goes through state b versus one that is extremely close but goes through b'.

1.1.3 Mathematical Foundations of Optimal Control

Bellman's Approach

Let us now examine the mathematical formalisms of optimal control. We have a system governed by known equations

$$\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, \mathbf{u}). \tag{1.1.1}$$

$\mathbf{x}$ is the state variable, and $\mathbf{u}$ is the control variable or command. We additionally have a performance criterion, $\mathcal{L}(\mathbf{x}, \mathbf{u})$, known variously as the Lagrangian or "cost-rate," that specifies the desirability of particular state-control pairs. We ask what costs we expect to accumulate after we arrive in a state at a given time and act optimally thenceforth. Again,

this is known as the optimal cost-to-go or the value function:

$$V(\mathbf{x}(t), t) = \min_{\mathbf{u}(\cdot)} \int_t^{t_f} \mathcal{L}(\mathbf{x}(t'), \mathbf{u}(t'), t')dt'. \tag{1.1.2}$$

A variable enclosing a dot, e.g., $\mathbf{u}(\cdot)$, indicates the entire sequence of values of that variable over the relevant time interval, from the initial time t to the final time t_f. As we noted before, there is implicitly a recursive structure in the definition of the cost-to-go. We can see that

$$\begin{aligned} V(\mathbf{x}(t), t) &= \min_{\mathbf{u}(\cdot)} \left\{ \int_t^{t+dt} \mathcal{L}(\mathbf{x}(t'), \mathbf{u}(t'), t')dt' + \int_{t+dt}^{t_f} \mathcal{L}(\mathbf{x}(t'), \mathbf{u}(t'), t')dt' \right\} \\ &= \min_{\mathbf{u}(t)} \left\{ \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t)dt + \min_{\mathbf{u}(t'>t)} \int_{t+dt}^{t_f} \mathcal{L}(\mathbf{x}(t'), \mathbf{u}(t'), t')dt' \right\} \\ &= \min_{\mathbf{u}(t)} \left\{ \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t)dt + V(\mathbf{x}(t+dt), t+dt) \right\}. \end{aligned} \tag{1.1.3}$$

In discrete-time, taking dt to be 1, this is known as the Bellman equation. It is closely related to the framework of Markov Decision Processes, in which we compute an expected value over the distribution of ensuing states, often discussed by researchers in computer science. In continuous-time, $dt \to 0$, we can proceed further by Taylor-expanding the right-hand side:

$$\begin{aligned} V(\mathbf{x}(t), t) &= \min_{\mathbf{u}(t)} \left\{ \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t)dt + V(\mathbf{x}(t), t) + \frac{\partial V(\mathbf{x}(t), t)}{\partial \mathbf{x}}^\top \dot{\mathbf{x}}(t)dt + \frac{\partial V(\mathbf{x}(t), t)}{\partial t}dt \right\} \\ &= \min_{\mathbf{u}(t)} \left\{ \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t)dt + V(\mathbf{x}(t), t) + \frac{\partial V(\mathbf{x}(t), t)}{\partial \mathbf{x}}^\top \mathbf{F}(\mathbf{x}(t), \mathbf{u}(t))dt + \frac{\partial V(\mathbf{x}(t), t)}{\partial t}dt \right\}. \end{aligned}$$

The value functions on both sides cancel, and the partial derivative of the value function with respect to time is independent of the control variable. All the remaining variables are

scaled by dt, so we can divide by it. Thus, we have

$$-\frac{\partial V(\mathbf{x}(t),t)}{\partial t} = \min_{\mathbf{u}(t)}\left\{\mathcal{L}(\mathbf{x}(t),\mathbf{u}(t),t) + \frac{\partial V(\mathbf{x}(t),t)}{\partial \mathbf{x}}^{\top}\mathbf{F}(\mathbf{x}(t),\mathbf{u}(t))\right\}. \tag{1.1.4}$$

This equation is known as the Hamilton-Jacobi-Bellman (HJB) equation. We can simplify the expression by defining the "Hamiltonian"

$$\mathcal{H}(\mathbf{x}(t),\mathbf{u}(t),t) \equiv \mathcal{L}(\mathbf{x}(t),\mathbf{u}(t),t) + \frac{\partial V(\mathbf{x}(t),t)}{\partial \mathbf{x}}^{\top}\mathbf{F}(\mathbf{x}(t),\mathbf{u}(t)). \tag{1.1.5}$$

Then, clearly, the Hamilton-Jacobi-Bellman equation reads

$$\frac{\partial V(\mathbf{x}(t),t)}{\partial t} = -\min_{\mathbf{u}(t)}\mathcal{H}(\mathbf{x}(t),\mathbf{u}(t),t). \tag{1.1.6}$$

If we write the optimal control as $\mathbf{u}^*(t)$ and the states along the optimal paths as $\mathbf{x}^*(t)$, then we can compress the notation further.

$$\frac{\partial V(\mathbf{x}^*(t),t)}{\partial t} = -\mathcal{H}(\mathbf{x}^*(t),\mathbf{u}^*(t),t). \tag{1.1.7}$$

The HJB equation is nonlinear due to the minimization operation and is a partial-differential equation. It is in general difficult to solve. However, if we do indeed find a way to compute the Hamiltonian $\mathcal{H}(\mathbf{x},\mathbf{u},t)$, then we can simply compute the optimal control command $\mathbf{u}^*(t)$ by minimizing it. Knowing the Hamiltonian disconnects the states from one another, allowing us to find the optimal command without extensively analyzing its future consequences.

Pontryagin's Approach

We want to find a sequence of control commands $\mathbf{u}(\cdot)$ that optimizes a criterion of long-term performance, while taking into account that the system evolves according to the model equations $\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, \mathbf{u})$. Here, we use a subtly different concept to describe long-term performance. Previously, the cost-to-go was just a function of each state of the system. The total cost J, by contrast, is a functional (a function of a function) of the entire system trajectory.

$$J[\mathbf{x}(\cdot), \mathbf{u}(\cdot)] = \int_{t_0}^{t_f} \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t)dt. \tag{1.1.8}$$

In the Dissertation Appendix, we derive the basic fact that we can use Lagrange multipliers to solve constrained optimization problems. The optima of a function $C(\mathbf{x})$, on the surface described by a constraint equation $s(\mathbf{x}) = 0$, satisfy $\nabla_{\mathbf{x}} C(\mathbf{x}) = -\lambda \nabla_{\mathbf{x}} s(\mathbf{x})$, where λ is an undetermined constant of proportionality, called a Lagrange multiplier. This implies that the optima of the augmented cost function $C(\mathbf{x}) + \lambda s(\mathbf{x})$ are the optima of our original cost function subject to the constraint. For equation 1.1.8, we use several Lagrange multipliers to impose constraints for each dimension of the dynamics equation.

$$\begin{aligned} J_{\text{constrained}}[\mathbf{x}(\cdot), \mathbf{u}(\cdot), \lambda(\cdot)] &= \int_{t_0}^{t_f} \left[\mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t) + \sum_i \lambda_i(t)(F_i(\mathbf{x}, \mathbf{u}) - \dot{x}_i) \right] dt \\ &= \int_{t_0}^{t_f} [\mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t) + \lambda(t)^\top (\mathbf{F}(\mathbf{x}, \mathbf{u}) - \dot{\mathbf{x}})] dt. \end{aligned} \tag{1.1.9}$$

Let us define a new quantity that we will also call a Hamiltonian. Here,

$$\hat{\mathcal{H}}(\mathbf{x}(t), \mathbf{u}(t), \lambda(t), t) = \mathcal{L}(\mathbf{x}(t), \mathbf{u}(t), t) + \lambda(t)^\top \mathbf{F}(\mathbf{x}, \mathbf{u}).$$

We will clarify the relationship between this Hamiltonian $\hat{\mathcal{H}}$ and the one in equation 1.1.5 in the next section. In terms of this Hamiltonian, equation 1.1.10 reads

$$J_{\text{constrained}}[\mathbf{x}(\cdot), \mathbf{u}(\cdot), \lambda(\cdot)] = \int_{t_0}^{t_f} [\hat{\mathcal{H}}(\mathbf{x}(t), \mathbf{u}(t), \lambda(t), t) - \lambda^\top \dot{\mathbf{x}}] dt. \tag{1.1.10}$$

We can state the equation solely in terms of the state variables, ridding ourselves of $\dot{\mathbf{x}}$, by integrating by parts.

$$J_{\text{constrained}}[\mathbf{x}(\cdot), \mathbf{u}(\cdot), \lambda(\cdot)] = \int_{t_0}^{t_f} [\hat{\mathcal{H}}(\mathbf{x}(t), \mathbf{u}(t), \lambda(t), t) + \dot{\lambda}^\top \mathbf{x}] dt - [\lambda^\top \mathbf{x}]_{t_0}^{t_f}.$$

At optima of this equation, any slight variation of the arguments to the total cost does not change it to first order:

$$\begin{aligned} J_{\text{constrained}}[\mathbf{x}(\cdot) + \delta\mathbf{x}(\cdot), \mathbf{u}(\cdot) + \delta\mathbf{u}(\cdot), \lambda(\cdot) + \delta\lambda(\cdot)] &= J_{\text{constrained}} + \delta J_{\text{constrained}} + O(\delta J_{\text{constrained}}^2) \\ &= J_{\text{constrained}} + O((\delta\mathbf{x}, \delta\mathbf{u}, \delta\lambda)^2). \end{aligned}$$

Or, for each t, $\delta J_{\text{constrained}}/\delta\mathbf{x}(t) = 0$, $\delta J_{\text{constrained}}/\delta\mathbf{u}(t) = 0$, and $\delta J_{\text{constrained}}/\delta\lambda(t) = 0$. From the first two, we conclude that

$$\begin{aligned} \dot{\lambda}(t) &= -\delta\hat{\mathcal{H}}/\delta\mathbf{x}(t), \\ \delta\hat{\mathcal{H}}/\delta\mathbf{u}(t) &= 0. \end{aligned}$$

To evaluate the variation with respect to the Lagrange multiplier, we use equation 1.1.10. We have

$$\dot{\mathbf{x}}(t) = \delta\hat{\mathcal{H}}/\delta\lambda(t).$$

Consolidating, we have

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \delta\hat{\mathcal{H}}/\delta\lambda(t), \\ \dot{\lambda}(t) &= -\delta\hat{\mathcal{H}}/\delta\mathbf{x}(t), \\ \delta\hat{\mathcal{H}}/\delta\mathbf{u}(t) &= 0. \end{aligned}$$

These equations have a pleasing anti-symmetry. The Lagrange multiplier is sometimes called the "costate" for its anti-symmetric relationship to the state, $\mathbf{x}$.[8] These equations only specify a minimum of the total cost if $\delta^2\hat{\mathcal{H}}/\delta\mathbf{u}(t)^2 > 0$, which is known as the Legendre-Clebsch condition (Stengel, 1994). When we cannot solve for the minimum algebraically because the equation $\delta\hat{\mathcal{H}}/\delta\mathbf{u}(t) = 0$ is complicated, we can still satisfy the state and costate differential equations and use $\delta\hat{\mathcal{H}}/\delta\mathbf{u}(t)$ to reduce the total cost by a gradient-based procedure.

The boundary conditions that appeared when we integrated by parts are important as well. We see that $\lambda(t_f)^\top\delta\mathbf{x}(t_f) = 0$. Since the state at the final time is free to vary, we conclude that $\lambda(t_f) = 0$. However, the state at the initial time t_0 is not free to vary, so the corresponding equation $\lambda(t_0)^\top\delta\mathbf{x}(t_0) = 0$ is trivially satisfied by $\delta\mathbf{x}(t_0) = 0$. If we were free to vary the state at the initial time however we wanted, the change in the total cost $\delta J_{\text{constrained}}$ would be exactly $\lambda(t_0)^\top\delta\mathbf{x}(t_0)$. This applies for any direction of variation, so we can say that on the optimal trajectory

$$\delta J_{\text{constrained}}/\delta\mathbf{x}(t_0) = \lambda(t_0). \tag{1.1.11}$$

Our argument here makes no significant assumption about t_0; we therefore conclude that

[8] Physicists may recognize that the costate serves the same role in these equations as does the momentum in Hamiltonian mechanics.

$\delta J_{\text{constrained}}/\delta \mathbf{x}(t) = \lambda(t)$ for general t.

Since the boundary condition for $\mathbf{x}$ is prescribed only at time t_0, we must solve the state equations forward-in-time. The boundary condition for λ is prescribed at the final time, t_f. We must integrate the costate equations backward-in-time. This is analogous to the backward-in-time calculation of dynamic programming. On the optimal trajectory, the value of the cost-to-go as a function of the state equals the total cost. Thus, we see from equation 1.1.11 that Pontryagin's method propagates derivatives of the value function with respect to the state backward-in-time, while the dynamic programming approach propagates the value function itself.

Deriving the Minimum Principle from the Hamilton-Jacobi Bellman Equation

If we differentiate the left side of the Hamilton-Jacobi Bellman equation 1.1.7 with respect to the state, we get $-\partial^2 V(\mathbf{x}^*(t), t)/\partial t \partial \mathbf{x}$. Because

$$\frac{d}{dt}\frac{\partial V(\mathbf{x}^*(t), t)}{\partial \mathbf{x}} = \frac{\partial^2 V(\mathbf{x}^*(t), t)}{\partial t \partial \mathbf{x}} + \frac{\partial^2 V(\mathbf{x}^*(t), t)}{\partial \mathbf{x} \partial \mathbf{x}^\top}\mathbf{F}(\mathbf{x}^*(t), \mathbf{u}^*(t)),$$

we can assert that

$$\frac{d}{dt}\frac{\partial V(\mathbf{x}^*(t), t)}{\partial \mathbf{x}} - \frac{\partial^2 V(\mathbf{x}^*(t), t)}{\partial \mathbf{x} \partial \mathbf{x}^\top}\mathbf{F}(\mathbf{x}^*(t), \mathbf{u}^*(t)) = -\frac{\partial \mathcal{H}(\mathbf{x}^*(t), \mathbf{u}^*(t))}{\partial \mathbf{x}}.$$

If we unpack the definition of the right-hand side, we have

$$\begin{aligned} -\frac{\partial \mathcal{H}(\mathbf{x}^*(t), \mathbf{u}^*(t))}{\partial \mathbf{x}} &= -\frac{\partial}{\partial \mathbf{x}}\left[\mathcal{L}(\mathbf{x}^*(t), \mathbf{u}^*(t)) + \frac{\partial V(\mathbf{x}^*(t), t)}{\partial \mathbf{x}}^\top \mathbf{F}(\mathbf{x}^*(t), \mathbf{u}^*(t))\right] \\ &= -\frac{\partial \mathcal{L}(\mathbf{x}^*(t), \mathbf{u}^*(t))}{\partial \mathbf{x}} - \frac{\partial^2 V(\mathbf{x}^*(t), t)}{\partial \mathbf{x} \partial \mathbf{x}^\top}\mathbf{F}(\mathbf{x}^*(t), \mathbf{u}^*(t)) \\ &\quad -\frac{\partial \mathbf{F}(\mathbf{x}^*(t), \mathbf{u}^*(t))}{\partial \mathbf{x}}^\top \frac{\partial V(\mathbf{x}^*(t), t)}{\partial \mathbf{x}}. \end{aligned}$$

Both the left-hand side and the right-hand side have the same term involving the Hessian of the value function, which can be cancelled, leaving

$$\frac{d}{dt}\frac{\partial V(\mathbf{x}^*(t),t)}{\partial \mathbf{x}} = -\frac{\partial \mathcal{L}(\mathbf{x}^*(t),\mathbf{u}^*(t))}{\partial \mathbf{x}} - \frac{\partial \mathbf{F}(\mathbf{x}^*(t),\mathbf{u}^*(t))}{\partial \mathbf{x}}^\top \frac{\partial V(\mathbf{x}^*(t),t)}{\partial \mathbf{x}}.$$

Define $\lambda(t) \equiv \partial V(\mathbf{x}^*(t),t)/\partial \mathbf{x}$. Then we recover,

$$\begin{aligned}\dot{\lambda} &= -\left[\frac{\partial \mathcal{L}(\mathbf{x}^*(t),\mathbf{u}^*(t))}{\partial \mathbf{x}} + \frac{\partial \mathbf{F}(\mathbf{x}^*(t),\mathbf{u}^*(t))}{\partial \mathbf{x}}^\top \lambda(t)\right] \\ &= -\frac{\partial \hat{\mathcal{H}}}{\partial \mathbf{x}}.\end{aligned}$$

This is the same equation we arrived at from the Minimum Principle. By the uniqueness theorem of ordinary differential equations, we can say that this λ is the same as the costate λ. Furthermore, the two Hamiltonians are really the same: $\hat{\mathcal{H}} = \mathcal{H}$.

1.1.4 Neural Network Models of Control

Equivalence of Backpropagation and Optimal Control

A discussion of the relationship between control theory and neural networks warrants an important digression. An interesting relationship exists between the minimum principle and the most widely used algorithm for training neural networks, backpropagation. Consider, for instance, the typical firing rate network studied in neuroscience:

$$\begin{aligned}\dot{\mathbf{x}} &= -\mathbf{x} + gJ\mathbf{r}, \\ \mathbf{r}(\mathbf{x}) &= \tanh(\mathbf{x}).\end{aligned}$$

These implicitly express a single functional form for the time derivative of the rates, $\dot{\mathbf{r}}$, dependent on the parameters, J. Suppose we have some particular Lagrangian, $\mathcal{L}(\mathbf{r}, J)$, which penalizes various states of the network and synaptic coupling strengths. (This penalty could express, for example, the squared difference between the rates of certain neurons and targets, while preventing the growth of large synapses.) Abstractly, of course, we just have a dynamical equation $\dot{\mathbf{r}} = \mathbf{F}(\mathbf{r}, J)$. Now, the coupling parameters serve the same role as the control variables do. The only minor difference is that the coupling parameters in typical models do not change at every time step of the simulation.[9] The optimization problem is exactly the same, except that J is not taken to be a function of time:

$$\int_{t_0}^{t_f} \Big[\mathcal{L}(\mathbf{r}, J) + \lambda^\top(\mathbf{F}(\mathbf{r}, J) - \dot{\mathbf{r}})\Big] dt.$$

For feedforward networks, we can also apply the same mathematical machinery (LeCun, 1988). In this case, instead of constructing a functional that integrates over time, we create a functional that sums over the activity at each layer

$$\sum_l \Big[\mathcal{L}(\mathbf{r}_l, J_l) + \lambda_l^\top(\mathbf{F}(\mathbf{r}_l, J_l) - \mathbf{r}_{l+1})\Big].$$

In Chapter 2 (Methods), we perform essentially this computation when we derive backpropagation-through-time for discrete-time systems.

[9] I expect people will relax this particular constraint in the future. At the same time, some dynamical meta-rule should then be imposed governing allowable changes in synapses. Otherwise, "learning" would involve no learning at all. The constrained problem in this case would look like

$$\int_{t_0}^{t_f} \Big[\mathcal{L}(\mathbf{r}(t), J(t)) + \lambda(t)^\top(\mathbf{F}(\mathbf{r}, J(t)) - \dot{\mathbf{r}}(t)) + \mathbf{p}(t)^\top(\mathbf{G}(\mathbf{r}(t), J(t)) - \dot{J}(t))\Big] dt.$$

The synaptic evolution rule $\mathbf{G}$ could be Hebbian or anything else. Hidden variables could also be incorporated into the synapses to create more memory.

Using Neural Networks to Control Other Systems

Broadly speaking, there are also two ways of performing optimal control computations using neural networks, corresponding respectively to the Bellman approach and the Pontryagin approach to optimal control. In the Bellman approach, the cost-to-go or value function is approximated with a neural network (Doya, 2000) (Stengel, 1994). This is usually done by some variation on "temporal difference" learning (Sutton and Barto, 1998). These methods are also known as "approximate dynamic programming" (Werbos, 1992). The details tend to vary, but one typically has a controller that takes in features of the state and produces the control command, $u_\pi(\mathbf{x}; \Theta_\pi)$. π denotes that this is a policy with associated parameters Θ_π. One also has a value function that tries to learn the cost-to-go of that policy, $V_\pi(\mathbf{x}; \Theta_V)$. The typical learning process is to make sure that the value function satisfies some variant of the Belman or Hamilton-Jacobi-Bellman equation. For example, in discrete-time, one can define the "temporal difference" error

$$\text{TD} = \left[V_\pi(\mathbf{x}; \Theta_V) - C(\mathbf{x}, \mathbf{u}) - V_\pi(\mathbf{F}(\mathbf{x}, \mathbf{u}); \Theta_V) \right]^2.$$

The cost-to-go of the present state is equal to the cost of that state plus the cost-to-go of the next state. The amount by which this is violated on successive time steps of operation is the error of the value function network, the "TD" error. To perform control, one simultaneously tries to minimize the value function by changing the policy parameters while also trying to estimate the value of the policy at the current set of parameters. This creates a moving target learning problem for both the controller and the value function network: the value function's learning depends on the controller, and the controller's learning depends on the value function. Often, these methods are not actually guaranteed to converge. If one manages to succeed, controller optimization now does not need to simulate the system dy-

namics at all because the Hamiltonian formed using the value function captures all relevant information for planning. Sutton and colleagues have recently claimed to have solved some of the convergence issues that have plagued value function methods (Maei et al., 2009), but they have not yet applied their new methods to objectively more difficult problems. The exact mechanisms used are somewhat tricky and the methods of analysis somewhat baroque, so we consider them incidental to this thesis.

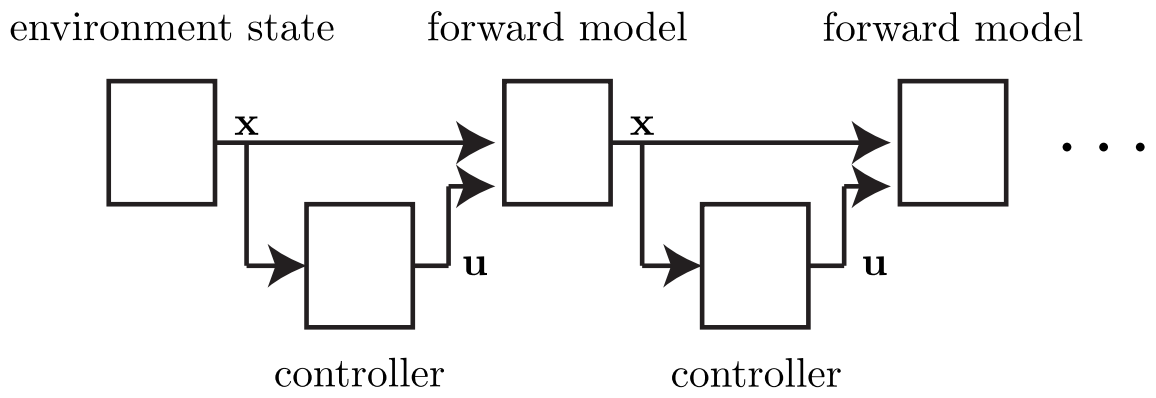

Figure 1.3: Forward Model and Controller.
Neural network modeling of motor control has achieved its greatest successes using optimization algorithms that are similar to the ones used in other machine learning tasks (Jordan and Rumelhart, 1992) (Sutskever, 2013) (Huh and Todorov, 2009), making use of forward models of the plant within the optimization. An optimization trial is depicted in which the veridical state of the system initializes a forward model. The controller and forward model recurrently interact over several time steps (left to right), and the parameters of the controller are optimized based on the evaluation of the total cost.

In the approach based on the Minimum Principle, one builds a forward model of the system using a neural network and then performs the standard computation using the Minimum Principle (Backpropagation-through-Time) (Figure 1.3). This approach was developed in the neural network community by (Nguyen and Widrow, 1989) and (Jordan and Rumelhart, 1992). Largely, control theorists avoid the use of neural networks for control, but when they do use them, this approach has shown the greatest capability so far. It does not suffer from the same issues of convergence. The principal difficulty in this case is to learn an

accurate forward model. If the forward model can be learned easily, one optimizes the total cost using the forward model either to find a sequence of control commands, $\mathbf{u}(\cdot)$, or the parameters of a fixed controller, $\mathbf{u}(\mathbf{x};\Theta)$.

REINFORCE

Williams (Williams, 1992) invented a method for reinforcement learning in stochastic systems. Here, we assume we have a stochastic policy $\Pr(\mathbf{u}_k|\mathbf{x}_k,\Theta)$ and a potentially stochastic environment $\Pr(\mathbf{x}_{k+1}|\mathbf{x}_k,\mathbf{u}_k)$. The total cost in a single episode is $J = \sum_l \mathcal{L}(\mathbf{x}_{l+1},\mathbf{u}_l)$. The expected total cost of this policy is

$$\begin{aligned}\langle J\rangle &= \left\langle \sum_l \mathcal{L}(\mathbf{x}_{l+1},\mathbf{u}_l)\right\rangle \\ &= \prod_{k=1}^{K}\int d\mathbf{u}_k \int d\mathbf{x}_{k+1}\,\Pr(\mathbf{u}_k|\Theta,\mathbf{x}_k)\Pr(\mathbf{x}_{k+1}|\mathbf{x}_k,\mathbf{u}_k)\sum_l \mathcal{L}(\mathbf{x}_{l+1},\mathbf{u}_l),\end{aligned}$$

where we have defined the Lagrangian on separate time steps of the state and control for convenience. The gradient of the expected cost with respect to Θ is

$$\begin{aligned}\nabla_\Theta\langle J\rangle &= \sum_{j=1}^{K}\int d\mathbf{u}_j\int d\mathbf{x}_{j+1}\nabla_\Theta\Pr(\mathbf{u}_j|\Theta,\mathbf{x}_j)\Pr(\mathbf{x}_{j+1}|\mathbf{x}_j,\mathbf{u}_j) \\ &\quad\times\prod_{k\neq j}^{K}\int d\mathbf{u}_k\int d\mathbf{x}_{k+1}\,\Pr(\mathbf{u}_k|\Theta,\mathbf{x}_k)\Pr(\mathbf{x}_{k+1}|\mathbf{x}_k,\mathbf{u}_k)\sum_l\mathcal{L}(\mathbf{x}_{l+1},\mathbf{u}_l) \\ &= \sum_{j=1}^{K}\int d\mathbf{u}_j\int d\mathbf{x}_{j+1}\nabla_\Theta\ln(\Pr(\mathbf{u}_j|\Theta,\mathbf{x}_j))\Pr(\mathbf{u}_j|\Theta,\mathbf{x}_j)\Pr(\mathbf{x}_{j+1}|\mathbf{x}_j,\mathbf{u}_j) \\ &\quad\times\prod_{k\neq j}^{K}\int d\mathbf{u}_k\int d\mathbf{x}_{k+1}\,\Pr(\mathbf{u}_k|\Theta,\mathbf{x}_k)\Pr(\mathbf{x}_{k+1}|\mathbf{x}_k,\mathbf{u}_k)\sum_l\mathcal{L}(\mathbf{x}_{l+1},\mathbf{u}_l) \\ &= \prod_{k=1}^{K}\int d\mathbf{u}_k\int d\mathbf{x}_{k+1}\,\Pr(\mathbf{u}_k|\Theta,\mathbf{x}_k)\Pr(\mathbf{x}_{k+1}|\mathbf{x}_k,\mathbf{u}_k)\sum_{j=1}^{K}\nabla_\Theta\ln(\Pr(\mathbf{u}_j|\Theta,\mathbf{x}_j))\sum_l\mathcal{L}(\mathbf{x}_{l+1},\mathbf{u}_l).\end{aligned}$$

Define $\mathbf{e}_j^\Theta \equiv \nabla_\Theta \ln(\Pr(\mathbf{u}_j|\Theta, \mathbf{x}_j))$. Then

$$\nabla_\Theta \langle J \rangle \;=\; \left\langle J \sum_j \mathbf{e}_j^\Theta \right\rangle.$$

This is a reinforcement learning rule in which the product of the total cost is taken with the sum of the quantities $\mathbf{e}_j^\Theta$, known as eligibility traces. REINFORCE is not a very efficient algorithm on its own, but we mention it because it does permit the learning of goal-directed behaviors and its offshoots have become very popular in theoretical neuroscience.

1.1.5 Overview of this Dissertation

The unifying thread in this dissertation is the internal model. We have already discussed the utility of internal models in engineering and, more briefly, the types of evidence for them amassed in motor control. To date, the internal model that has been most widely considered is the "forward model," a model that predicts the sensory consequences of motor commands.

In this dissertation we develop two more kinds of internal model. The fundamental idea that powers our inventiveness is the belief that different parts of the brain may model one other, in the same way that motor control researchers believe the central nervous system models its periphery.

In Chapter 2, we consider the question of how the nervous system can use hierarchy to "divide-and-conquer." We build a controller network for motor control with the use of standard optimal control calculations. We ask whether this controller network can in turn be controlled by propagating commands to it. To understand the sensory consequences of these commands, we must build another model, a "higher-level forward model." Given our higher-level forward model, we can again perform the computations of optimal control at

this second level. Our procedure for constructing hierarchies is thus recursive.

By introducing hierarchy, we increase the level of abstraction at which decisions are made. The lower-level controller in our simulation learns how to drive a semi-truck in reverse toward different goal locations. It lives in a world of steering wheels and short time-scales. The higher-level controller learns the effect of asking the lower-level to drive to particular locations. It deals with time scales that are orders of magnitude larger and problem domains of considerably greater complexity. In a limited way, this system can plan about the deeper future.

Because our control systems are neural networks, we inherit many of the beautiful properties of these devices. We show that we can implement a simple but high-dimensional sensory system that allows the higher-level controller to keep track of obstacles in its environment. Collectively, the two controllers can navigate around obstacles toward distant goals, even though the lower-level controller is completely ignorant of those obstacles. To our knowledge, our system breaks new ground in the combination of optimal control, hierarchy, and high-dimensional sensory perception.

The algorithmic computations of optimal control, even when embedded in neural networks, are widely perceived to be "biologically implausible." When these models are used as explanatory models in neuroscience, the conventional defense is that only the fully-optimized controller network stands as a model of the motor system. We quote Todorov (Todorov, 2008): "Such optimization is not meant to model the process of biological learning but rather the outcome of that process." There is admittedly much value to be gained from separating the problem of execution from the problem of optimization. As contended repeatedly, high human task-performance does not guarantee the existence of analogues to our conventional optimization algorithms in the nervous system.[10] Additionally, without

[10]The brain has been optimized over evolutionary time, not just the time to plan a single reach. Or, to put

making claims about how the neural responses arise through learning and development, features of the trained models can be compared with neural recordings and questions about the dynamics of neural computation can be answered (Sussillo and Barak, 2013).

Still, despite our uncertainty about the entire enterprise, we view it as a disappointment that the question of how the nervous system optimizes motor performance has been largely tabled for an unknown, later date. Todorov's words above are very similar to the sentiments of Crick (Crick, 1989), a full 19 years earlier. Although proof of online optimization in neural systems is inconclusive, the very flexibility of motor behavior – in particular the ability to take on new goals and achieve them rapidly – suggests that a search for optimization schemes in the nervous system is warranted and not a counter-productive fantasy.

In Chapter 3 of this thesis, we take on the challenge of articulating what features of optimal control calculations are the least "biologically plausible" and attempt to construct an efficient neural network controller that eschews them. We show that we can emulate the computations of optimal control, entirely within neural networks, and apply our construction to two prototypical problems in optimal control. The workhorse of our construction is called a "sensitivity model" – a model of the Jacobian matrix of the forward model itself. Implicitly, anyone who has ever programmed an optimal control calculation has built a sensitivity model without knowing it. We show that a very simple procedure can embed a sensitivity model in a neural network. We couple this sensitivity model with a lesser-known arrangement of the calculations of optimal control that works forward-in-time, in contrast to most common solution methods for optimal control problems, while still permitting goal-directed planning.

Before we begin in earnest, we offer one more cautionary, albeit radical, perspective about

it more sarcastically, no one has proposed that the cockroach brain computes its escape response using the literal computations involved in optimal control, despite the response's evident effectiveness (Vaidyanathan et al., 2012).

optimization in the brain. The universal computer has also proven capable of producing optimized motions in robotics and in simulation without any architectural bias toward executing optimization algorithms. Instead, the computer joins a finite state machine (the central processing unit) with a read-write memory (the registers, cache, random-access memory, and other assorted drives). The joining of a finite state machine with a read-write memory enables the computer to construct arbitrary data structures and run arbitrary algorithms (Minsky, 1967) (**?**) (Hillis and Hart-Davis, 1998), among them the algorithms that control physical plants. If we grant the plausibility of read-write memory, many of the suspiciously implausible algorithms used in computer science begin to look plausible after all. They can run in the brain as software, not hardware. The relative slowness of serial processing in the nervous system compared to the clock-speed of a modern computer suggests that considerations of biological plausibility should turn less on spatial and temporal "locality" of processing; the relevant criterion for the plausibility of an algorithm is whether it can be achieved quickly enough with components that cycle at the time scale of milliseconds instead of nanoseconds.

Chapter 2

A Design Procedure For Hierarchical Neural Control

If there is a problem you can't solve, then there is an easier problem you can solve: find it. – George Pólya

It seems that any systematic formulation of the adaptive control problem leads to a meta-problem which is not adaptive. – J.J. Florentin

A Design Procedure for Hierarchical Neural Control

Greg Wayne[1] **and L.F. Abbott**[1,2]

[1] Department of Neuroscience

[2] Department of Physiology and Cellular Biophysics

Columbia University College of Physicians and Surgeons

New York, NY 10032-2695 USA

Keywords: Optimal Feedback Control, Internal Models, Neural Networks

Abstract

We propose and develop a hierarchical approach to network control of complex tasks. In this approach, a low-level controller directs the activity of a “plant," the system that performs the task. However, the low-level controller may only be able to solve fairly simple problems involving the plant. To accomplish more complex tasks, we introduce higher-level controllers, i.e. controllers of controllers, that divide the overall task into simpler sub-tasks. Each controller issues commands to the controller below it in the hierarchy and receives commands from the controller above it. These commands set task sub-goals that become more ambitious as the hierarchy is ascended. The command received by the highest-level controller is the final goal of the task itself. We use a system based on this idea to direct an articulated truck to a specified location through an environment filled with static or moving obstacles. The final system consists of networks that have memorized associations between the sensory data they receive and the commands they issue. These networks are trained on a set of optimal associations that are generated by minimizing cost functions.

Cost function minimization requires predicting the sensory consequences of sequences of commands, which is achieved by constructing forward models, including models of the controllers themselves. The forward models and cost minimization are only used during training, allowing the trained networks to respond rapidly. The resulting system divides complex tasks into more manageable sub-tasks, and the optimization procedure and the construction of the forward models and controllers are performed in similar ways at every level of the hierarchy. This allows the system to be modified to perform other tasks or to be extended for more complex tasks without retraining lower-level elements.

Introduction

A common strategy used by humans and machines for performing complex, temporally extended tasks is to divide them into sub-tasks that are more easily and rapidly accomplished. In some cases, the sub-tasks themselves may be quite difficult and time-consuming, making it necessary to further divide them into sub-sub-tasks. This approach can be iterated as many times as necessary until the task is manageable. We mimic this strategy to design a hierarchical control system. The top-level controller in such a hierarchy receives an external command that specifies the overarching task objective, whereas the bottom-level controller issues commands that actually generate actions. A series of controllers acts between these extremes to sub-divide the task into successively simpler and less time-consuming sub-tasks. At each level, a controller receives a command from the level immediately above it describing the goal it is to achieve and issues a command to the controller immediately below it describing what that controller is supposed to do.

We design neural networks that use this hierarchical strategy to perform control tasks. The tasks we consider are dynamic and ongoing, so the commands describing the sub-goals must be generated continuously in time (actually at each small time step in our simulations). For this reason, each controller must come up with the command it issues rapidly. To realize the required speed, our controllers are neural networks that implement complex lookup tables. Each controller receives input describing the goal it is to achieve and, in addition, “sensory" input providing information about the environment relevant to achieving this goal. Its output is the command specifying the goal for the controller one level down in the hierarchy. Getting this to work requires training each controller to implement the appropriate lookup table and, most essentially, generating the data for this table. This two-stage training procedure is made considerably easier by the fact that each controller in the hierarchy is basically doing the same thing: receiving and issuing commands describing

goals. Thus, we can apply the same training procedure at each level. Another advantage of our approach is that lower-level controllers do not need to be retrained if the overall task changes. In addition, the hierarchy can be extended by adding more levels if the task gets more difficult, again without requiring retraining of the lower levels.

Generating data for the look-up table is the more involved of the two steps in our training procedure. These data consist of optimal output commands given a particular input command (goal) and particular sensory information. Optimality is defined by a cost function specified at each level. Optimization is achieved with the aid of a neural network implementing a forward model of the controller being commanded. The forward model is only used for optimization during learning; the fully-trained model consists only of the controller networks. The training procedure involves what is effectively a control-theory optimization in which the "plant" being controlled is actually the network controller at the next lower level of the hierarchy (except, of course, for the lowest-level controller in which case it is the actual plant performing the task). This research thus extends ideas about forward models and optimization from the problem of controlling a plant to that of controlling a controller. Once the optimal output commands are determined for a large set of input commands and sensory inputs, these are used as training data for the controller, which effectively "memorizes" them.

We apply this approach to a problem that requires two levels of control. The basic problem is to drive a simulated articulated semi-truck backward to a specified location that we call the final target location (the truck is driven backward because this is harder than driving forward). The backward velocity of the truck is held constant, so the single variable that has to be controlled is the angle of the truck's wheels. This problem was first posed and solved by Nguyen and Widrow (Nguyen and Widrow, 1989), and their work is an early example of the successful solution of a nonlinear control problem by a neural network. We

make this problem considerably harder by moving the final target location quite far away from the truck and, inspired by the swimmer of Tassa, Erez, and Todorov (Tassa et al., 2011), by distributing a number of obstacles across the environment. Although the lower-level controller can drive to a nearby location when no obstacles are in the way, it cannot solve this more difficult task. Thus, we introduce a higher-level controller that feeds a series of unobstructed, closer locations that we call sub-targets to the lower-level controller that generates the wheel-angle commands. The job of the higher-level controller is to generate a sequence of sub-targets that lead the truck to its ultimate goal, the final target location, without hitting any obstacles. Thus, we divide the problem into lower-level control of the truck and higher-level navigation.

Results

We begin by describing how the hierarchical approach, consisting of lower- and higher-level controllers, operates after both of these network controllers have been fully trained. We do this sequentially, first showing the lower-level controller operating the truck when its sub-target data are generated externally (by us), rather than by the higher-level controller. We then discuss how the higher-level controller generates a sequence of sub-target locations to navigate through the environment. To allow the higher-level controller to detect and locate obstacles, we introduce a sensory grid system. To complete this first section of the results, we present and analyze the complete hierarchical system with the two controllers working together. The bulk of the results, presented after we have shown the trained system in operation, covers the procedures and auxiliary networks used to train the controllers.

All of the networks we consider run in discrete time steps, and we use this step as our unit of time. All times are thus integers. Distance is measured in units such that the length of the truck cab is 6, the trailer is 14, and both have a width of 6. In these units, the backward speed of the truck is 0.2. The final target for the truck and the obstacles it must avoid have a

radius of 20. Distances from the initial position of the truck to the final target are typically in the range of 100 to 600.

Driving the Truck

Our hierarchical model for driving the truck (Figure 1) starts with a lower-level controller that sends out a sequence of commands $u(t)$ that determines the angle of the wheels of the truck. This controller is provided with "proprioceptive" sensory information, namely the cosine and sine of the angle between the cab and trailer of the truck. $[\cos(\theta_{\text{rel}}), \sin(\theta_{\text{rel}})]$, and a sub-target location toward which it is supposed to direct the truck (to ensure continuity and promote smoothness, we process all angles by taking their cosines and sines). The target information is provided as a distance from the truck to the sub-target, d_{st}/L ($L = 100$ is a scale factor), and the cosine and sine of the angle from the truck to the sub-target, $[\cos(\theta_{\text{st}}), \sin(\theta_{\text{st}})]$. This sub-target information is provided by a higher-level controller that receives the same proprioceptive input from the truck as the lower-level controller but also receives sensory information about obstacles in the environment (to be described later). In addition, the higher-level controller is provided with external information about the distance from the truck to the final target location and also the cosine and sine of the angle from the truck to this location, $[\log(1 + d_{\text{ft}}/L), \cos(\theta_{\text{ft}}), \sin(\theta_{\text{ft}})]$. The task of the higher-level controller is to provide a sequence of sub-targets to the lower-level controller that lead it safely past a set of obstacles to the final target location. Note that the higher-level controller receives the logarithm of the distance to the final target, $\log(1 + d_{\text{ft}}/L)$, rather than d_{ft}/L itself. This allows for operation over a larger range of distances without saturating the network activities. The logarithm is not needed for d_{st}/L because the distance to the sub-target is maintained within a constrained range by the higher-level controller.

The Lower-Level Controller

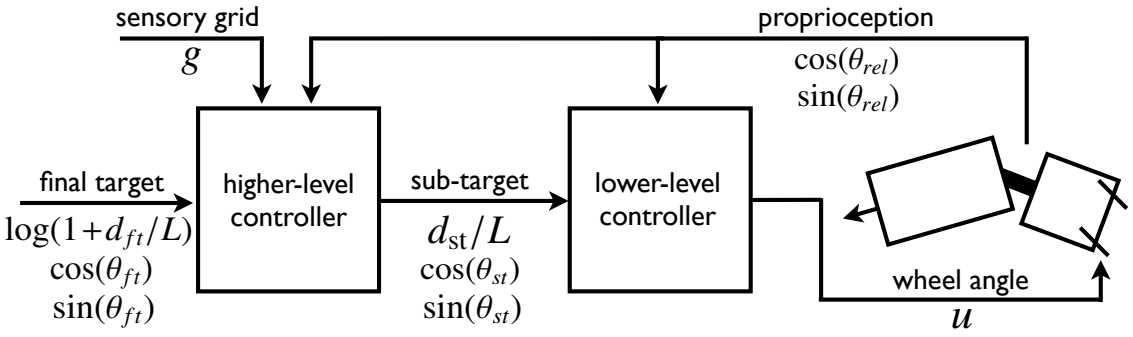

Figure 1. Flow diagram of the hierarchical control system. Commands that control the wheel angle of the truck are issued by the lower-level controller, which receives information about a sub-target direction toward which the truck should be driven from the higher-level controller. Both controllers receive proprioceptive information about the angle between the cab and trailer of the truck, and the higher-level controller also receives information about obstacles in the environment from a grid of sensors. In addition, the higher-level controller receives input about the final target that the truck is supposed to reach.

The job of the lower-level controller is to generate a sequence of wheel angles, $u(t)$, given the proprioceptive data, $[\cos(\theta_{\mathrm{rel}}(t)), \sin(\theta_{\mathrm{rel}}(t))]$, and a sub-target location specified by $[d_{\mathrm{st}}(t)$, $\cos(\theta_{\mathrm{st}}(t), \sin(\theta_{\mathrm{st}}(t))]$ (Figure 1). The proprioceptive information is needed by the controller not only to move the truck in the right direction but also to avoid jackknifing. The lower-level controller is a 3-layer basis-function network with the 5 inputs specified above, 100 Gaussian-tuned units in a hidden-layer, and 1 output unit that reports u as a linear function of its input from the hidden layer (Methods). Figure 2A shows an example in which the sub-target location is held fixed and the lower-level controller directs the truck along the backward path indicated by the curved line.

At this point, we are showing the lower-level controller working autonomously with the sub-targets specified by us, but when the truck is directed by the higher-level controller, it will be given a time-dependent sequence of sub-targets. To test whether it can deal with sequential sub-targets, we switched the sub-target we provide every time the truck got close to it, using a rather fanciful sequence of sub-targets (Figure 2B). This indicates that the lower-level control is up to the job of following the directions that will be provided by

Figure 2. A) The lower-level controller directs the truck to a sub-target (white square). The black trace shows the path of the back of the truck. B) The lower-level controller directs the truck to trace the constellation *Ursa Major* (white line is the path of the truck) by approaching sub-targets at the locations of the stars. The sub-targets appear one at a time; when the back of the truck arrives close to the current sub-target, it is replaced by the next sub-target. Photo by Akira Fujii.

the higher-level controller.

The Higher-Level Controller

The higher-level controller is a five-layer, feedforward network with a bottleneck architecture. It has 205 inputs (3 specifying the final target location, 2 the angle between the cab and trailer of the truck, 199 describing the state of the sensory grid described below, and a bias input; Figure 1), hidden layers consisting of 30, 20 and 30 units, and three command outputs providing the sub-target information for the lower-level network (Methods). The bottleneck layer with 20 units ensures that the network responds only to gross features in the input that reliably predict the desired higher-level command. When we initially trained the higher-level controller without any form of bottleneck, it did not generalize well to novel situations.

In the absence of any obstacles, the job of the higher-level controller is to provide a sequence of sub-targets to the lower-level controller that lead it to the location of the final target, which is specified by the variables $[d_{\text{ft}}, \cos(\theta_{\text{ft}}), \sin(\theta_{\text{ft}})]$ that the higher-level controller receives as external input. The higher-level controller also receives a copy of the proprioceptive input provided to the lower-level controller (Figure 1). Figure 3 shows a trajectory generated by the higher-level controller and the motion of the truck as directed

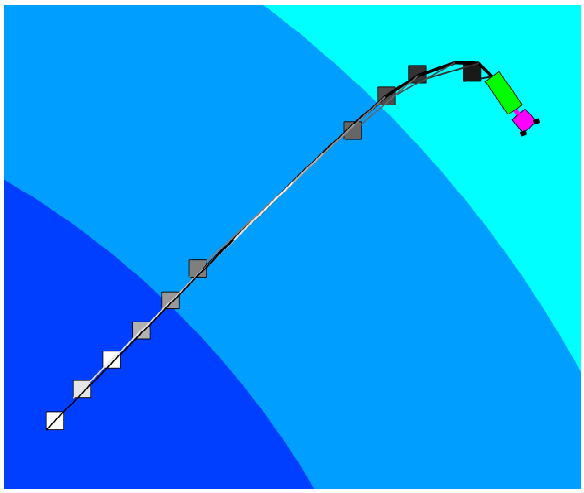

Figure 3. The truck following a sequence of sub-targets provided by the higher-level controller. The sub-targets are indicated by black/grey/white squares with darker colors representing earlier times in the sequence. The trajectory that the truck follows in pursuing the sub-targets is shown in black. A connecting line indicates the sub-target that is active when the truck reaches particular trajectory points. The background coloration indicates the distance to the final target, located off the lower-left corner.

by the lower-level controller, leading to a target just beyond the bottom-left corner of the plot. Note the sequence of target locations that lead the truck along the desired path. Although this example shows that the higher-level controller is operating as it should and that the lower-level controller can follow its lead, this task is quite simple and could be handled by the lower-level controller alone. To make the task more complex so that it requires hierarchical control, we introduced obstacles into the environment.

The obstacles are discs with the same radius as the final target scattered randomly across the arena (Figure 4B). These are soft obstacles that do not limit the movement of the truck, but during training we penalize commands of the higher-level controller that cause the truck to pass too close to them (see below). Making this environmental change requires us to introduce a sensory system that provides the higher-level controller with information about the locations of the obstacles. Just as the final and sub-target locations are provided in “truck-centric" coordinates (distances and angles relative to the truck), we construct this sensory system in a truck-centric manner (Figure 4A).

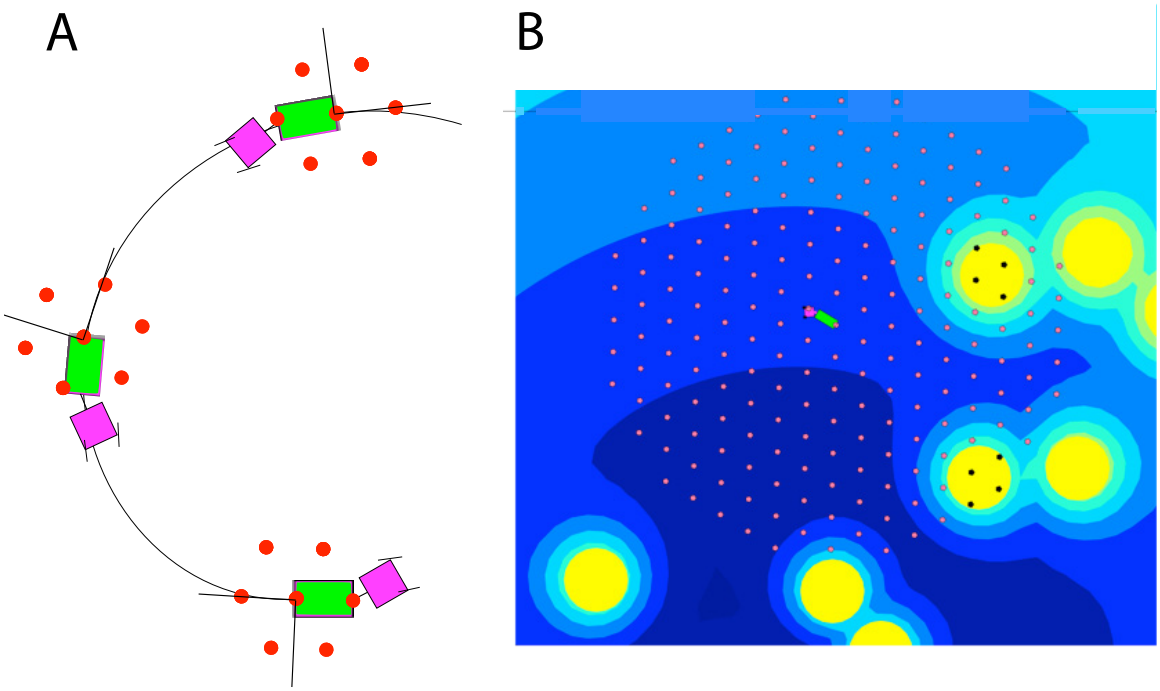

Figure 4. A) An egocentric coordinate system surrounds the truck, composed of grid points. During movement, the grid shifts with the truck. Only a small fraction of the grid points is shown here. B) The full set of grid points in an environment with obstacles (yellow circles). Those points that lie within an obstacle are blackened, indicating that the grid element is activated.

Specifically, we construct a hexagonal grid of points around the truck (Figure 4). The grid is a lattice of equilateral triangles with sides of length 20 units. One grid point lies at the back of the trailer, and the most distant grid points are 150 units away from this point. In total, there are 199 grid points. These points move with the truck and align with the longitudinal axis of the trailer (Figure 4A). If a grid point lies inside an obstacle, we consider it to be activated; otherwise, it is inactive. The state of the full grid is specified by an 199-component binary vector **g** with component i specifying whether grid point i is active ($g_i = 1$) or inactive ($g_i = 0$). Neither topological closeness nor Euclidean distance information is explicit in this vector representation. The grid vector is provided as additional input to the higher-level controller (Figure 1).

Operation of the Full System

We now show how the full system operates when the higher-level controller provides the lower-level controller with sub-targets as they drive the truck together through a field of obstacles to the final target (Figure 5A). Figure 5B shows a number of guided trajectories through an obstacle-filled arena.

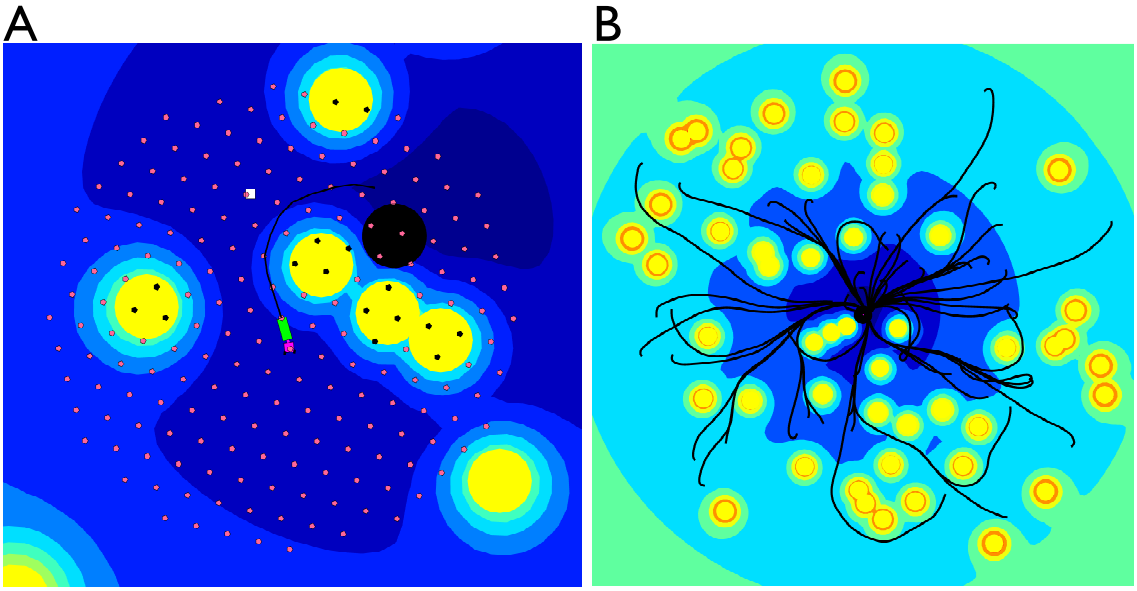

Figure 5. A) The truck is directed to avoid the obstacles and reach the final target. The white square indicates the first sub-target; note that it is not at a position the truck actually reaches. The higher-level controller merely uses this to indicate the desired heading to the lower-level controller. B) With 50 static obstacles, more than the 20 that were present during training, the higher-level controller steers the truck around all of the obstacles to the final target on each of 50 consecutive trials. The black lines show the paths taken by the back of the trailer.

To quantify the performance of the system, we executed 100 trials in 100 different environments with 20 obstacles. The hierarchical controller avoids the obstacles on each trial; the minimum distance to an obstacle never decreases below one obstacle radius (20 units; Figure 6A). As the number of environmental obstacles is increased (Figure 6B), the probability of obstacle collision grows slowly. This occurs even though the controllers were trained with only 20 obstacles in the environment. The control system directs the truck to the final target along short paths (Figure 6C) that are comparable in length to the straight line dis-

tance between the initial position and the nearest edge of the final target, with deviations when the truck must execute turning maneuvers or circumnavigate obstacles.

The trained higher-level controller continuously generates sub-targets based on the sensory information it receives (Video 1). Because all the contingencies are memorized, it needs very little time to compute these plans. Thus, the higher-level controller can respond quickly to changes in the environment, even though it was trained on data from static environments. To illustrate this, we tested the system with obstacles that moved around, although it was trained with stationary obstacles. The obstacle motions were generated as random walks. The probability of collision grows slowly with increasing diffusion constant of the random walk (Figure 6D). At high rates of diffusion, the obstacles move significantly further than the truck for small numbers of time steps. For example, when the diffusion coefficient is 1 (units [L^2/T]), the obstacles typically diffuse (but can diffuse farther than) the width of the trailer within 9 time steps. It takes the truck 30 time steps to travel the same distance. Another example can be seen in Video 2.

The Training Procedure

The controllers at each level of the hierarchy work because they have been trained to generate commands (sub-targets or wheel angles for the truck) that are “optimal" for the input they are receiving at a given time. Recall that this input consists of the sub-target received from the upstream controller and whatever sensory information is provided. In the following, the combination of the sensory data and the target-related information that form the input to a controller is referred to as "sensory" input, even though the target information comes in the form of a command. For most of this section, we discuss how we define and generate optimal commands for a large number of input conditions, but, for now, assume that these data are available. Then, training each controller is straightforward. We apply a particular input to the controller network and use backpropagation to modify its param-

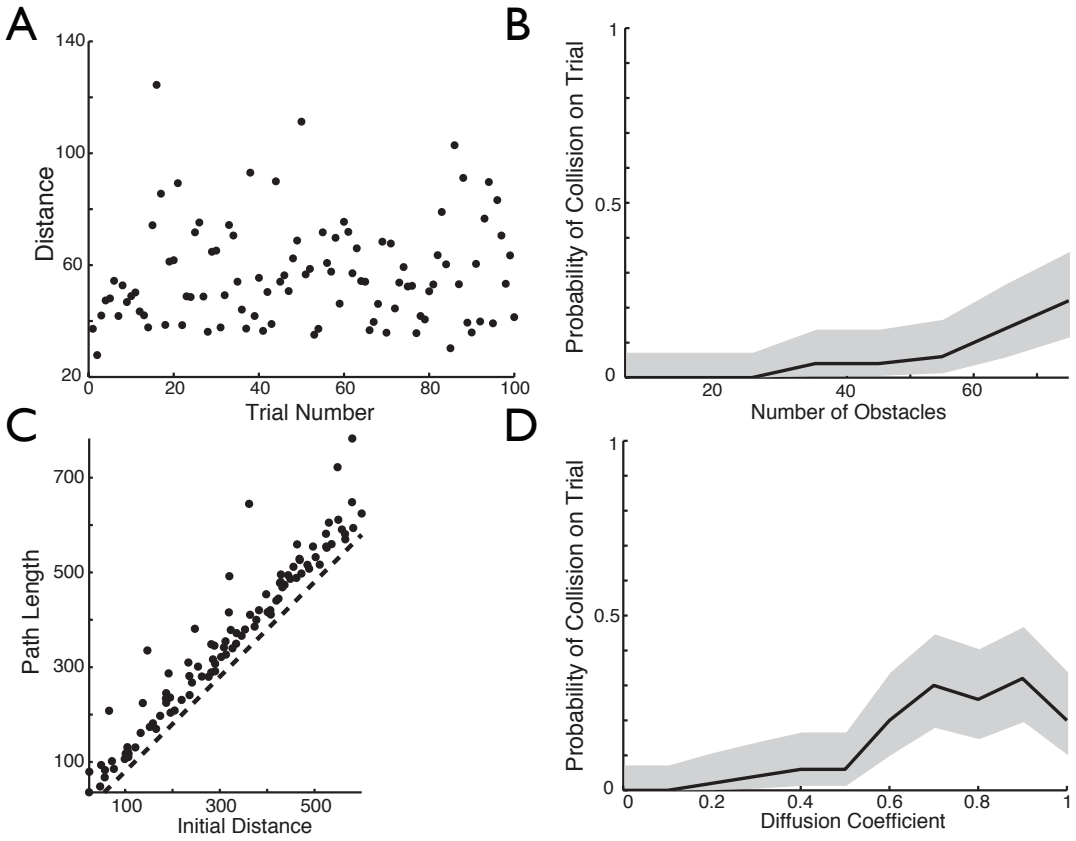

Figure 6. Performance measures. A) Obstacle Avoidance. Black dots show the minimum distance between the truck and any target averaged over 50 runs to the final goal with 20 obstacles in the environment. B) Collisions versus Obstacles. The black line shows the probability of a collision with an obstacle per trip to the final target as a function of the number of obstacles. The shaded regions are 95% confidence intervals. As the number of obstacles increases from 5 to 75, the probability of a collision grows slowly, despite high obstacle densities. C) Target-directedness. The black dots show the lengths of paths taken to the final target, averaged over 50 trials, in an environment with 20 obstacles. The dashed line show the straight-line distance from the initial location of the truck to the nearest edge of the final target. D) Brownian Obstacle Motion. The black line shows the probability of a collision with an obstacle per trip as a function of the diffusion constant of the obstacle motion. The grey region is as in B. Although we did not explicitly train the controllers to handle obstacle movement, the controller can frequently navigate to the goal without collision in an environment of 20 obstacles undergoing Brownian motion.

eters in order to minimize the squared difference between the output command given by the controller and the optimal command for that particular set of inputs. After a sufficient number of such trials (Methods), the controller network learns to produce the desired command in response to a particular input. Furthermore, if the network is properly designed it

will generalize to novel inputs by smoothly interpolating among the trained examples. The previous section showed that this indeed works. Thus, we turn to the problem of generating the optimal commands that provide the training data for the controllers.

One feature that makes tasks difficult is that a significant amount of time may elapse before the cost associated with a particular command strategy can be assessed. In our lower-level example, it takes a while for the truck to move far enough to reveal that the wheels are not at a good angle. Typically, as tasks get more complex, this delay gets longer. For example, it takes longer to evaluate whether a sub-target issued by the higher-level controller is going to get the truck closer to the final target without leading it into an obstacle. One consequence of this delay is that we cannot assess the cost associated with a single command; we must evaluate the cost of a sequence of commands.

To deal with the hierarchy of timescales that are associated with a hierarchy of control levels, we introduce two time scales. One is associated with the temporal scale over which a process needs to be controlled. There is no point in issuing commands that change more rapidly than the dynamics of the object being controlled. In the truck example, wiggling the wheels back and forth rapidly is not an intelligent way to drive the truck, and swinging the sub-target around wildly is not a good way to guide the lower-level controller. At level l of the hierarchy, we call this dynamic timescale T_l. For the truck problem, we take $T_1 = 6$ and $T_2 = 72$ time steps. The second time scale is the length of the sequence of commands needed to compute a cost reliably. At level l, we denote this number by K_l. In other words, it requires K_l commands, spaced apart by T_l times steps, to determine the cost of a particular command strategy. In the case of the truck, we set $K_1 = 15$ and $K_2 = 10$.

At each level, l, we must solve two problems: 1) Find a cost function that allows the system to achieve the final goal of the task, and 2) Given the sensory input at time t, find the set of K_l commands, one every T_l time steps, that minimizes this cost function.

We start by addressing problem 2, leaving a discussion of the cost function to the following section. We denote a command given at time t by the level l controller by the vector $\mathbf{m}_l(t)$. For the lower-level controller in the truck example, $\mathbf{m}_1(t) = u(t)$, and for the upper-level controller $\mathbf{m}_2(t) = [d_{\text{st}}(t), \cos(\theta_{\text{st}}(t)), \sin(\theta_{\text{st}}(t))]$. We want to compute the cost of issuing a sequence of commands $[\mathbf{m}_l(t), \mathbf{m}_l(t + T_l), \mathbf{m}_l(t + 2T_l), \ldots, \mathbf{m}_l(t + K_l T_l)]$. We also need to define a vector $\mathbf{s}_l(t)$ that represents the sensory input to layer l at time t upon which the decision to issue the command sequence is based. For the case of the truck, $\mathbf{s}_1(t) = [\cos(\theta_{\text{rel}}), \cos(\theta_{\text{rel}}), d_{\text{st}}, \cos(\theta_{\text{st}}), \sin(\theta_{\text{st}})]$ and $\mathbf{s}_2 = [\cos(\theta_{\text{rel}}), \cos(\theta_{\text{rel}}), \mathbf{g}, \log(1 + d_{\text{ft}}/L), \cos(\theta_{\text{ft}}), \sin(\theta_{\text{ft}})]$ (figure 1). The cost function takes the general form

$$\mathcal{S}_l = \sum_{k=0}^{K_l} \mathcal{L}_l(\mathbf{s}_l(t + (k + 1)T_l), \mathbf{m}_l(t + kT_l)) \,. \tag{2.2.1}$$

The functions $\mathcal{L}_l$ for the lower- ($l = 1$) and higher- ($l = 2$) level controllers are specified in the following section.

Equation 2.2.1 raises a significant new problem: the cost function depends not only on the sequence of commands being given but also on the entire sequence of sensory consequences of those commands. In other words, $\mathcal{S}_l$ depends on $\mathbf{s}_l(t) \ldots \mathbf{s}_l(t + K_l T_l)$, and only $\mathbf{s}_l(t)$ is provided. We therefore need a way to predict the future sensory data resulting from the command sequence. We do this by building a forward model at each level of the hierarchy.

It is important to note that, for the truck problem, the future sensory inputs, for the purpose of command optimization, are defined slightly differently from the actual sensory inputs used by the controllers during their normal operation. Recall that the sensory command includes information about the distance and angle to the target (either the sub-target for level 1 or the final target for level 2). The actual target location is held fixed during training, but because the truck moves, the distance and angle to this location change. Thus, when the

forward model predicts the future sensory data, it estimates the actual distance and cosine and sine of the angle to the target.

We have now addressed the second problem listed above; we determine an optimal sequence of commands by minimizing the cost function of equation 2.2.1 using a forward model to predict the resulting sensory inputs. In the following section, we present the cost functions used for the truck-driving task. In the section after that, we discuss how we build the required forward models. Before proceeding, however, it is useful to clarify the full process for training the controllers in light of the procedure just described for computing optimal command sequences.

The same procedure applies at every level. We randomly select an input vector $\mathbf{s}(t)$ and determine the optimal command sequence arising from that initial input. We then train the controller to associate this particular sensory input with the first motor command in the sequence. Note that we do not use the full sequence of optimal commands for training the controller, we only use the full sequence to evaluate the cost function. The reason for this is that all the future commands beyond the first one in the sequence are based on sensory inputs predicted by the forward model. Training the controller to associate these predicted inputs with their paired commands in the sequence introduces errors into the controller, because the sensory predictions of the forward model are not entirely accurate. Learning these predicted associations degrades the performance of the controller.

The Cost Functions

We would like the truck to drive toward the target along a straight angle-of-attack, without articulating the link between the cab and trailer too much, and using minimal control effort. A cost function satisfying these requirements can be constructed from

$$\mathcal{L}_1 = \alpha_1 d_{\text{st}} + \beta_1 \theta_{\text{st}}^2 + \gamma_1 (|\theta_{\text{rel}}| - \theta_{\max})^2 \Theta (|\theta_{\text{rel}}| - \theta_{\max}) + \zeta_1 u^2. \tag{2.2.2}$$

The third term makes use of the Heaviside step function, which is 1 if $x \geq 0$ and 0 otherwise. We use the convention that all angles are in radians, and we center all angles around 0 so they fall into the range between $\pm\pi$. The parameters α_1, β_1, γ_1, ζ_1, and $\theta_{\max}$ are given in Table 1 of the Methods.

The higher-level cost function is divided into three parts: $\mathcal{L}_2 = \mathcal{L}_2^{\text{sensory}} + \mathcal{L}_2^{\text{command}} + \mathcal{L}_2^{\text{obstacle}}$. All the parameters in these cost functions are given in Table 1 of the Methods. The sensory cost contains a distance-dependent term, but we no longer need to penalize large cab-trailer angles because the lower-level controller takes care of this on its own, so

$$\mathcal{L}_2^{\text{sensory}} = \alpha_2 \log (1 + d_{\text{ft}}/L) \ . \tag{2.2.3}$$

The higher-level motor command portion of the cost is given by

$$\begin{aligned} \mathcal{L}_2^{\text{command}} \ = \ & \beta_2 \left(\cos(\theta_{\text{st}})^2 + \sin(\theta_{\text{st}})^2 - 1 \right)^2 + \gamma_2 \Big((d_{\text{st}} - d_{\min})^2 \, \Theta (d_{\min} - d_{\text{st}}) \\ & + (d_{\text{st}} - d_{\max})^2 \, \Theta (d_{\text{st}} - d_{\max}) \Big) \ . \end{aligned} \tag{2.2.4}$$

The first term in this equation may look strange because the sum of the squares of a cosine and a sine is always 1. However, the optimization procedure does not generate an angle θ_{st} and take its cosine and sine. Instead, it generates values for the cosine and sine directly

without any constraint requiring that these obey the laws of trigonometry. As a result, this constraint needs to be included in the cost function. The distance-dependent terms in equation 2.2.4 penalize sub-target distances that are either too short or too long.

The final term in the higher-level cost function, $\mathcal{L}_2^{\text{obstacle}}$ penalizes truck positions that are too close to an obstacle. We are not concerned with the distance from the truck to every single obstacle; rather, we care primarily if the truck is too near a single obstacle, the closest one. Thus, we choose the smallest distance to an obstacle, $d_{\text{min}}^{\text{obstacle}}(t)$, and impose a Gaussian penalty for proximity to this closest obstacle with standard deviation equal to the disc's radius, σ_{disc}. We also add a smaller, flatter penalty with a larger standard deviation, σ_{areola}, as a warning signal to prevent the truck from wandering nearby the obstacle. The resulting cost function is

$$\mathcal{L}_2^{\text{obstacle}} = \rho_2 \exp\left(-\frac{\left(d_{\text{min}}^{\text{obstacle}}\right)^2}{2\sigma_{\text{disc}}^2}\right) + \nu_2 \exp\left(-\frac{\left(d_{\text{min}}^{\text{obstacle}}\right)^2}{2\sigma_{\text{areola}}^2}\right). \tag{2.2.5}$$

Recall that the obstacles are detected by the sensory grid system show in Figure 4, and thus the distances to obstacles are not directly available. We solve this problem by introducing a network that evaluates the cost function 2.2.5 directly from the sensory grid information $\mathbf{g}(t)$. We call this network the "obstacle critic" because it serves the same role as critic networks in reinforcement learning (Widrow et al., 1973) (Sutton and Barto, 1998): predicting the cost of sensory data, ultimately to train another network. It has 199 grid inputs and one bias and a single output, representing the estimated cost of the sensor reading (Methods). We train this network to predict the obstacle cost from grid data by creating a large number of measurement scenarios and computing the true cost function.

The complete higher-level cost function is the sum of the costs given in equations 2.2.3, 2.2.4, and 2.2.5. The trajectories that minimize the complete higher-level cost function are

goal-seeking and obstacle-avoiding.

The Forward Models

The forward model for level l predicts the future sensory data $\mathbf{s}_l(t + T_l)$ that result from passing the command $\mathbf{m}_l(t)$ to the level below it. We use the convention that the forward model is named after the level that issues the commands and receives the sensory data, not the level that follows those commands and causes the sensory data to change. Thus, the higher-level forward model is actually modeling the lower-level controller, and the lower-level forward model, the truck.

The lower-level forward model, which is a single network, is trained by randomly choosing a set of sensory data, $\mathbf{s}_1(t)$, consisting of the distance and cosine and sine of the angle to a target, and a command $\mathbf{m}_1(t)$, defining a wheel angle, within their allowed ranges. We then simulate the motion of the truck for a time T_1 and determine the sensory data $\mathbf{s}_1(t + T_1)$, indicating t he articulation of the cab with respect to the trailer and where the truck lies in relation to the sub-goal. We continue to gather data for the lower-level forward model by applying another command to the truck and taking a new sensory measurement after the delay. In this way, we generate several command sequences along a single trajectory to gather more data. To create a diversity of training cases for the forward model, we periodically terminate a trajectory and start a new trial from a random initial condition. On the basis of several thousand such measurements, we train the lower-level forward model network to predict the motion and articulation of the truck over the full range of initial conditions and motor commands.

The higher-level forward model is composed of three networks: proprioceptive, goal-related, and obstacle-related (Methods). Each of these networks receives the command $\mathbf{m}_2(t)$. The proprioceptive network also receives the proprioceptive information, [$\cos(\theta_{\text{rel}}(t))$,

$\sin(\theta_{\text{rel}}(t))]$ and predicts $[\cos(\theta_{\text{rel}}(t+T_2)), \sin(\theta_{\text{rel}}(t+T_2))]$. The goal-related forward model receives these proprioceptive data and the goal-related information, $[\log(1+d_{\text{ft}}/L), \cos(\theta_{\text{ft}}), \sin(\theta_{\text{ft}})]$, and predicts the goal-related variables at time $t+T_2$. The obstacle-related forward model is the most complicated. It receives the proprioceptive information and the obstacle grid data $\mathbf{g}(t)$. Unlike the other forward model networks we have described, which make deterministic predictions, the predictions of the goal-related forward model are probabilistic. This is necessary because the grid predictions are underdetermined. For example, at time t the grid input cannot provide any information about obstacles outside its range, but one of these may suddenly appear inside the grid at time $t+T_2$. The goal-related forward model cannot predict such an event with certainty. Therefore, we ask the obstacle-related network to predict the probability that each grid point will be occupied at time $t+T_2$, given a particular grid state and command issued at time t. This is, obviously, a number between 0 and 1, in contrast with the true value of $g_i(t+T_2)$, which would be either 0 or 1. We explain how this is done in the Methods.

When we use forward model networks to predict sensory information along a trajectory, $\mathbf{s}_l(t), \mathbf{s}_l(t+T_l), \mathbf{s}_l(t+2T_l), \ldots$, we simply iterate, using the predicted sensory data time $t+kT_l$ to generate a new prediction at $t+(k+1)T_l$. This allows us to compute the summed cost functions of equation 2.2.1 for both controller levels (Figure 7).

Computing Optimal Commands

Our procedure for generating optimal command sequences is schematized in Figure 7. We begin by initializing states of the environment and truck randomly (within allowed ranges) and to measure $\mathbf{s}_l(t)$. We then choose an initial (or "nominal") sequence of commands $[\mathbf{m}_l(t), \mathbf{m}_l(t+T_l), \ldots, \mathbf{m}_1(t+K_lT_l)]$. For the lower-level controller, we choose $\mathbf{m}_1(t+kT_1) = u(kT_1) = 0$ for all k, indicating that pointing the wheels straight is our first guess for the optimal command. For the higher-level controller, we choose the initial command sequence

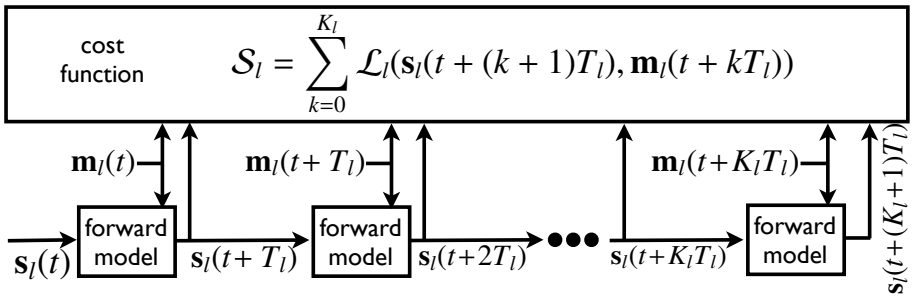

Figure 7. Schematic showing the iterated use of the forward model for computing an optimal command sequence. The system is provided with sensory input at time t through the vector $\mathbf{s}_l(t)$. The forward model is used repeatedly to generate predictions of the sensory vector a times $t + T_l, \ldots, t + (K_l + 1)T_l$. The command sequence $\mathbf{m}_l(t), \mathbf{m}_l(t + T_l), \ldots, \mathbf{m}_1(t + K_lT_l)$ is an input to both the cost-function computation and the forward model, and it is optimized.

to consist of identical commands placing a sub-target 50 length units directly behind the truck. We apply these commands sequentially to calculate an entire truck trajectory.

The command sequences are then optimized by using the Dynamic Optimization algorithm described in the Methods (also known as Pontryagin's Minimum Principle (Stengel, 1994) or Backpropagation-through-Time (LeCun, 1988)). Briefly, we calculate the effect that a small change to each command will have on the total trajectory cost and determine the gradient of the cost function with respect to the parameters defining the commands. This gradient information is used to change the commands according to a variable-metric update accomplished with the limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method in "minFunc" (Schmidt, 2013). L-BFGS uses successive gradient computations to compute an approximation of the inverse Hessian of the cost function, which speeds up optimization considerably compared to steepest descent. We iterate the optimization until performance does not improve (more details are provided in the Methods).

The above procedure generates a single command sequence that is optimal for the initial sensory data $\mathbf{s}_l(t)$. However, we can use the sensory data after the first command has generated its sensory consequences (at time $t + T_l$), to optimize a new trajectory starting from

the new sensory data. This process can obviously be repeated at subsequent times. We periodically terminate the trajectory and begin a new trial to obtain sufficient variety in the initial sensory data. When this process is completed, the pairings of initial sensory data and optimal commands constitute a training set for the controller being trained, which learns to associating each sensory measurement with the appropriate command.

Computing an optimal trajectory of K_1 steps at the lower-level takes on average 1.3 seconds on our computer (averaged over 100 such optimizations). Computing an optimal trajectory of K_2 steps at the higher-level takes on average 12.47 seconds (averaged over 10 such optimizations). Once the lower- and higher-level controllers have been trained, only 0.001 seconds is needed to run the two controllers in series. Clearly, memorization provides a tremendous advantage in speed over online optimization.

Discussion

The training procedure we have described provides a strategy to design a hierarchical control system for problems that can be formulated and solved by division into easier subproblems. Many interesting tasks have this structure, so our strategy should be quite generally applicable. The construction of the hierarchy is recursive or self-similar. The lowest level controls the plant itself, and higher levels control the levels below them.

Apart from navigation problems, the hierarchical scheme and training procedure using forward models should be useful for tackling problems that require simultaneous application of sensorimotor and cognitive skills. Consider, for example, a robotic gripper moving blocks. It would be quite a challenge to construct a unitary network that directs the gripper to pick up, move and drop blocks and that decides how to arrange them into a prescribed pattern at the same time. It is easier to separate the problem into a manual coordination task and a puzzle-solving task, and a natural vehicle for this separation is the network ar-

chitecture itself. Surprisingly, despite the intuitiveness of this idea, it has received little attention. The closest antecedents of our work on hierarchical network control are to be found in the work of Kawato and colleagues, who designed a low-level network to solve an inverse kinematic problem and a higher-level network to solve an inverse dynamics problem (Kawato et al., 1987). In this work, however, both problems were still essentially motor-level problems. In robotics, it has now become conventional to build hierarchies in which the system levels appropriate different task levels. Dominant paradigms include Brooks's subsumption architecture (Brooks, 1986) and three-layer architectures, so dubbed by Gat (Gat, 1998), and used to very impressive effect in the self-driving car Stanley, winner of the DARPA Grand Challenge (Thrun et al., 2006). It is not conventional, however, in robotics for higher levels to possess forward models the lower levels.

The idea of controlling a physical system by building a model of it is quite old, entering the connectionist literature with Nguyen and Widrow (Nguyen and Widrow, 1989) and Jordan and Rumelhart (Jordan and Rumelhart, 1992), but existing in a prior incarnation as "model-reference adaptive control." To the best of our knowledge, the idea of using a network to model the feedback from another network and then to use that model to control the modeled network is novel. We expect it to have ramifications that exceed the traditional framework of motor control. Forward models could be constructed either by motor babbling or by a more intelligent experimentation procedure. Once the controllers have been trained, they exhibit *automaticity*, an ability to generate answers without extensive computation. Automating complex computations by "caching" has been proposed before (Kavukcuoglu et al., 2008) (Dayan, 2009) in different contexts, but using cached circuits as lower-level substrates for higher-level control circuits has not.

Although we have trained the forward-model and controller networks sequentially, we do not preclude the attractive possibility of training them at the same time, maybe even at

multiple levels of the hierarchy simultaneously. To do so, we expect it would be crucial to account for errors in the forward models; we could generate cautious commands by penalizing those commands that a forward model is unlikely to predict accurately, or we could also generate commands that attempt to probe the response properties of the level below to improve the model.

Although our networks are certainly not models of biological neural systems, we cannot resist drawing a connection to physiology. A long-standing hypothesis is that the cerebellum implements a forward model that predicts the delayed sensory consequence of a motor command (Miall et al., 1993) (Shadmehr et al., 2010). Intriguingly, a recent study suggests that neurons in Clarke's column in the spinal cord simultaneously receive input from cortical command centers and from proprioceptive sensory neurons, and the authors speculate that this joining of sensory and motor inputs could function as a predictor, akin to a forward model (Hantman and Jessell, 2010). Experimentalists have therefore already proposed multiple loci for forward models, functioning at different levels of the motor system. This connection suggests that motor behavior may be produced by hierarchies of controllers that are trained by forward models at each level of the hierarchy.

This study presents a hard result, a hierarchical neural network, but it also bolsters a soft view. To create intelligent behaviors from neuron-like components, we need to embed those neurons in modules that perform specific functions and operate to a large extent independently. These modules need to have protocols for interfacing one to another, protocols which effectively hide the complexity of the full computation from the constituents. In experimental neuroscience, studying how such modules interact may require us to move beyond single-area recordings and understand the causal interactions between connected regions, in part to identify the goals of the computations performed within any one region.

Methods

A. Parameters

truck & obstacles

Parameter	Value
r (speed)	0.2
l_{cab} (length)	6
w_{cab} (width)	6
l_{trailer}	14
w_{trailer}	6
σ_{disc} (radius)	20

lower-level cost function

Parameter	Value
α_1	1
β_1	0.5
γ_1	0.5
ζ_1	0.5
θ_{max}	$\frac{\pi}{2} - \frac{\pi}{6}$

higher-level cost function

Parameter	Value
α_2	1
β_2	100
γ_2	100
d_{min}	0.1
d_{max}	0.9
σ_{disc}	20
σ_{areola}	30
ρ_2	2
ν_2	0.5

lower-level training

Parameter	Value
T_1	6
K_1^{model}	15
$K_1^{\text{optimization}}$	15
K_1^{total}	30
$N_{\text{c},1}$	2×10^4
$N_{\text{fm},1}$	5×10^4

higher-level training

Parameter	Value
T_2	$12 \cdot T_1$
K_2^{model}	1
$K_2^{\text{optimization}}$	10
K_2^{total}	10
$N_{\text{c},1}$	5×10^5
$N_{\text{fm},2}$	2×10^6

Table 1. Parameters for (from left to right) the truck and obstacles, the lower- and higher- level cost function, and the lower- and higher-level training. $N_{\text{c},l}$ denotes the number of example patterns used to train the controller and $N_{\text{fm},l}$ the number for the forward model. K_l^{total} refers to the number of steps taken when optimizing trajectories before restarting the truck in a new environment.

B. Network Structure and Training

The lower-level makes use of radial-basis function (RBF) neural networks. RBF networks possess a strong bias toward generating smoothly-varying outputs as a function of their inputs and can train very quickly on low-dimensional input data, two qualities that are useful at the lowest level of the hierarchy. The functional targets for the higher-level networks were more complicated and higher-dimensional, so we had to develop "deep" (or many-layered) networks to approximate them accurately.

The architectures of the networks are:

Network	Dimensions and Activation Functions
Lower-Level Forward	$6 \times$ [G]$150 \times$ [L]5
Lower-Level Controller	$5 \times$ [G]$100 \times$ [L]1
Higher-Level Forward (Proprio.)	$5 \times$ [T]$40 \times$ [T]$20 \times$ [T]$20 \times$ [L]2
Higher-Level Forward (Goal)	$8 \times$ [T]$40 \times$ [T]$20 \times$ [T]$20 \times$ [L]3
Higher-Level Forward (Obstacle)	$205 \times$ [T]$300 \times$ [T]$200 \times$ [T]$200 \times$ [T]$300 \times$ [Si]199
Higher-Level Critic (Obstacle)	$199 \times$ [T]$300 \times$ [T]$100 \times$ [T]$100 \times$ [So]1
Higher-Level Controller	$205 \times$ [So]$30 \times$ [So]$20 \times$ [So]$30 \times$ [L]3

The bracketed letters indicate the activation functions used for the units. [G] is a normalized Gaussian radial basis function for input x,

$$\frac{\exp\left(-\|\mathbf{x}-\mu_i\|^2/(2\sigma_i^2)\right)}{\sum_j \exp\left(-\|\mathbf{x}-\mu_j\|^2/(2\sigma_j^2)\right)},$$

with basis function centers μ_i and standard deviations σ_i. The radial basis function centers were chosen by randomly selecting exemplars from the input data. They were not further adapted in training. The RBF standard deviations were initialized to scale linearly with the number of input dimensions. To avoid division-by-zero, we actually computed using the inverse standard deviations $1/\sigma_i$.

All the other multiplication signs in Table 1 imply matrix multiplication followed by an activation function. [L] is a linear activation function, [T] is $\tanh(x)$, [Si] is a logistic sigmoid $1/(1+\exp(-x))$, and [So] is a "soft-rectification" function $\log(1+\exp(x))$. We chose these activation functions using a mixture of prior knowledge and experimentation. The soft-rectification in the obstacle critic imposed the constraint that costs are positive. The logistic function in the obstacle grid forward model bounds the outputs between 0 and 1 and allows us to interpret them as the probabilities that the grid points will be occupied.

The soft-rectification in the higher-level controller alleviated over-fitting.

A weight matrix from a layer of size M to another layer was initialized to have Gaussian entries of mean 0 and standard deviation $1/\sqrt{M}$. All multi-layer networks included a single additional bias unit in their inputs.

We chose to interpret an obstacle grid model output as the probability that the corresponding grid element would be occupied after movement. We consequently used the "cross-entropy" cost function to train this network. We can derive this cost function from a maximum-likelihood argument. Suppose we have a data set of patterns $(\mathbf{x}_k, \mathbf{y}_k)_{k=1}^{N_{\text{patterns}}}$ and a neural network with M outputs whose i-th output unit as a function of the k-th input vector given by $z_i(\mathbf{x}_k)$ represents the binomial probability that $y_i(\mathbf{x}_k) = 1$. The likelihood of the data is binomial, so the log-likelihood is

$$\text{Log-Likelihood} = \sum_{k=1}^{N_{\text{patterns}}} \sum_{i=1}^{M} y_i(\mathbf{x}_k) \log z_i(\mathbf{x}_k) + (1 - y_i(\mathbf{x}_k)) \log(1 - z_i(\mathbf{x}_k)) \,.$$

When negated, this gives the cross-entropy cost function.

Optimization of the network parameters was accomplished using batch training with the quasi-Newton optimization method L-BFGS in "minFunc" (Schmidt, 2013). To train the lower-level forward model or the lower-level controller, at the beginning of each trial a random angle θ_{st} was drawn along with a random distance d_{st} in the range between 0 and 500. The trailer angle θ_{trailer} was similarly drawn uniformly. The cab angle was initialized to be within $\pm(\pi/2 - \pi/64)$ radians of the trailer angle.

The higher-level proprioceptive and goal-related models were trained to predict not the values of their targets but the difference between the values of their targets before and after movement. This reduced training time because the interesting predictions of many forward models are the deviations from the identity.

C. Equations for the Truck

The truck is a kinematic model of a cab and trailer (Figure 8), first defined by Nguyen and Widrow (Nguyen and Widrow, 1989). The cab is connected to the trailer by a rigid linkage.

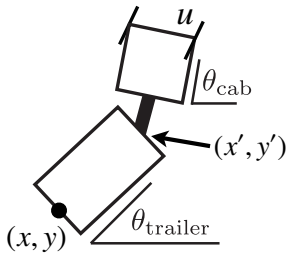

Figure 8. Variables describing the truck. The position of the center of the back of the trailer is given by (x, y), and the center of the front of the trailer is at (x', y'). The angle of the cab with respect to the x-axis is $\theta_{\text{cab}}(t)$ and for the trailer, $\theta_{\text{trailer}}(t)$. The angle by which the front wheels deviate from straight ahead is u, and the relative angle between the cab and trailer is $\theta_{\text{rel}} = \theta_{\text{cab}} - \theta_{\text{trailer}}$.

The wheels are connected to the front of the cab and translate backward by distance r in one time step. Note that the wheels drive the cab the same way that the linkage drives the trailer, so we can solve for the motion of the cab and trailer in a similar way.

We begin by decomposing the motion of the front of the cab caused by the wheels into a component orthogonal to the front of the cab, defined as $A = r\cos(u(t))$, and a component parallel to the front of the cab, $C = r\sin(u(t))$. Only the orthogonal component, A, gets transferred through the linkage to the trailer. Performing a similar decomposition of the motion of the front of the trailer, we find an orthogonal component $B = A\cos(\theta_{\text{cab}}(t) - \theta_{\text{trailer}}(t))$ and a parallel component $D = A\sin(\theta_{\text{cab}}(t) - \theta_{\text{trailer}}(t))$. In one time step, the center of the front of the trailer (Figure 8) therefore moves by

$$\begin{aligned} x'(t+1) &= x'(t) - B\cos(\theta_{\text{trailer}}(t)) + D\sin(\theta_{\text{trailer}}(t)) \\ y'(t+1) &= y'(t) - B\sin(\theta_{\text{trailer}}(t)) - D\cos(\theta_{\text{trailer}}(t)) \,. \end{aligned}$$

The back of the trailer is constrained to move straight backward, so

$$\begin{aligned} x(t+1) &= x(t) - B\cos(\theta_{\text{trailer}}(t)) \\ y(t+1) &= y(t) - B\sin(\theta_{\text{trailer}}(t))\,. \end{aligned}$$

If the length of the trailer is L_{trailer}, $x'(t) = x(t) + L_{\text{trailer}}\cos(\theta_{\text{trailer}}(t))$ and $y'(t) = y(t) + L_{\text{trailer}}\sin(\theta_{\text{trailer}}(t))$. The tangent of the angle of the trailer is equal to $(y' - y)/(x' - x)$, so

$$\tan(\theta_{\text{trailer}}(t+1)) = \frac{L_{\text{trailer}}\sin(\theta_{\text{trailer}}(t)) - D\cos(\theta_{\text{trailer}}(t))}{L_{\text{trailer}}\cos(\theta_{\text{trailer}}(t)) + D\sin(\theta_{\text{trailer}}(t))}.$$

An identical argument applied to the cab yields

$$\tan(\theta_{\text{cab}}(t+1)) = \frac{L_{\text{cab}}\sin(\theta_{\text{cab}}(t)) - C\cos(\theta_{\text{cab}}(t))}{L_{\text{cab}}\cos(\theta_{\text{cab}}(t)) + C\sin(\theta_{\text{cab}}(t))}\,.$$

Consolidating all of the equations, we have

$$\begin{aligned} A &= r\cos(u(t)) \\ B &= A\cos(\theta_{\text{cab}}(t) - \theta_{\text{trailer}}(t)) \\ C &= r\sin(u(t)) \\ D &= A\sin(\theta_{\text{cab}}(t) - \theta_{\text{trailer}}(t)) \\ x(t+1) &= x(t) - B\cos(\theta_{\text{trailer}}(t)) \\ y(t+1) &= y(t) - B\sin(\theta_{\text{trailer}}(t)) \\ \theta_{\text{cab}}(t+1) &= \tan^{-1}\left(\frac{L_{\text{cab}}\sin(\theta_{\text{cab}}(t)) - C\cos(\theta_{\text{cab}}(t))}{L_{\text{cab}}\cos(\theta_{\text{cab}}(t)) + C\sin(\theta_{\text{cab}}(t))}\right) \\ \theta_{\text{trailer}}(t+1) &= \tan^{-1}\left(\frac{L_{\text{trailer}}\sin(\theta_{\text{trailer}}(t)) - D\cos(\theta_{\text{trailer}}(t))}{L_{\text{trailer}}\cos(\theta_{\text{trailer}}(t)) + D\sin(\theta_{\text{trailer}}(t))}\right). \end{aligned}$$

It is worth mentioning that $(x, y, \theta_{\text{cab}}, \theta_{\text{trailer}})$ is a four-dimensional state vector with a one-dimensional control variable, u. Control theorists call problems in which the control vector is lower-dimensional than the state vector "under-actuated"; such problems are typically more difficult than "fully-actuated" problems because it may take more than one step to modify a given state variable, and it may be impossible to drive the system to an arbitrary point in the state space (a concept known as "controllability"). The truck is also a nonlinear system and an unstable one because setting $u = 0$ amplifies any angular deviation of the cab from the trailer.

D. Minimizing the Cost Functions

In this section we describe how to minimize the cost functionals with respect to the command parameters. We are minimizing a cost functional of the form

$$\mathcal{S} = \sum_{k=0}^{K} \mathcal{L}(\mathbf{s}(k+1), \mathbf{m}(k)) \tag{2.4.6}$$

as in equation 2.2.1, but to streamline the notation, we have dropped the time t and the temporal scale factor T that appear in equation 2.2.1. This means that we have shifted the time variable to the starting at time t, and measure time in units of T. The original equations can be recovered by shifting and scaling back. We have also dropped the subscripts l because the same procedure is applied at each level.

The sensory vector $\mathbf{s}$ is estimated by a forward model, and we denote the output of the forward model by $\mathbf{F}(\mathbf{s}, \mathbf{m})$, so that $\mathbf{s}(k+1)$ is estimated as $\mathbf{F}(\mathbf{s}(k), \mathbf{m}(k))$. To implement this constraint, we introduce Lagrange multipliers for every component of $\mathbf{m}$ and at every moment in time and minimize

$$\mathcal{S}_{\text{constrained}} = \sum_{k=0}^{K} \mathcal{L}(\mathbf{s}(k+1), \mathbf{m}(k)) + \lambda(k)^{\top}\Big(\mathbf{F}(\mathbf{s}(k), \mathbf{m}(k)) - \mathbf{s}(k+1)\Big). \tag{2.4.7}$$

For convenience, we define $\lambda(K+1)=0$.

The gradient of this function with respect to the sensory vector is

$$\frac{\delta \mathcal{S}_{\text{constrained}}}{\delta \mathbf{s}(k)} = \mathcal{L}_{\mathbf{s}}(\mathbf{s}(k), \mathbf{m}(k-1)) + \mathbf{F}_{\mathbf{s}}(\mathbf{s}(k), \mathbf{m}(k))^{\top} \lambda(k) - \lambda(k-1)\,,$$

where the subscript $\mathbf{s}$ indicates a derivative with respect to that variable. At a minimum of the cost function we find the backward equation for the Lagrange multipliers

$$\lambda(k-1) = \mathcal{L}_{\mathbf{s}}(\mathbf{s}(k), \mathbf{m}(k-1)) + \mathbf{F}_{\mathbf{s}}(\mathbf{s}(k), \mathbf{m}(k))^{\top} \lambda(k)\,. \tag{2.4.8}$$

It is easy to find an extremum of the cost functional with respect to the sensory variables and the Lagrange multipliers. It is more difficult to minimize with respect to the command variables, where the relevant gradient is

$$\frac{\delta \mathcal{S}_{\text{constrained}}}{\delta \mathbf{m}(k)} = \mathcal{L}_{\mathbf{m}}(\mathbf{s}(k+1), \mathbf{m}(k)) + \mathbf{F}_{\mathbf{m}}(\mathbf{s}(k), \mathbf{m}(k))^{\top} \lambda(k)\,.$$

This is done using the following algorithm:

Dynamic Optimization

Input: Initial state $\mathbf{s}(0)$ and "nominal" control tape $\{\mathbf{m}(k)\}_{k=0}^{K}$
  repeat
    for k := 0 to K **do**
      $\mathbf{s}(k+1) := \mathbf{F}(\mathbf{s}(k), \mathbf{m}(k))$;
    end for
    $\lambda(K+1) := 0$;
    for k := K+1 down to 1 **do**
      $\lambda(k-1) := \mathcal{L}_{\mathbf{s}}(\mathbf{s}(k), \mathbf{m}(k-1)) + \mathbf{F}_{\mathbf{s}}(\mathbf{s}(k), \mathbf{m}(k))^{\top} \lambda(k)$;
      $\frac{\delta \mathcal{S}_{\text{constrained}}}{\delta \mathbf{m}(k-1)} := \mathcal{L}_{\mathbf{m}}(\mathbf{s}(k), \mathbf{m}(k-1)) + \mathbf{F}_{\mathbf{m}}(\mathbf{s}(k-1), \mathbf{m}(k-1))^{\top} \lambda(k)$;
      Update $\mathbf{m}(k-1)$ using a gradient-based method (steepest descent, L-BFGS, etc.) provided with $\frac{\delta \mathcal{S}_{\text{constrained}}}{\delta \mathbf{m}(k-1)}$.
    end for
  until convergence

Chapter 3

Sensitivity Models for Local, Error-Driven Control

Life can only be understood backward, but it must be lived forward. - Søren Kierkegaard

Sensitivity Models for Local, Error-Driven Control

Greg Wayne

Department of Neuroscience

Columbia University College of Physicians and Surgeons

New York, NY 10032-2695 USA

Abstract

We try to investigate the intuitive notions of "biological plausibility" that are commonly used to reject learning mechanisms based on optimal control formalisms. These notions capture the beliefs that biological systems do not use external, non-neural memory structures to keep record of planning and optimization computations and that they cannot calculate the derivatives necessary to implement gradient descent. We describe one possible way to avoid these implausibilities by first learning a forward model of the plant and either subsequently or simultaneously learning a "sensitivity model" of the forward model (another neural network that calculates the Jacobian matrix of the forward model). We integrate the forward model and the sensitivity model with a forward-in-time computation, known as "forward accumulation," to optimize controller structures for two model optimal control problems: the torque-limited pendulum swing-up and the cart-pole swing-up. Our method is model- and gradient-based, so, once the models have been learned, it can compute control responses extremely efficiently.

3.1 Introduction

In this chapter, we play the role of the scientific realist. We consider the computations involved in optimal control and ask whether neural networks, respecting certain architectural desiderata, can account for those computations. We want to examine what the often unspoken, consensus view of "biological plausibility" actually dictates and see if we can construct an optimal controller based on a restriction to elements admissible under that view. Consequently, this chapter's results are in part sociological. As opinions about biological plausibility are informed by more experiments, the arguments in this chapter may become outmoded – or more compelling. Our ulterior motive is that we would ultimately like to create spiking neural networks that can perform optimal control calculations. This chapter's results serve as a useful milestone on that path.

In the last chapter, we used the Dynamic Optimization (DO) algorithm or Backpropagation-through-Time (BPTT) to optimize sequences of control commands to apply to our dynamical systems. This algorithm is frequently described as biologically implausible. What, however, does this mean?

Consider a dynamical system defined by the equation $\dot{\mathbf{x}} = \mathbf{F}_{\text{system}}(\mathbf{x}, \boldsymbol{\Theta})$, where $\boldsymbol{\Theta}$ stands either for a set of system parameters or control inputs $\mathbf{u}(t)$. To apply DO, a precise record of the states the system passes through must be maintained. We must store a list (or a stack) of the states $\mathbf{x}(t), \mathbf{x}(t + dt), \mathbf{x}(t + 2dt), \ldots$, and so on. This list of states is typically not a part of the model *per se* but a part of a conventional computer memory. When the system is a recurrent neural network, the memory storage required for the application of DO completely dwarfs the memory capacity of the network itself.[1] In some cases (not

[1]This is perhaps not true if you count the memory capacity latent in synapses. Of course, this comes down to a scaling argument. DO requires resources that scale with time, and if long enough time intervals are considered, the spatial resources, even synaptic ones, would be exhausted.

in all), this leads to obvious absurdities: to train a network by DO to store and maintain information as recirculating activity for a long time, a conventional use for a recurrent network, one needs a pre-existent computer memory that can already solve the problem without training. (Of course, one may not have on hand the algorithm or process that has synthesized the training data, so a recurrent neural network's generalization capacity may still prove useful.)

DO computes the derivatives of cost functions with respect to the system parameters or the system inputs. Regardless of the exact problem, one always needs to calculate the Jacobian matrices $\partial \mathbf{F}_{\text{system}}(\mathbf{x}(t), \boldsymbol{\Theta}) / \partial (\mathbf{x}(t), \boldsymbol{\Theta})$ at every moment in time. Again, this calculation is not carried out by the units in a neural network model.

Even more conspicuously, DO divides time into two phases. In the first phase, the system equations are run forward for some predetermined interval of time. The system is then stopped, and in the second phase the list of states is traversed in the backward direction $\mathbf{x}(t + Kdt), \mathbf{x}(t + (K-1)dt), \mathbf{x}(t + (K-2)dt), \ldots$ from the present moment back into the past. At least within the motor system, we cannot recall a time when anyone has ever claimed that a group of neurons retraces a sequence of activity backward in time.[2][3]

To summarize, there seem to be three primary reasons why DO is not considered biologically plausible. They are:

1. A separate, non-neural memory structure, whose capacity scales with the length of

[2] In the study of more cognitive tasks, psychologists have examined the ability of human subjects to search for an optimal plan in games like chess. An interesting result is that people have difficulty recounting the steps backward from a planned point B, a final state, to point A, the initial state (Newell, 1994), in much the same way as it is difficult to recite the alphabet backward. Consequently, people seem to plan by tracing paths forward from initial states and dropping off the paths back to the initial states when the current one is deemed problematic. This stands in contrast to common techniques in artificial intelligence where a game-playing AI might perform a "depth-first" search. In chess and go, it is a mark of skill to be able to trace one's moves backward from the end game to the beginning. It is a bit of a stretch to argue that the same computational difficulty should be inherent in motor domains, but the result does give one pause.

[3] We omit discussion of hippocampal replay, which is probably not important for motor learning.

the trajectory to be optimized, is used to record the historical activity of the plant.

2. Derivatives of the plant need to be computed, but they are not themselves computed by a network.

3. Using the memory structure and the derivatives, error signals are calculated by a process that steps backward through the historical record of activity.

We therefore ask whether it is possible to construct an efficient neural network control system that avoids (1), (2), and (3). We require that the error signals that modify the functioning of our control system be represented within one or several networks.

This chapter now begins with a discussion of the computational requirements and tradeoffs in optimal control. We then discuss how the terms in the control problem can be approximated by neural networks, gradually building up a control system that avoids the three implausibilities we have listed. Finally, we apply our methods to two well-known optimal control problems, the torque-limited pendulum swing-up and the cart-pole swing-up.

3.2 Discounted Optimal Control

We consider first the case of a receding-horizon, discounted control task. At every time t, we look forward into the future until time $t + N\tau$ and try to optimize the time integral of a cost function

$$\mathcal{S} = \int_t^{t+N\tau} dt' \frac{1}{\tau} \exp\Big[-(t' - t)/\tau \Big] \mathcal{L}(\mathbf{x}(t'), \mathbf{u}(t')). \tag{3.2.1}$$

τ is the discounting time scale. $N\tau$ is our optimization horizon. One useful reason to discount is that the cost-integral is well-defined even as $N \to \infty$ as long as $\mathcal{L}$ is bounded. Also, as $\tau \to \infty$ but $N \sim K/\tau$, we recover undiscounted receding-horizon control. Because the Lagrangian $\mathcal{L}(\mathbf{x}, \mathbf{u})$ depends on time only through state and control variables, the cost-

integral is time-translation invariant or Markovian. In the previous chapter, we optimized cost-integrals by solving for a list of controls $\mathbf{u}(t)$. In this chapter, we take a different perspective. We construct a network feedback controller (also known as a "policy") that produces the control as a function of the state $\mathbf{u}(\mathbf{x};\boldsymbol{\Theta})$ with parameters $\boldsymbol{\Theta}$. Our targets to optimize are now the parameters of this controller.

3.3 Methods of Computing Gradients

Perturbation

If we include the policy within the cost-integral, we have the following equation:

$$\mathcal{S} = \int_t^{t+N\tau} dt' \frac{1}{\tau} \exp\Big[-(t'-t)/\tau \Big] \mathcal{L}\Big(\mathbf{x}(t'), \mathbf{u}(\mathbf{x}(t');\boldsymbol{\Theta})\Big). \tag{3.3.1}$$

To optimize the policy we need to compute $\frac{\partial \mathcal{S}}{\partial \boldsymbol{\Theta}}$ by some means.[4] A simple method to compute $\partial\mathcal{S}/\partial\boldsymbol{\Theta}$ is to run twice as many simulations as there are parameters $|\boldsymbol{\Theta}|$. For every parameter, we make a small increment $\Theta_i \to \Theta_i + \Delta$ and a decrement $\Theta_i \to \Theta_i - \Delta$ and simulate once with each change to the parameter. The difference of the cost-integral between these two experiments gives us the required partial derivative

$$\frac{\partial \mathcal{S}}{\partial \Theta_i} \approx \Big[\mathcal{S}(\boldsymbol{\Theta}|\Theta_i + \Delta) - \mathcal{S}(\boldsymbol{\Theta}|\Theta_i - \Delta)\Big]/2\Delta,$$

where the notation $\mathcal{S}(\boldsymbol{\Theta}|\Theta_i \pm \Delta)$ implies that only the indexed parameter is changed. The obvious downside to this "finite-difference" or "serial perturbation" procedure is that an

[4] It is also possible to optimize the policy by methods that do not rely on gradient computation, namely, by choosing several different random parameter settings and keeping the best one or by mixing solutions generated by random selection as in genetic algorithms, but these methods are typically less efficient than methods that rely on gradients. That said, we still have some lingering interest in such methods for some problems. Let's stay on the lookout!

extremely large number of simulations must be executed from identical initial conditions. Such an algorithm is simply not practical to execute on a real system, and it is only mildly practical if the equations can be executed quickly on a model system. The benefit of this approach from the perspective of the mechanisms involved is that the only signal to be communicated back to the parameters is a global one, $\mathcal{S}$. Finite-differencing is thus closely related to reinforcement learning algorithms. It can be performed on either the system itself, where each run is performed by restarting the system after a short trial period, or it can be performed on a forward model of the system; in this case, one has more liberty and can, for example, restart the trials from identical states very easily. If one has a forward model, one can optimize before committing to action on the real system, a notion known as "planning" in the control theory literature.

The major failing of the serial perturbation procedure is that it completely ignores the causal structure of the problem. In particular, the parameters only affect the cost through their effect first on the control output and then the effect of the control output on the state. This effect must be relatively stable if the learning problem is solvable at all. "Backward" and "forward accumulation" are two methods that take into account the causal structure of the problem by making use of models of the system and can thereby significantly reduce the number of trials required to calculate gradients.

Backward Accumulation

Backward accumulation is another term for backpropagation. We prove in the Appendix that for equation 3.3.1, the gradient of the cost-integral with respect to the parameters can

be computed from the system of equations:[5]

$$
\begin{aligned}
\mathbf{p}(t + N\tau) &= 0, \\
\dot{\mathbf{p}} &= -\delta\mathcal{L}/\delta\mathbf{x}^\top - \delta\mathbf{u}/\delta\mathbf{x}^\top \cdot \delta\mathcal{L}/\delta\mathbf{u}^\top \\
&\quad -\left(\delta\mathbf{F}/\delta\mathbf{x}^\top + \delta\mathbf{u}/\delta\mathbf{x}^\top \cdot \delta\mathbf{F}/\delta\mathbf{u}^\top\right) \cdot \mathbf{p} + \frac{1}{\tau}\mathbf{p}, \\
\delta\mathcal{S}/\delta\mathbf{u}(t') &= \frac{1}{\tau}\exp\Big[-(t' - t)/\tau\Big]\Big(\delta\mathcal{L}/\delta\mathbf{u} + \mathbf{p}(t')^\top \cdot \delta\mathbf{F}/\delta\mathbf{u}\Big), \\
\partial\mathcal{S}/\partial\boldsymbol{\Theta} &= \int_t^{t+N\tau} dt' \delta\mathcal{S}/\delta\mathbf{u}(t') \cdot \partial\mathbf{u}(t')/\partial\boldsymbol{\Theta}.
\end{aligned}
$$

$\mathbf{p}$, the "costate," is integrated backward from its boundary condition at time $t + N\tau$, and the gradient with respect to the policy parameters can be computed from it. The computational advantage of this procedure is that a model $\mathbf{F}$ of the system need be run forward once from time t to $t + N\tau$ and then the equations used to compute $\mathbf{p}$ are run backward once. This gives us the entire gradient. If the system is a real physical system, the model is distinct from the system.

In backward accumulation, the quantity that is transported from one place and time to another is the costate. In a neural system, we would mean this literally: the terms of the costate would have to be transmitted neurally through the different circuits. In his review of optimal control theory's contribution to understanding motor performance, Todorov has written (Todorov, 2004): "[I]magine that the costate vector is encoded by some population of neurons – which would not be surprising given its fundamental role in the computation of optimal controls. . . ." In the neural network literature, the transportation of the costate is called "backpropagation of error" for a reason. As we prove in the Appendix, the costate

[5] We use the following conventions: 1) a derivative with respect to a time-varying variable's value at a specific instant is marked with a "δ"; a partial derivative with respect to a constant parameter is marked with a "∂"; a total derivative with respect to a constant parameter that includes all paths by which the parameter influences the dependent variable is marked with a "d." 2) typical vectors are column vectors, while gradients are row vectors.

represents the change to the cost-integral $\delta\mathcal{S}$ that is caused by a change to the state $\delta\mathbf{x}(t)$ at time t all other things equal.

Forward Accumulation

Surprisingly, there is another way to optimize the cost-integral with respect to the controller parameters. In this approach, the quantity that is transported from one place and time to another is not an error signal at all; it is the sensitivity of each state variable to each of the parameters, $dx_i(t')/d\Theta_j$. This sensitivity can be calculated forward-in-time alongside the integration of the model equations. Define the matrix $G_{ij}(t') = dx_i(t')/d\Theta_j$ and the vector $\mathbf{q} = \partial\mathcal{S}/\partial\mathbf{\Theta}$. At time t, these variables are initialized identically to 0. We prove in the Appendix that the following system of equations calculates $\mathbf{q}$:

$$\dot{G} = \delta\mathbf{F}/\delta\mathbf{x} \cdot G + \delta\mathbf{F}/\delta\mathbf{u} \cdot \Big(\partial\mathbf{u}/\partial\mathbf{\Theta} + \delta\mathbf{u}/\delta\mathbf{x} \cdot G\Big) \tag{3.3.2}$$

$$\dot{\mathbf{q}} = \frac{1}{\tau}\exp\Big[-(t'-t)/\tau\Big]\Big(\delta\mathcal{L}/\delta\mathbf{u} \cdot \partial\mathbf{u}/\partial\mathbf{\Theta} + \delta\mathcal{L}/\delta\mathbf{x} \cdot G\Big). \tag{3.3.3}$$

To our knowledge, this method of optimizing dynamical systems was first proposed by McBride and Narendra (McBride and Narendra, 1965). It has been rediscovered numerous times, though (Williams and Zipser, 1989) (Kolter, 2010) (Atkeson, 2012). Because the model dynamics run forward-in-time and require no external memory resource beyond the matrix G and vector $\mathbf{q}$, we find forward accumulation to be more convenient for approximation in the form of a system of neural networks.

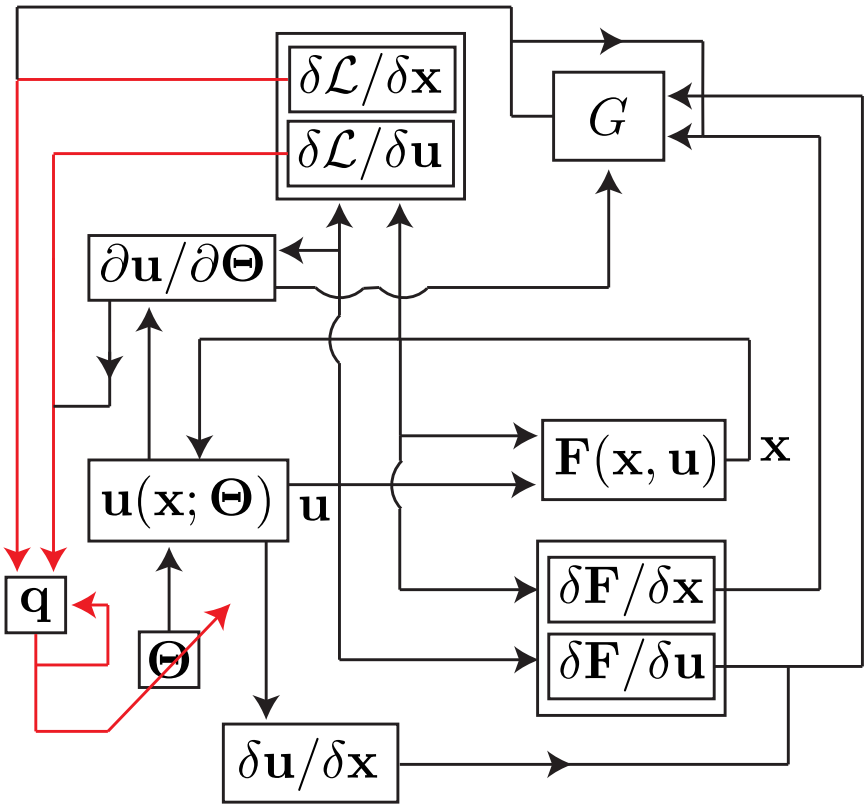

Figure 3.1: Flow Diagram of the Forward Accumulation Computation.

The boxes are functions. Lines entering a box comprise the arguments of the function. The red paths convey information about the cost function; these are error paths. A merging of lines represents multiplication. A box with an outgoing line that loops back to itself represents a differential equation. The crescent loops indicate that two paths are non-interacting. A red arrow crossing another object represents error-driven parameter modificiation. As the forward model $\mathbf{F}$ is integrated, terms involving the sensitivity matrices $\delta\mathbf{F}/\delta\mathbf{x}$ and $\delta\mathbf{F}/\delta\mathbf{u}$ are integrated to form G, the sensitivity of the state variables to the parameters. G is combined with the cost function derivatives $\delta\mathcal{L}/\delta\mathbf{x}$ and $\delta\mathcal{L}/\delta\mathbf{u}$ and integrated to determine $\mathbf{q}$, which modifies $\boldsymbol{\Theta}$ via a gradient-based procedure.

3.3.1 Planning versus Acting

Very importantly, the optimization of equation 3.3.1 by forward or backward accumulation cannot be performed on the actual system since we do not know the Jacobian matrices

of the system. It can only be performed on a model of that system. This has significant consequences.

Suppose we attempt to execute a trial of the control policy on the real system. Unfortunately, the system equations $\dot{\mathbf{x}} = \mathbf{F}_{\text{system}}(\mathbf{x}, \mathbf{u})$ may be inaccurately approximated by our model $\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, \mathbf{u})$. Both forward and backward accumulation rely on estimating $\delta\mathbf{F}_{\text{system}}/\delta\mathbf{x}$ and $\delta\mathbf{F}_{\text{system}}/\delta\mathbf{u}$, which we approximate with our model equations $\delta\mathbf{F}/\delta\mathbf{x}$ and $\delta\mathbf{F}/\delta\mathbf{u}$. There is nothing to be done but to make sure the model is a good one.

Since all optimization is done on a model of the system, we lose nothing, but gain a great deal, by optimizing using the model before acting. If we already possess a system model, we can avoid the time and effort required to work with the real system during optimization computations. Planning computations, in which optimization for the interval of time between t and $t+N\tau$ occurs before or at time t, become attractive. To perform these computations, at a minimum, we need to integrate $\dot{\mathbf{x}}$ from time t to $t+N\tau$ using the forward model several times as we change the parameters of the controller. Planning is especially practical if the computations can be executed quickly enough not to hinder the actual operation of the system. The optimization can be significantly expedited if we need not perform it *de novo*. If we have performed a partial optimization in the past, the current optimization will take less time.

It is also possible to optimize the equations without planning. We can integrate G and $\mathbf{q}$ using our models $\delta\mathbf{F}/\delta\mathbf{x}$, $\delta\mathbf{F}/\delta\mathbf{u}$, $\delta\mathbf{u}/\delta\mathbf{x}$, ... as we interact with the actual system $\mathbf{F}_{\text{system}}$. As we accumulate the gradent, we can make slow changes to $\boldsymbol{\Theta}$ based on the current value of $\mathbf{q}$. However, this method requires a large number of trials on the actual system, and, for the sake of the present study, the efficiency of our network compared to reinforcement learning methods is more apparent if we allow ourselves the freedom to optimize by planning.

3.3.2 Models

Forward Model

To perform model-based planning, we first need the forward model, **F**. In the present study, we construct the forward model as a feedforward neural network. To learn the model, we merely actuate the system with random controls **u** and use the difference between the system state and the model prediction as an error signal.

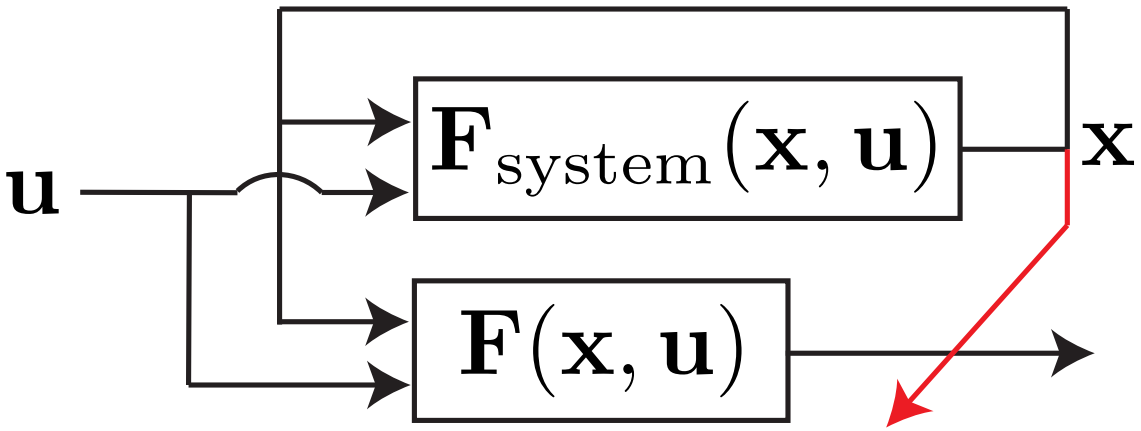

Figure 3.2: Learning a Forward Model.

We provide random control inputs to the system and the forward model. The difference between the output of the system and the forward model is used as an error signal for the model. The crossed arrowheads indicate that the output of the system trains the forward model. The error signal that trains the model is completely unrelated to the error signals used in the control problem.

3.3.3 Sensitivity Models

System Sensitivities

Equation 3.3.2 demands that we calculate the sensitivities $\delta\mathbf{F}/\delta\mathbf{x}$ and $\delta\mathbf{F}/\delta\mathbf{u}$. We will represent them in neural networks. Kolter (Kolter and Ng, 2009) has suggested that they can be guessed for some robotic systems, but in general the sensitivities are time-varying and can even reverse sign as the system state changes. Outside of the framework of optimal control, these Jacobian matrices have been learned by Gaàl (Gaàl, 1995), Hinton and McClelland (Hinton and McClelland, 1988), and by Abdelghani et al. (Abdelghani et al., 2008) by measuring time derivatives of the system dynamics. In such an approach, we take advantage of the fact that

$$\dot{\mathbf{F}}_{\text{system}} = \delta\mathbf{F}_{\text{system}}/\delta\mathbf{x} \cdot \dot{\mathbf{x}} + \delta\mathbf{F}_{\text{system}}/\delta\mathbf{u} \cdot \dot{\mathbf{u}}.$$

Define the sensitivity model $J(\mathbf{x}, \mathbf{u})$ to be a network approximation of

$$[\delta\mathbf{F}_{\text{system}}/\delta\mathbf{x}, \delta\mathbf{F}_{\text{system}}/\delta\mathbf{u}].$$

Then

$$\frac{1}{2}\left\|\ddot{\mathbf{x}} - J(\mathbf{x}, \mathbf{u}) \cdot [\dot{\mathbf{x}}; \dot{\mathbf{u}}]\right\|^2$$

is a reconstruction error that can be used to train the sensitivity model. A nice property of this error is that we can compute it while operating the system. There are some drawbacks as well. Time derivatives are notoriously noisy, and we are cavalierly taking the second time derivative. Additionally, we are only learning J projected along the curve taken by the system dynamics. To guarantee that we can properly identify the components of J, we would need to excite the components of $[\dot{\mathbf{x}}; \dot{\mathbf{u}}]$ independently. This may be difficult since we have direct responsibility only over $\dot{\mathbf{u}}$. Finally, we also cannot control the magnitudes

of the time derivatives in a simple way; to do so would be tantamount to controlling the system, which is begging the question.

Yet since we already possess a forward model, we use it instead to estimate the sensitivity model. As we interact with the system, $\mathbf{F}_{\text{system}}$, at every operating point $\mathbf{z} = (\mathbf{x}, \mathbf{u})$, we run several iterations (N_{probe}) of sensitivity analysis on the model, $\mathbf{F}$, in which we choose perturbations $(\delta\mathbf{x}, \delta\mathbf{u})$ randomly with mean 0 and variance σ^2_{probe} and evaluate the reconstruction error. The reconstruction error is

$$\frac{1}{2}\left\|\Big(\mathbf{F}(\mathbf{z}+\delta\mathbf{z}) - \mathbf{F}(\mathbf{z})\Big) - J(\mathbf{z})\cdot\delta\mathbf{z}\right\|^2. \tag{3.3.4}$$

Here, we have total control over the perturbations and can excite the components of the state as easily as we can excite the control components. After a perturbation, we train the sensitivity model parameters to minimize this reconstruction error. The error derivative with respect to the (i, j)-unit in the sensitivity model is

$$\Big(\delta F_i - \sum_k J_{ik}(\mathbf{z})\cdot\delta z_k\Big)\delta z_j. \tag{3.3.5}$$

The expectation value of this error derivative scales with the variance of the perturbation $\langle\delta z_k \cdot \delta z_j\rangle = \sigma^2_{\text{probe}}\delta_{kj}$, where we have chosen uncorrelated perturbations. For example, before learning, when $J(\mathbf{z}) = 0$, the expected error derivative is

$$\begin{aligned}
\langle(\delta F_i - 0)\delta z_j\rangle &= \left\langle \sum_k (\delta F_i/\delta F_k) z_k z_j \right\rangle \\
&= \sum_k (\delta F_i/\delta F_k)\sigma^2_{\text{probe}}\delta_{kj} \\
&= (\delta F_i/\delta F_j)\sigma^2_{\text{probe}}.
\end{aligned}$$

To correct for this scaling, we divide by the variance of the perturbation

$$\left(\delta F_i - \sum_j J_{ik}(\mathbf{z}) \cdot \delta z_k\right)\delta z_j / \sigma^2_{\text{probe}} \tag{3.3.6}$$

when we deliver the error to the (i, j)-unit in the sensitivity model.

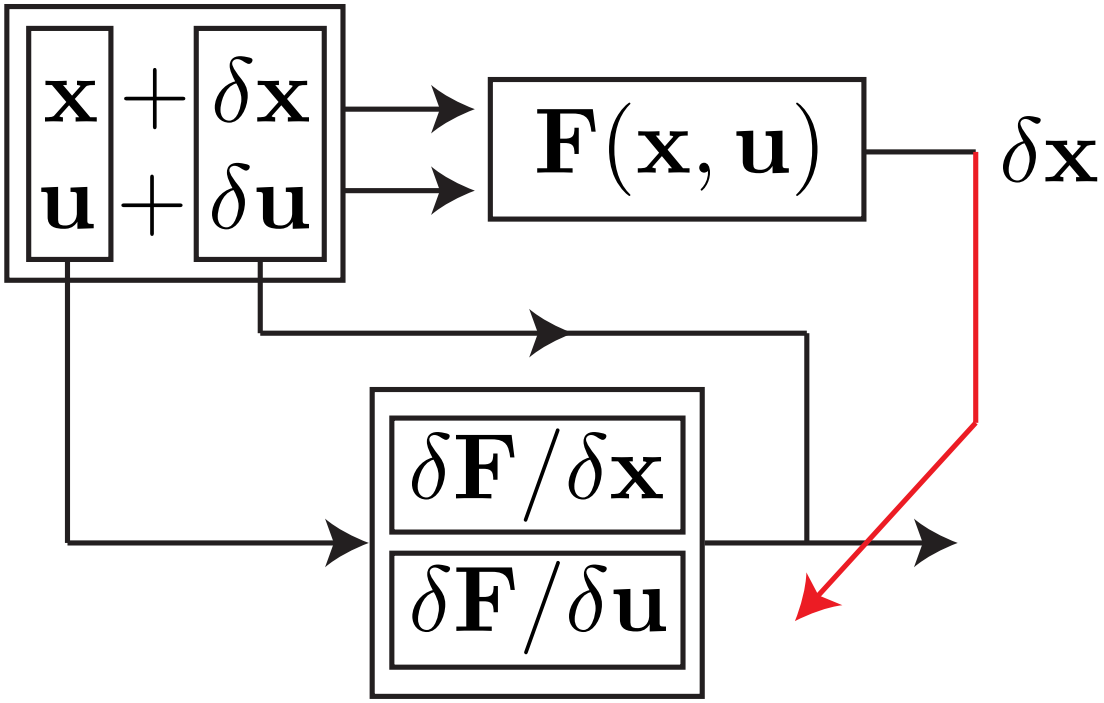

Figure 3.3: Learning a Sensitivity Model.

The sensitivity model linearizes the dynamics around the operating point $(\mathbf{x}, \mathbf{u})$. It takes the operating point as input and multiplies its output against the perturbation around this point to calculate the estimated perturbation to the output of the forward model.

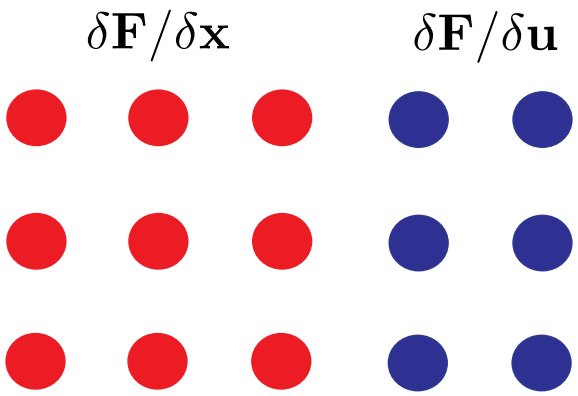

Figure 3.4: The Outputs of the Sensitivity Model.
Each output unit of the sensitivity model corresponds to one entry in the Jacobian matrices $\delta\mathbf{F}/\delta\mathbf{x}$ and $\delta\mathbf{F}/\delta\mathbf{u}$.

Controller Sensitivities

In addition to the sensitivity model of the plant, we also need the sensitivity derivatives of the controller itself, $\partial\mathbf{u}/\partial\mathbf{\Theta}$ and $\delta\mathbf{u}/\delta\mathbf{x}$. Although we could use the same mechanism as above to estimate them, we chose to evaluate these sensitivities analytically by backpropagation. Our choice is primarily one of convenience; to model the sensitivities with networks would create another inner loop of learning which would slow down the simulations due to the added computational burden and training time.

Cost Function Sensitivities

Finally, we need to know $\delta\mathcal{L}/\delta\mathbf{x}$ and $\delta\mathcal{L}/\delta\mathbf{u}$. Our method actually does not need to compute $\mathcal{L}(\mathbf{x},\mathbf{u})$ itself, so we produce the cost function gradients directly. These gradients are specified algebraically in a function.

3.3.4 The Planning Cycle

Once we have trained the forward model and the forward sensitivity model, we can address the control problem in full. At every small time step of our simulation, we perform the

following operations:

1: for n from 1 to N_{plan}.
2: initialize G, $\mathbf{q}$ to 0
3: for time t' step by dt' from t to $t + N\tau$
4: compute control $\mathbf{u}(\mathbf{x}; \boldsymbol{\Theta})$
5: integrate state by $\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, \mathbf{u})$
6: integrate G by equation 3.3.2
7: integrate $\mathbf{q}$ by equation 3.3.3
8: end for
9: update $\boldsymbol{\Theta}$ using $\mathbf{q}$ (using a gradient-based method – e.g., Nesterov's method)
10: end for
11: apply $\mathbf{u}(\mathbf{x}(t); \boldsymbol{\Theta})$ to system

3.4 Simulations

3.4.1 Torque-Limited Pendulum Swing-up

Problem Formulation

In the pendulum swing-up problem (Doya, 2000), a pendulum in a gravitational field is dropped at a random angle, and a torque must be applied to the pendulum bob to swing the pendulum into the upright position and balance it there. The pendulum equations are

$$\begin{aligned} \dot{\theta} &= \omega \\ ml^2\dot{\omega} &= -\mu\omega + mgl\sin(\theta) + u. \end{aligned}$$

We report specific parameter values in the Appendix. The state is $\mathbf{x} = [\theta; \omega]$, and the control is the single torque u. We make a transformation of the state to $\mathbf{x}' = [\cos(\theta); \sin(\theta); \omega]$ but from now on will refer to the transformed state as $\mathbf{x}$. The cost function for the problem takes the form

$$\begin{aligned} \mathcal{L}(\mathbf{x}, \mathbf{u}) &= \mathcal{L}_{\text{state}}(\mathbf{x}) + \mathcal{L}_{\text{control}}(\mathbf{u}) \\ &= -\cos(\theta) + \beta\omega^2/2 + \alpha(2/\pi)^2 u_{\max}\log(|\cos((\pi/2)u/u_{\max})|) \\ &= -x_1 + \beta x_3^2/2 + \alpha(2/\pi)^2 u_{\max}\log(|\cos((\pi/2)u/u_{\max})|). \end{aligned} \tag{3.4.1}$$

The form of $\mathcal{L}_{\text{control}}(\mathbf{u})$ deserves further discussion. We, of course, do not make use of backward accumulation to compute optimal controls in this paper, but, as we derive in the Appendix, in terms of the costate, the gradient of the cost-integral with respect to the

control variables is

$$\delta\mathcal{S}/\delta\mathbf{u} = \delta\mathcal{L}/\delta\mathbf{u} + \mathbf{p}^\top \cdot \delta\mathbf{F}/\delta\mathbf{u}$$

We can substitute the control cost from equation 3.4.1 into this equation.

$$\begin{aligned}\delta\mathcal{S}/\delta\mathbf{u} &= \delta(\alpha(2/\pi)^2 u_{\max}\log(|\cos((\pi/2)u/u_{\max})|)/\delta u + \mathbf{p}^\top \cdot \delta\mathbf{F}/\delta\mathbf{u} \\ &= (\alpha/u_{\max})(2/\pi)\tan((\pi/2)u/u_{\max}) + \mathbf{p}^\top \cdot \delta\mathbf{F}/\delta\mathbf{u}\end{aligned}$$

At an extremum, we have $\delta\mathcal{S}/\delta\mathbf{u} = 0$, so

$$u = u_{\max}(2/\pi)\tan^{-1}\Big(-(\pi/2)(u_{\max}/\alpha)\mathbf{p}^\top \cdot \delta\mathbf{F}/\delta\mathbf{u}\Big).$$

We also note that in this system the control enters linearly into the model equations, so $\delta^2\mathbf{F}/\delta\mathbf{u}^2 = 0$. Thus, the second derivative is simply the positive definite quantity

$$\delta^2\mathcal{S}/\delta\mathbf{u}^2 = (\alpha/u_{\max}^2)\sec((\pi/2)u/u_{\max})^2 > 0,$$

and we are guaranteed that the extremum is a minimum. Since $\tan^{-1}$ is bounded by $\pi/2$, the maximum torque is $u_{\max}$. Provided that the maximal control torque is small compared to the maximal torque imparted by gravity, the pendulum must be swung repeatedly back-and-forth to gain enough energy to reach the goal state, where x_1 and x_3 are nearly 0. For this reason, the torque-limited pendulum swing-up is a good model problem in optimal control. To minimize the cost integral, the optimal trajectory needs to pass transiently through states of high immediate cost, an exhibition of "delayed gratification."

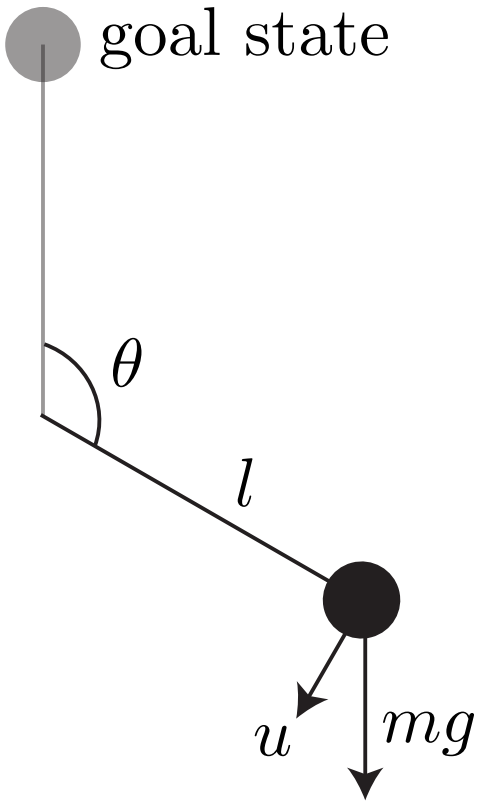

Figure 3.5: Torque-Limited Pendulum Swing-Up.
A torque u is applied to the pendulum bob to swing the pendulum toward the goal state. If the maximum torque $u_{\max}$ is limited so that $|u_{\max}| < |mgl|$, the trivial solution of pushing the pendulum directly to the top is not possible. Instead, a pumping solution must be found in which the pendulum swings back and forth in order to gain enough energy to reach the top.

For the sake of implementation we make a few adjustments to our equations. Unfortunately, if we represent θ as $(\cos(\theta), \sin(\theta))$ and examine the Jacobian in the new coordinates, we find discontinuities. Consider the term $\partial \dot{x}_1 / \partial x_1$. $\dot{x}_1 = d\cos(\theta)/d\theta \cdot \dot{\theta} = -\sin(\theta)\dot{\theta} = -\omega x_2$. This is

$$\begin{aligned} \partial[-\omega x_2]/\partial x_1 &= -\omega \cdot d\sin(\theta)/d\cos(\theta) \\ &= \omega \cot(\theta) \\ &= \omega x_1 / x_2 \end{aligned}$$

This diverges when $x_2 = 0$ at $\theta = 0$. This discontinuity is due to the interaction of our parameterization with the system equations. Our simple correction to remove the problem is to drop the use of differential equations. Instead of using time derivatives $\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, \mathbf{u})$, we integrate the equations for a time $\Delta t'$: $\mathbf{x}(t' + \Delta t') = \mathbf{F}_{\text{discrete}}(\mathbf{x}(t'), \mathbf{u}(t'))$. In discrete time, our updates for G and $\mathbf{q}$ change slightly. Now,

$$\begin{aligned}
\partial \mathbf{x}(t' + \Delta t')/\partial \mathbf{\Theta} &= \delta \mathbf{F}_{\text{discrete}}(\mathbf{x}(t'), \mathbf{u}(t'))/\delta \mathbf{x}(t') \cdot d\mathbf{x}(t')/d\mathbf{\Theta} \\
&+ \delta \mathbf{F}_{\text{discrete}}(\mathbf{x}(t'), \mathbf{u}(t'))/\delta \mathbf{u}(t') \cdot \partial \mathbf{u}(t')/\partial \mathbf{\Theta} \\
&+ \delta \mathbf{F}_{\text{discrete}}(\mathbf{x}(t'), \mathbf{u}(t'))/\delta \mathbf{u}(t') \cdot \delta \mathbf{u}(t')/\delta \mathbf{x}(t') \cdot d\mathbf{x}(t')/d\mathbf{\Theta}.
\end{aligned}$$

Or,

$$\begin{aligned}
G(t' + \Delta t') &= \delta \mathbf{F}_{\text{discrete}}/\delta \mathbf{x}(t') \cdot G(t') \\
&+ \delta \mathbf{F}_{\text{discrete}}/\delta \mathbf{u}(t') \cdot \partial \mathbf{u}(t')/\partial \mathbf{\Theta} \\
&+ \delta \mathbf{F}_{\text{discrete}}/\delta \mathbf{u}(t') \cdot \delta \mathbf{u}(t')/\delta \mathbf{x}(t') \cdot G(t').
\end{aligned}$$

Since our system really does evolve smoothly in time, the forward model's state prediction at time $t' + \Delta t'$ should be very close to the state at time t'. Therefore, it is useful to write the discrete time equations instead as $\mathbf{x}(t' + \Delta t') = \mathbf{x}(t') + \mathbf{F}_{\text{discrete}}(\mathbf{x}(t'), \mathbf{u}(t'))$, loading only the state changes in the forward model. It is this form that we have used in our computations. In this case,

$$\begin{aligned}
G(t' + \Delta t') &= G(t') + \delta \mathbf{F}_{\text{discrete}}/\delta \mathbf{x}(t') \cdot G(t') \\
&+ \delta \mathbf{F}_{\text{discrete}}/\delta \mathbf{u}(t') \cdot \partial \mathbf{u}(t')/\partial \mathbf{\Theta} \\
&+ \delta \mathbf{F}_{\text{discrete}}/\delta \mathbf{u}(t') \cdot \delta \mathbf{u}(t)/\delta \mathbf{x}(t') \cdot G(t').
\end{aligned}$$

In either case for $\mathbf{q}$ we have

$$\mathbf{q}(t' + \Delta t') \quad = \quad \mathbf{q}(t') + \frac{1}{\tau} \exp\Big[(t' - t)/\tau\Big]\Big(\delta\mathcal{L}/\delta\mathbf{u} \cdot \delta\mathbf{u}/\delta\mathbf{\Theta} + \delta\mathcal{L}/\delta\mathbf{x} \cdot G(t')\Big).$$

Network Structure

We have used radial basis function networks. The forward model and the forward sensitivity model have input $(\mathbf{x}, \mathbf{u})$, whereas the controller network has only state-related input $\mathbf{x}$. For an input vector $\mathbf{y}$, these networks compute their hidden layer activities $h_j(\mathbf{y})$ and outputs $z_i(\mathbf{h})$ as

$$\begin{aligned} h_j(\mathbf{y}) &= \frac{\exp(-\|\mathbf{y} - \mu_j\|^2/2\sigma^2)}{\sum_k \exp(-\|\mathbf{y} - \mu_k\|^2/2\sigma^2)}, \\ z_i(\mathbf{h}) &= \sum_j W_{ij} h_j(\mathbf{y}). \end{aligned}$$

In this study, we learn only the outgoing weights W, which are initialized to 0. We train the forward model and the forward sensitivity model using a normalized version of Amari's natural gradient method (Amari, 1998), which we derive in the Appendix.

Our pendulum controller has an additional output nonlinearity and produces its output according to $u_{\max} \tanh(z(\mathbf{h}))$. This guarantees that the control output stays inside the range where the cost function $\mathcal{L}_{\text{control}}$ is finite. It also gives the controller another saturating nonlinearity to work with, which simplifies the production of torques that are sensitive to the state but never grow too large.

Examining the Forward Model

We train the forward model by actuating with white-noise torques in the range $u \in (-5, 5)$. We start each trial from a uniform angle in (π, π) with angular velocity $\omega \in (-5/2\pi, 5/2\pi)$.

On every fifth learning trial, starting from trial 0, we compute the average prediction error of the forward model over the entire state space (with $u = 0$). To do so, we discretize the state space by marking a 25-by-25 grid on the θ interval from $(-\pi, \pi)$ and on the ω interval from $(-5/2\pi, 5/2\pi)$. We average the mean-squared error over those 25^2 grid points.

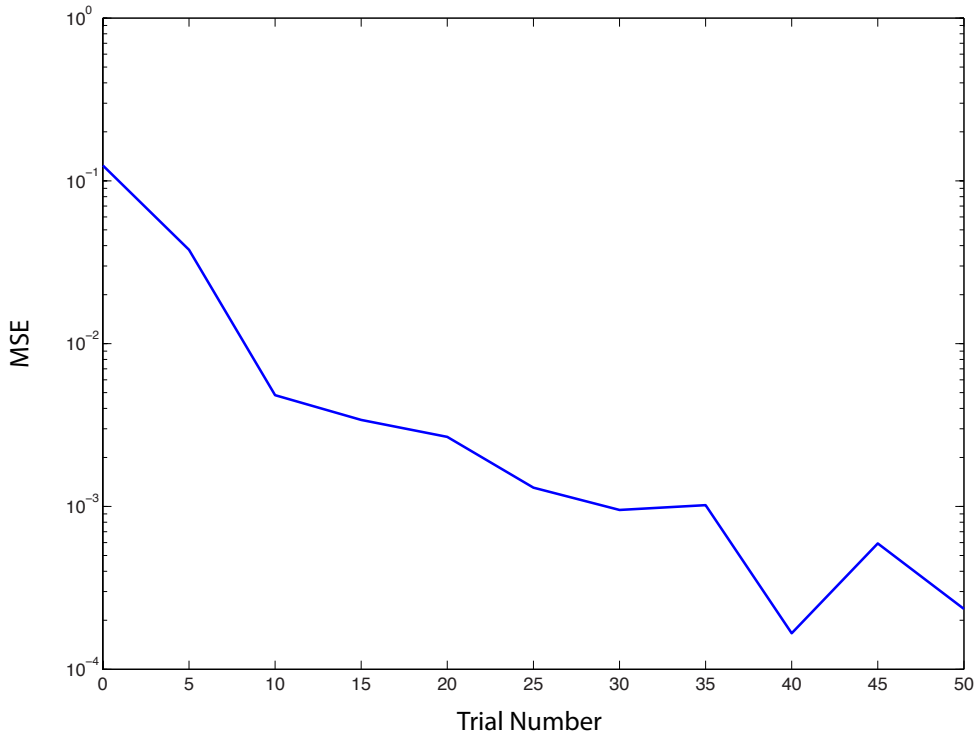

Figure 3.6: The Learning Curve for the Forward Model.
Within tens of trials, the forward model's predictions are extremely good.

Examining the Sensitivity Model

After we have trained the forward model for 50 trials, we train the sensitivity model. (Training the two models sequentially is not necessary. For the cart-pole problem in the next section, we trained the models at the same time. However, to generate learning curves for the sensitivity model that reflect learning processes in the sensitivity model alone, the forward model must be finalized before the sensitivity model is trained.) At each time step, we perform 10 perturbations on the forward model. The learning curve for the sensitivity model is very steep, and the outputs of the network very closely approximate the entries of the Jacobian of the forward model (Figure 3.6). The outputs are much less good at approximating the system Jacobian as calculated by finite differences in our sine and cosine parameterization (Figure 3.7). Since the forward model's error is low, and the sensitivity model is very accurate in its estimate of the Jacobian of the forward model, it would seem striking that the sensitivity model is not more accurate at predicting the Jacobian of the system equations. This is again an artifact of our parameterization, using sinusoids in place of angles. To perform finite differences, we add perturbations to the $(\sin(\theta), \cos(\theta))$ terms of the state vector, invert the parameterization to get θ, apply the system equations, then re-apply the transformation to get the new sine and cosine terms. By doing this, we have violated the constraint that these terms square to 1. The appropriate way to compute the system Jacobian under these constraints deserves further study, but for the present moment we leave it as a puzzle.

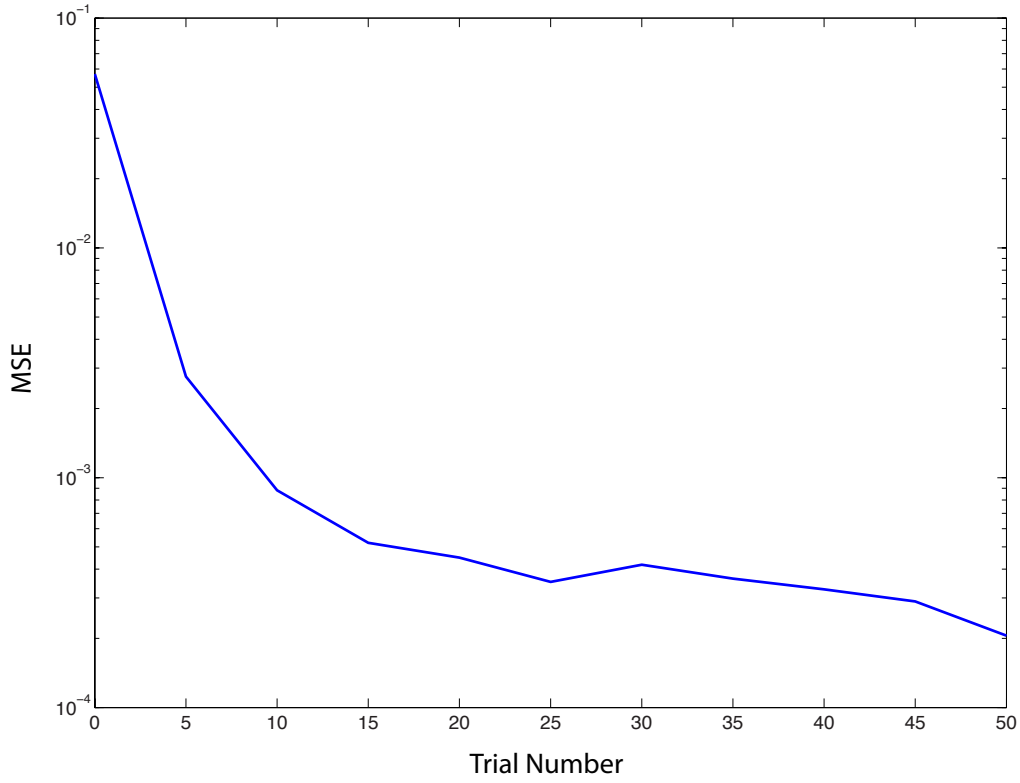

Figure 3.7: The Learning Curve for the Sensitivity Model.
Here, we are comparing the sensitivity model's predictions against the analytically computed Jacobian of the forward model.

Optimizing the Controller

At every time step of simulation, we run a planning cycle. After we calculate the gradient term **q**, we update the controller parameters, that is, the outgoing weights of the radial basis network, using Nesterov's accelerated gradient method (Nesterov, 1983), detailed in the Appendix. Nesterov's method is an entirely local, first-order update algorithm, but it has a convergence rate that is provably superior to simple gradient descent methods when applied to convex functions.[6] Once we have learned the forward model and forward sensitivity

[6]By way of disclaimer: our functions are not clearly convex, and we do not use the optimal parameter settings that Nesterov requires for his proof because extra calculations are necessary to set them.

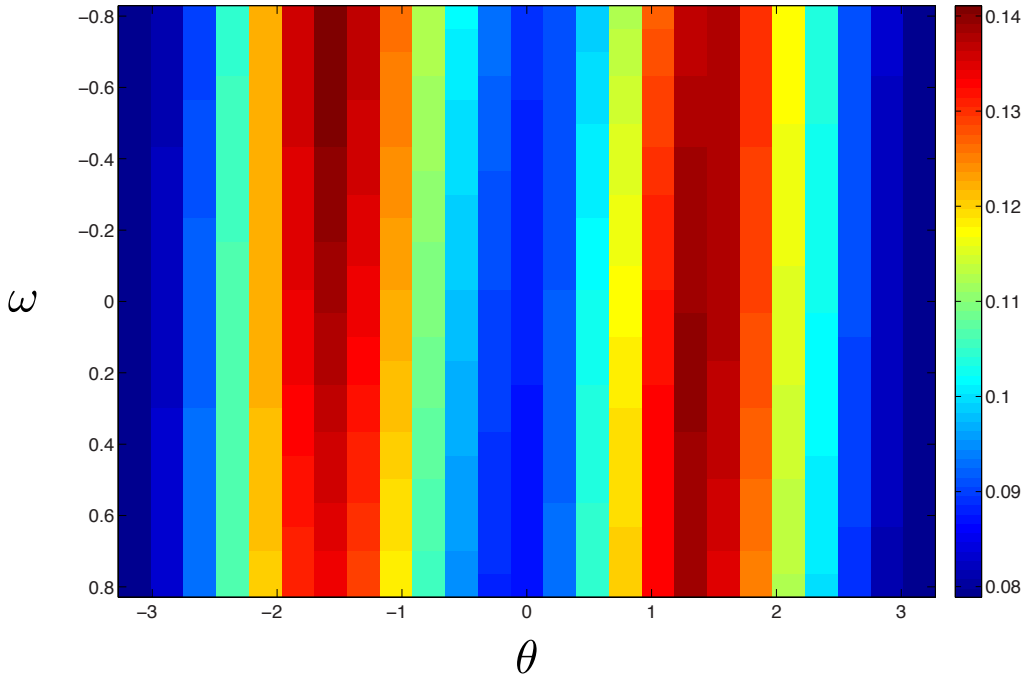

Figure 3.8: Finite-Differences Computation of Sensitivity Model Error.
The mean-squared error of the sensitivity model at each point on the grid based on finite-differencing the actual system dynamics when no torque is applied ($u = 0$). The error of the approximation to the system Jacobian is much higher than the approximation error of the Jacobian of the forward model was in Figure 3.7.

model, the planning cycle for the controller is sufficient to achieve swing-up on the first trial for the controller (Figure 3.9), using surprisingly few evaluations per cycle. We achieved success with fewer than 5 planning evaluations per time step but chose to use 5 because this provided very smooth swing-up trajectories (Video 3.1). In contrast, Doya (Doya, 2000) has exhibited a method that uses the exact system Jacobian yet still requires approximately 20 trials to attain frequent success.

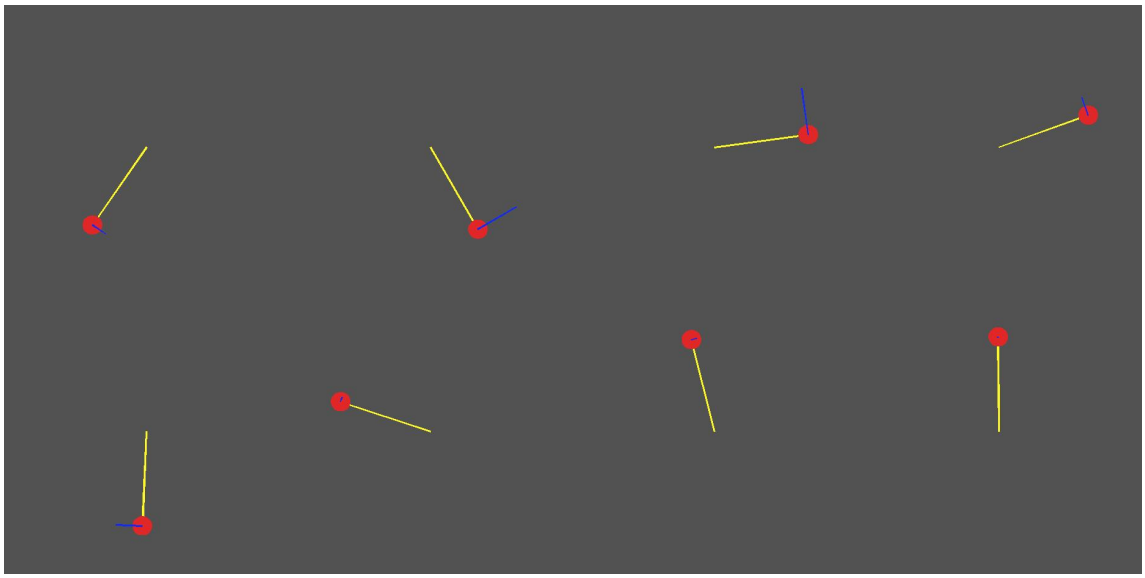

Figure 3.9: Key Frames of Pendulum Swing-Up.
To reach the top, a back-and-forth swinging action has been discovered on the first trial. The frames read top-left to top-right, then bottom-left to bottom-right.

3.4.2 Cart-Pole Swing-Up

Problem Formulation

In the cart-pole problem, a cart on a track is attached to a pole that must be swung up from an arbitrary angle and balanced. There are 4 state variables (θ, ω, x, v) and one control variable u, a force that is applied to push the cart back-and-forth. The equations of the cart (Florian, 2007) are

$$\begin{aligned}
\dot{\theta} &= \omega, \\
\dot{\omega} &= \frac{(m_c + m_p)g\sin(\theta) + \cos(\theta)(-u - m_p l\dot{\theta}^2\sin(\theta))}{l(\frac{4}{3}(m_c + m_p) - m_p\cos(\theta)^2)}, \\
\dot{x} &= v, \\
\dot{v} &= \frac{u + m_p l(\dot{\theta}^2\sin(\theta) - \dot{\omega}\cos(\theta))}{m_c + m_p}.
\end{aligned}$$

An important property of these dynamics is that the derivative with respect to the control variable is a function of the state in the second equation. By contrast, in the pendulum problem, the sensitivities with respect to the control are constant. This makes the construction of the forward and sensitivity models for the cart-pole system qualitatively more difficult; hence, the cart-pole problem serves as a more strenuous test of our methods.

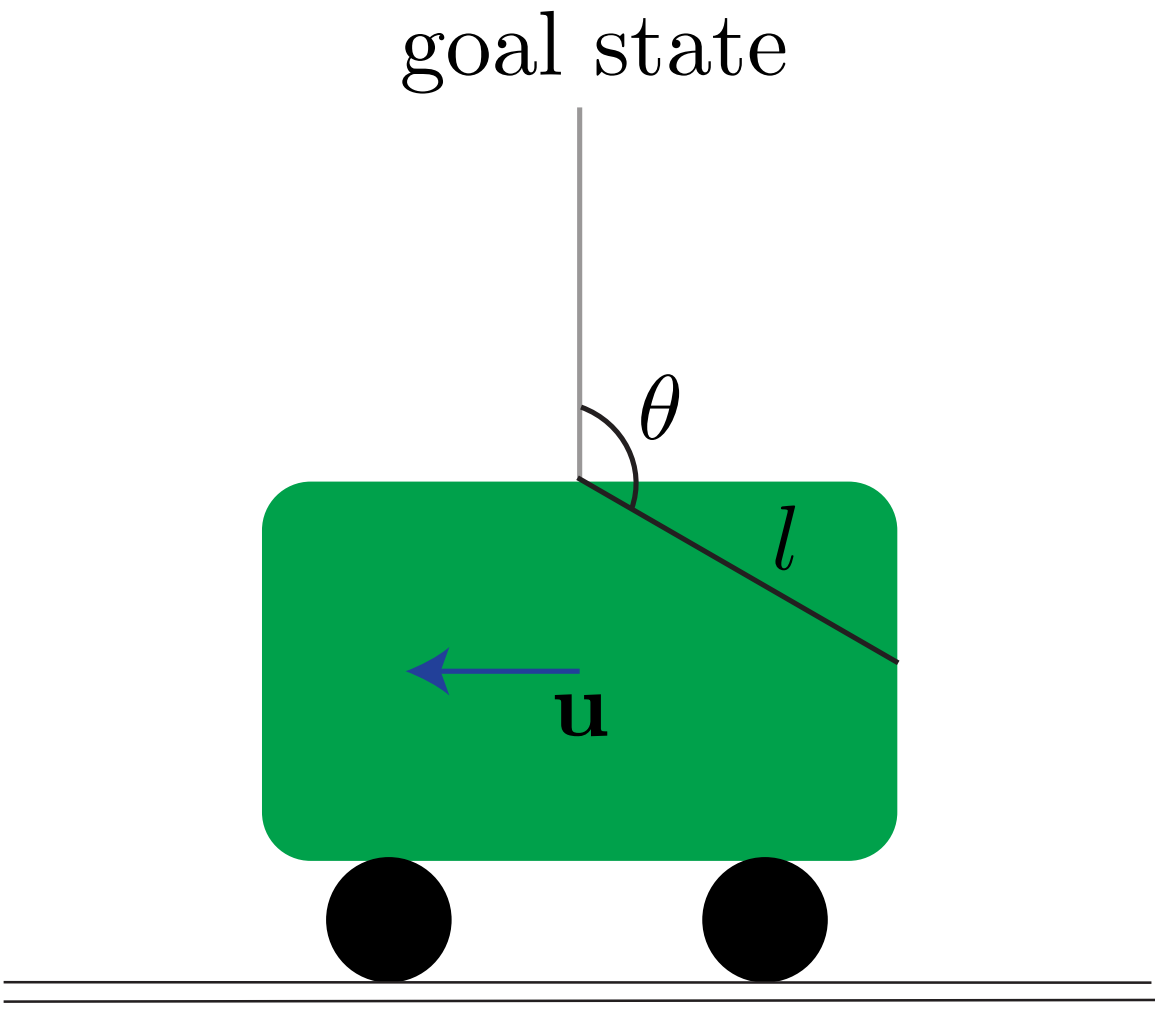

Figure 3.10: Cart-Pole Swing-Up.
In the cart-pole problem, a limited force must be applied to a cart on a track to swing the stick upright and balance it.

We again process the angle by taking the sine and cosine to get the entire state vector $\mathbf{x} = [\cos(\theta); \sin(\theta); \omega; x; v]$ and define a cost function that rewards upright pole positions while discouraging the use of large forces and states with high rotational and translational velocities.

$$\begin{aligned} \mathcal{L}(\mathbf{x}, \mathbf{u}) &= -\cos(\theta) + \alpha u^2/2 + \beta v^2/2 + \gamma \omega^2/2 \\ &= -x_1 + \alpha u^2/2 + \beta x_5^2/2 + \gamma x_3^2/2. \end{aligned}$$

Building the Models

Because of the increased dimensionality of this problem, learning the forward model and the sensitivity models was challenging. None of our models received information about the x-coordinate, facilitating the learning of translationally invariant policies. To expedite learning, we switched among learning trials that started from several different likely configurations along paths to the goal state. In one configuration, the angle of the pole was random, the angular velocity was 0, and the cart was still. In another configuration, the pole was close to upright with 0 angular velocity, and the cart was moving quickly. In a third, we uniformly sampled the state space, bounding the maximum velocities. We simultaneously trained the forward model and the sensitivity model by first training the forward model on a new pattern, then experimenting on the forward model with 5 perturbations to train the sensitivity model. Once the models were trained, we could again optimize trajectories on the first trial, using as few as 10, but usually 20, gradient updates per planning cycle. Our resultant controller was able to solve the problem in real-time (Video 3.2).

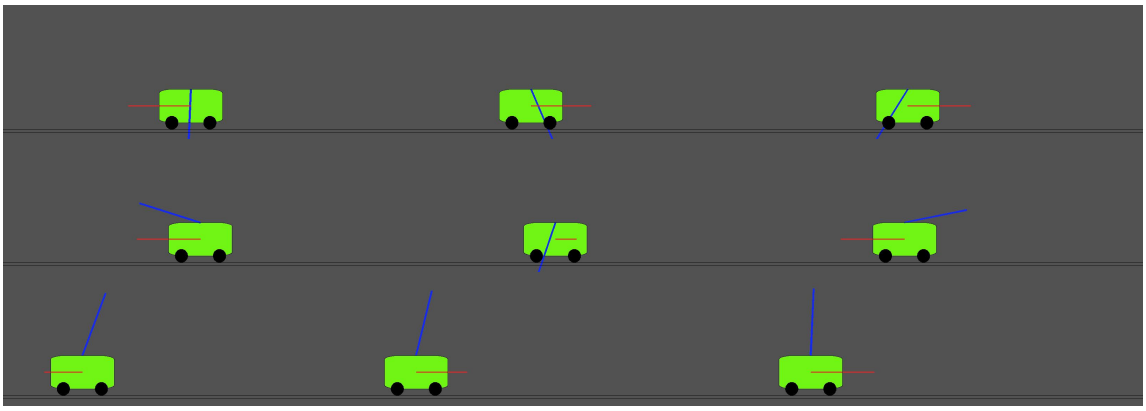

Figure 3.11: Key Frames of Cart-Pole.
Key frames in the first trial of the cart-pole swing-up problem.

3.5 Discussion

We have demonstrated that it is possible to learn forward models of unstable dynamical systems and build sensitivity models by experimentally perturbing those forward models. Despite the presence of modeling inaccuracies, we can still solve optimal control problems, and we can do so with a *tabula rasa* controller network on the first trial.

By combining our sensitivity models with forward accumulation-based optimization, we can forego the usual memory structures required by conventional solution methods used in optimal control. One of the typical critiques of optimal control calculations is that they are not performed by network structures but instead use the full resources of the computer on which the model is executed to perform external computations. Such critiques have long compelled researchers to investigate reinforcement learning algorithms for network optimization, despite known efficiency limitations. We hope that this study stimulates researchers to think broadly about what networks truly cannot do. The architecture of the motor system may be more structured, strange, and marvelous than we currently speculate.

3.6 Appendix

3.6.1 Derivation of Backward Accumulation

In a similar manner to our previous derivation of backpropagation in Chapter 1, we can get cost-integral gradients for equation (2). First, we create a constrained optimization problem, guaranteeing that the dynamical equations of the plant remained satisfied.

$$\begin{aligned}\mathcal{S} &= \int_t^{t+N\tau} dt' \frac{1}{\tau} \exp\Big[-(t'-t)/\tau\Big]\mathcal{L}\Big(\mathbf{x}(t'), \mathbf{u}(\mathbf{x}(t'); \mathbf{\Theta})\Big) + \lambda(t')^\top[\mathbf{F}(\mathbf{x}(t'), \mathbf{u}(\mathbf{x}(t'); \mathbf{\Theta}) - \dot{\mathbf{x}}(t')] \\ &= \int_t^{t+N\tau} dt' \frac{1}{\tau} \exp\Big[-(t'-t)/\tau\Big]\Big\{\mathcal{L}\Big(\mathbf{x}(t'), \mathbf{u}(\mathbf{x}(t'); \mathbf{\Theta})\Big) + \mathbf{p}(t')^\top[\mathbf{F}(\mathbf{x}(t'), \mathbf{u}(\mathbf{x}(t'); \mathbf{\Theta}) - \dot{\mathbf{x}}(t')]\Big\},\end{aligned}$$

defining $\mathbf{p}(t') = (1/\tau)\exp[(t'-t)/\tau)]\lambda(t')$ to simplify further calculation. The term

$$\int_t^{t+N\tau} dt' \frac{1}{\tau} \exp\Big[-(t'-t)/\tau\Big]\mathbf{p}(t')^\top[-\dot{\mathbf{x}}(t')]$$

can be integrated by parts:

$$\begin{aligned}\int_t^{t+N\tau} dt' \frac{1}{\tau} \exp\Big[-(t'-t)/\tau\Big]\mathbf{p}(t)^\top[-\dot{\mathbf{x}}(t')] &= \Big[\frac{1}{\tau}\exp[-(t'-t)/\tau]\mathbf{p}(t')^\top[-\mathbf{x}(t')]\Big]\Big|_t^{t+N\tau} \\ &\quad + \int_t^{t+N\tau} dt' \frac{d}{dt'}\Big[\frac{1}{\tau}\exp[-(t'-t)/\tau]\mathbf{p}(t')^\top\Big]\mathbf{x}(t') \\ &= -\Big[\frac{1}{\tau}\exp[-(t'-t)/\tau]\mathbf{p}(t')^\top\mathbf{x}(t')\Big]\Big|_t^{t+N\tau} \\ &\quad + \int_t^{t+N\tau} dt' \frac{1}{\tau}\exp[-(t'-t)/\tau]\Big[\dot{\mathbf{p}}(t')^\top - \frac{1}{\tau}\mathbf{p}(t')^\top\Big]\mathbf{x}(t').\end{aligned}$$

If we reinsert this into the full equation, we have

$$\begin{aligned}\mathcal{S} &= \int_t^{t+N\tau} dt' \frac{1}{\tau}\exp\Big[-(t'-t)/\tau\Big]\Big\{\mathcal{L}\Big(\mathbf{x}(t'),\mathbf{u}(\mathbf{x}(t');\boldsymbol{\Theta})\Big)+\mathbf{p}(t')^\top\mathbf{F}(\mathbf{x}(t'),\mathbf{u}(\mathbf{x}(t');\boldsymbol{\Theta})\\ &\quad+\Big[\dot{\mathbf{p}}(t')^\top-\frac{1}{\tau}\mathbf{p}(t')^\top\Big]\mathbf{x}(t')\Big\}-\Big[\frac{1}{\tau}\exp[-(t'-t)/\tau]\mathbf{p}(t')^\top\mathbf{x}(t')\Big]\Big|_t^{t+N\tau}.\end{aligned}$$

We can take the variation of $\mathcal{S}$ with respect to the state variables at a given time. Ignoring the boundary condition for a moment, the variation at times not on the boundary is given by

$$\begin{aligned}\delta\mathcal{S}/\delta\mathbf{x}(t')\cdot\delta\mathbf{x}(t') &= \Big(\delta\mathcal{L}/\delta\mathbf{x}+\delta\mathcal{L}/\delta\mathbf{u}\cdot\delta\mathbf{u}/\delta\mathbf{x}\Big)\cdot\delta\mathbf{x}+\mathbf{p}^\top\cdot\Big(\delta\mathbf{F}/\delta\mathbf{x}+\delta\mathbf{F}/\delta\mathbf{u}\cdot\delta\mathbf{u}/\delta\mathbf{x}\Big)\cdot\delta\mathbf{x}\\ &\quad+\Big(\dot{\mathbf{p}}^\top-\frac{1}{\tau}\mathbf{p}^\top\Big)\cdot\delta\mathbf{x}.\end{aligned}$$

At an optimum, the variation with respect to the state variables is 0. Therefore,

$$\begin{aligned}0 &= \Big(\delta\mathcal{L}/\delta\mathbf{x}+\delta\mathcal{L}/\delta\mathbf{u}\cdot\delta\mathbf{u}/\delta\mathbf{x}\Big)\cdot\delta\mathbf{x}+\mathbf{p}^\top\cdot\Big(\delta\mathbf{F}/\delta\mathbf{x}+\delta\mathbf{F}/\delta\mathbf{u}\cdot\delta\mathbf{u}/\delta\mathbf{x}\Big)\cdot\delta\mathbf{x}\\ &\quad+\Big(\dot{\mathbf{p}}^\top-\frac{1}{\tau}\mathbf{p}^\top\Big)\cdot\delta\mathbf{x}.\end{aligned}$$

For nonzero $\delta\mathbf{x}$, this can only be satisfied if

$$\dot{\mathbf{p}} = -\delta\mathcal{L}/\delta\mathbf{x}^\top-\delta\mathbf{u}/\delta\mathbf{x}^\top\cdot\delta\mathcal{L}/\delta\mathbf{u}^\top-\Big(\delta\mathbf{F}/\delta\mathbf{x}^\top+\delta\mathbf{u}/\delta\mathbf{x}^\top\cdot\delta\mathbf{F}/\delta\mathbf{u}^\top\Big)\cdot\mathbf{p}+\frac{1}{\tau}\mathbf{p}.$$

The variation due to the control variables can be calculated similarly:

$$\delta\mathcal{S}/\delta\mathbf{u}(t') = \frac{1}{\tau}\exp\Big[-(t'-t)/\tau\Big]\Big(\delta\mathcal{L}/\delta\mathbf{u}+\mathbf{p}^\top\cdot\delta\mathbf{F}/\delta\mathbf{u}\Big).$$

This is blessedly simple in comparison. Since the parameters $\boldsymbol{\Theta}$ only interact with the cost through the control variables, we can calculate the gradient with respect to them as

$\int dt' \delta \mathcal{S}/\delta \mathbf{u}(t') \cdot \partial \mathbf{u}(t')/\partial \mathbf{\Theta}$.

For the variation at the boundary:

$$-\left[\frac{1}{\tau}\exp[-(t'-t)/\tau]\mathbf{p}(t')^\top \delta \mathbf{x}(t')\right]\Big|_t^{t+N\tau} = \frac{1}{\tau}\mathbf{p}(t)^\top \delta \mathbf{x}(t) - \frac{1}{\tau}\exp[-N]\mathbf{p}(t+N\tau)^\top \delta \mathbf{x}(t+N\tau).$$

Since the variation with respect to these terms must also be 0, we have that $\mathbf{p}(t+N\tau) = 0$. We also see that $\delta \mathcal{S}/\delta \mathbf{x}(t) = \frac{1}{\tau}\mathbf{p}(t)$. The former equation implies that we must calculate equation 3.6.1 by integrating backward from $t + N\tau$ since the boundary condition is only known at that time. The latter equation demonstrates that the gradient of the cost-integral with respect to a state at a given time is given by the quantity $\frac{1}{\tau}\mathbf{p}(t)$, also known as the "costate."

3.6.2 Derivation of Forward Accumulation

First, define the quantity $G_{ij}(t') \equiv dx_i(t')/d\Theta_j$. The system equations dictate that $\mathbf{x}(t') = \int_t^{t'} ds \mathbf{F}(\mathbf{x}(s), \mathbf{u}(\mathbf{x}(s); \mathbf{\Theta}))$. Therefore,

$$\begin{aligned} G(t') &= \int_t^{t'} ds \partial \mathbf{F}(\mathbf{x}(s), \mathbf{u}(\mathbf{x}(s); \mathbf{\Theta}))/\partial \mathbf{\Theta} \\ &= \int_t^{t'} ds \delta \mathbf{F}/\delta \mathbf{x}(s) \cdot d\mathbf{x}(s)/d\mathbf{\Theta} + \delta \mathbf{F}/\delta \mathbf{u} \cdot \left(\partial \mathbf{u}/\partial \mathbf{\Theta} + \delta \mathbf{u}(s)/\delta \mathbf{x}(s) \cdot d\mathbf{x}(s)/d\mathbf{\Theta}\right) \\ &= \int_t^{t'} ds \delta \mathbf{F}/\delta \mathbf{x}(s) \cdot G(s) + \delta \mathbf{F}/\delta \mathbf{u} \cdot \left(\partial \mathbf{u}/\partial \mathbf{\Theta} + \delta \mathbf{u}/\delta \mathbf{x} \cdot G(s)\right) \end{aligned}$$

We can generate a differential equation for G:

$$\dot{G} = \delta \mathbf{F}/\delta \mathbf{x} \cdot G + \delta \mathbf{F}/\delta \mathbf{u} \cdot \left(\partial \mathbf{u}/\partial \mathbf{\Theta} + \delta \mathbf{u}/\delta \mathbf{x} \cdot G\right)$$

We then use the decomposition

$$\delta\mathcal{S}/\delta\mathbf{\Theta} = \int_t^{t+N\tau} dt' \frac{1}{\tau} \exp\Big[-(t'-t)/\tau \Big] \Big(\delta\mathcal{L}/\delta\mathbf{u}(t') \cdot \partial\mathbf{u}(t')/\partial\mathbf{\Theta} + \delta\mathcal{L}/\delta\mathbf{x}(t') \cdot G(t') \Big).$$

Define $\mathbf{q} \equiv \partial\mathcal{S}/\partial\Theta$. Our previous equation, too, can be integrated forward-in-time:

$$\dot{\mathbf{q}} = \frac{1}{\tau} \exp\Big[-(t'-t)/\tau \Big] \Big(\delta\mathcal{L}/\delta\mathbf{u} \cdot \partial\mathbf{u}/\partial\mathbf{\Theta} + \delta\mathcal{L}/\delta\mathbf{x} \cdot G \Big).$$

3.6.3 Pendulum Problem

Simulation Parameter	Value	Notes
m	$1\ kg$	
l	$1\ m$	
$u_{\max}$	$5\ kg \cdot m^2/s^2$	units of torque
α	0.1	
β	0.05	
g	$9.8 \cdot m/s^2$	
μ	$0.1 kg \cdot m^2/s$	
τ	$2s$	
dt	$0.09s$	
N_{horizon}	1	We optimize over $1\tau \Rightarrow \lceil \tau/dt \rceil = 23$ steps
N_{plan}	5	
N_{pert}	10	
η_{fm}	1	forward model learning rate
η_{sm}	0.1	sensitivity model learning rate
ρ	0.95	for Nesterov's method
ν	0.1	for Nesterov's method
σ_{forward}	2.4	
$\sigma_{\text{sensitivity}}$	2.4	
$\sigma_{\text{controller}}$	0.12	

We use a second-order Runge-Kutta method to integrate the pendulum equations. Every time we call this method, we integrate 3 successive times with time step $dt/3$. This allows us to maintain numerical accuracy while still learning the forward model on the relatively

coarse time scale of 0.09 seconds.

3.6.4 Cart-Pole Problem

Simulation Parameter	Value	Notes
m_p	0.1 kg	pole mass
m_c	1.0 kg	cart mass
l	0.5 m	pole length is $2l$
$u_{\max}$	10 $kg \cdot m/s^2$ (Newtons)	
α	0.05	
β	0.05	
γ	0.05	
g	$9.8 \cdot m/s^2$	
μ	$0.1kg \cdot m^2/s$	
τ	$1s$	
dt	$0.09s$	
N_{horizon}	1	We optimize over $1\tau \Rightarrow \lceil \tau/dt \rceil = 12$ steps
N_{plan}	20	
N_{pert}	5	
η_{fm}	1	forward model learning rate
η_{sm}	0.1	sensitivity model learning rate
ρ	0.99	for Nesterov's method
ν	0.01	for Nesterov's method
σ_{forward}	15.5	
$\sigma_{\text{sensitivity}}$	15.5	
$\sigma_{\text{controller}}$	0.99	

3.6.5 Network Learning

Setting the Radial Basis Function Parameters

For the μ-s, we tile the space of allowed state configurations evenly. For each independent dimension d, we define a lower bound b and upper bound a. The interval length $I = a - b$ is divided into $M_d + 1$ pieces, and we define $\mu_j^d = j \cdot (I/M_d) + b$ for $j = 0 \ldots M_d$. When our dimensions represent cosines and sines, we tile in the space of $\theta \in [0, 2\pi]$ and then transform by taking the cosine and sine. Our basis function centers are constructed as the terms of the Cartesian product of the center points along the individual dimensions $\mu_{k,l,m,\ldots} = (\mu_k^1, \mu_l^2, \mu_m^3, \ldots)$. Finally, we collapse or vectorize the multi-index $\mu_{k,l,m,\ldots}$ into a single linear index μ_j for $j = 1 \ldots N_{\text{basis}}$. To set σ, we compute the volume of the whole space V. (Again, co-dependent coordinates do not count as multiple dimensions.) Then the volume occupied by the basis functions is of order $O(N_{\text{basis}} \cdot \sigma^D)$. Setting $V = N_{\text{basis}} \cdot \sigma^D$, we find we cover the whole space if $\sigma \approx (V/N_{\text{basis}})^{1/D}$. In practice, this order of magnitude estimate for σ was multiplied by a coefficient, and the coefficient was tested over a range of 10^{-1} through 10 for fastest learning.

For the pendulum problem basis function centers, we grid the θ interval $(-\pi, \pi)$ at 15 points, the ω interval $(-5/2\pi, 5/2\pi)$ at 15 points, and the u interval $(-5, 5)$ at 10 points.

For the cart-pole problem basis function centers, we grid the θ interval $(-\pi, \pi)$ at 8 points, the ω interval $(-3\pi, 3\pi)$ at 6 points, the v interval $(-3, 3)$ at 5 points, and the u interval $(-10, 10)$ at 5 points. We did not include the x variable as input to any of the networks.

A Normalized Version of Amari's Natural Gradient

Given a metric g_{ij} that determines vector lengths, we construct an online learning rule that simultaneously makes the error on the current pattern zero while minimizing the length of

the change to the weight vector according to the metric.

Define Δw_i to be the change to weight w_i. Our training set consists of patterns $(\mathbf{x}, y)$, where $\mathbf{x}$ is a vector and y is a scalar. (We perform the derivation here for scalar output, but the results generalize trivially at the end to vector output.) Our cost function for a new pattern $\mathbf{x}$ is:

$$\mathcal{L}(\mathbf{x}, \Delta\mathbf{w}, \lambda) = \frac{1}{2}\sum_{i,j} \Delta w_i g_{ij} \Delta w_j + \lambda[y - \sum_i (w_i + \Delta w_i) x_i]$$

If we minimize with respect to Δw_i, we get

$$\frac{\partial \mathcal{L}(\mathbf{x}, \Delta\mathbf{w}, \lambda)}{\partial \Delta w_i} = \sum_j g_{ij} \Delta w_j - \lambda x_i = 0.$$

We can invert the metric by raising indices to get

$$\Delta w_k = \lambda \sum_i g^{ki} x_i.$$

To complete the specification of the learning rule, we must impose the constraint that $y - \sum_i (w_i + \Delta w_i) x_i = 0$. If we define the error on the current pattern as $\delta = y - \sum_i w_i x_i$, then we can see that

$$\begin{aligned} y - \sum_i (w_i + \Delta w_i) x_i &= 0 \\ \Rightarrow \delta &= \sum_i \Delta w_i x_i \\ &= \lambda \sum_{i,j} x_i g^{ij} x_j. \end{aligned}$$

We can solve for λ.

$$\lambda = \frac{\delta}{\sum_{i,j} x_i g^{ij} x_k}.$$

Substituting λ into the weight update gives us

$$\Delta w_k = \frac{\delta \cdot \sum_i g^{ki} x_i}{\sum_{i,j} x_i g^{ij} x_j}. \tag{3.6.1}$$

This generalizes the normalized least mean squares learning rule (Haykin, 2003) to the case of a general metric. Amari (Amari, 1998) has studied Riemannian metrics associated with the parameter spaces of many networks. Here, we merely use the sample covariance matrix $C_{ij}(T) = (1/T) \sum_{t=1}^{T} x_i(t) x_j(t)$ as a simple but powerful metric. In this case, on the T-th time step, we update the weights by

$$\Delta w_k(T) = \frac{\delta \cdot \sum_i C^{ki}(T) x_i(T)}{\sum_{i,j} x_i(T) C^{ij}(T) x_j(T)}, \tag{3.6.2}$$

where we have taken the inverse covariance matrix by raising the indices. We can also compute C^{ki} or C^{-1} recursively. Suppose we have an estimate at time T of the inverse covariance $C^{-1}(T)$, and we want an estimate of the inverse covariance at time $T + 1$. First note that the covariance matrix itself can be written recursively:

$$\begin{aligned} C_{ij}(T+1) &= \frac{1}{T+1} \sum_{t=1}^{T+1} x_i(t) x_j(t) \\ &= \frac{1}{T+1} \sum_{t=1}^{T} x_i(t) x_j(t) + \frac{1}{T+1} x_i(T+1) x_j(T+1) \\ &= \frac{T}{T+1} C_{ij}(T) + \frac{1}{T+1} x_i(T+1) x_j(T+1) \end{aligned}$$

To find a recursive update for the inverse of the covariance matrix, we employ the Sherman-

Morrison lemma which states that for an invertible matrix A and a rank-one outer-product $\mathbf{u}\mathbf{v}^\top$

$$(A + \mathbf{u}\mathbf{v}^\top)^{-1} = A^{-1} - \frac{A^{-1}\mathbf{u}\mathbf{v}^\top A^{-1}}{1 + \mathbf{v}^\top A^{-1}\mathbf{u}} \tag{3.6.3}$$

Here, we can apply the result to compute the inverse covariance recursively.

$$\begin{aligned} C^{-1}(T+1)^{-1} &= \left(\frac{T}{T+1}C(T) + \frac{1}{T+1}\mathbf{x}(T+1)\mathbf{x}^\top(T+1)\right)^{-1} \\ &= (T+1)\left(T^{-1}C^{-1}(T) - \frac{T^{-1}C^{-1}(T)\mathbf{x}(T+1)\mathbf{x}(T+1)^\top T^{-1}C^{-1}(T)}{1 + \mathbf{x}(T+1)^\top T^{-1}C^{-1}(T)\mathbf{x}(T+1)}\right) \\ &= \frac{T+1}{T}\left(C^{-1}(T) - \frac{C^{-1}(T)\mathbf{x}(T+1)\mathbf{x}(T+1)^\top C^{-1}(T)}{T + \mathbf{x}(T+1)^\top C^{-1}(T)\mathbf{x}(T+1)}\right). \end{aligned}$$

We can efficiently automate the learning rule update at time $T+1$ using the following algorithm:

1: $\mathbf{k} := C^{-1}(T)\mathbf{x}(T+1)$
2: $d := \mathbf{x}(T+1)^\top \mathbf{k}$
3: $\gamma := 1/(T+d)$
4: $C^{-1}(T+1) = \frac{T+1}{T}(C^{-1}(T) - \gamma\mathbf{k}\mathbf{k}^\top)$
5: $\mathbf{w}(T+1) := \mathbf{w}(T) + \gamma\delta\mathbf{k}^\top$

For multidimensional outputs, the vector $\mathbf{w}$ is just replaced with a matrix W, and the scalar δ becomes a vector of errors. If we have noisy target patterns, making the error 0 with each learning step is overly aggressive. In this case, we can modify the weight update above to make $\mathbf{w}(T+1) := \mathbf{w}(T) + \eta\gamma\delta\mathbf{k}^\top$ for some $\eta \in (0, 1]$. Appealingly, for $\eta = f/100$, we make an $f\%$ reduction in error after learning on the current pattern.

Nesterov's Accelerated Gradient

1: for n from 1 to $N_{\mathbf{plan}}$
2: evaluate $\mathbf{q}|_{\boldsymbol{\Theta}+\rho\mathbf{v}}$ using one iteration of the planning cycle
3: $\mathbf{v} := \rho\mathbf{v} - \nu\mathbf{q}$
4: $\boldsymbol{\Theta} := \boldsymbol{\Theta} + \mathbf{v}$
5: end for

Chapter 4

Conclusion

4.1 What Should We Look For?

Nate Sawtell has asked me to consider what a physiologist should search for if he were to take any of these ideas seriously. I have to admit, I'm not very well-prepared for this question, even though it is the most natural question to ask. It is true that the levels of description of algorithms and biophysics are very different, and one superficially might think this is the difficulty. But this difference is a subjective one. The real problem is not that optimal control theory is difficult to pin down neurally, it is that algorithms themselves have proved elusive. I often feel that if we had just one fully worked-out circuit that we could clearly understand both dynamically and functionally, it would serve as a Rosetta Stone for the rest of the nervous system. We know that the brain must partake of exorbitant reuse of its functional components. The genome is only a gigabyte, after all. A little window may be enough to get a full glimpse.

We *do* know something about what a forward model might look like. The forward models we build take in the state and control command information and predict the state information at a delayed time. One might expect that this delay is exactly the conduction delay from the initiating motor discharge down to the spinal cord and musculature plus the time for proprioceptive signals to travel back up the spinal cord. In the human locomotor system, such a delay could be on the order of 200 milliseconds. In the Mormyrid electric fish, this delay is probably on the order of tens of milliseconds. A possible signature of the forward model then is a circuit that uses sensory and motor information at time t to predict sensory information that arrives at time $t + \Delta t$. To learn to associate the information at time t with the information at time $t + \Delta t$, the early-arriving information must be delayed artificially. In the Mormyrid electric fish, the unipolar brush cell could serve as a delay element.

The sensitivity model is a more speculative module, but it is perhaps simpler to look for. It

pairs a perturbation generator with a difference calculation. That is, the difference between the circuit output before and after the perturbation needs to be estimated. Such a difference can indeed be calculated by time derivatives; perhaps a mechanism of synaptic facilitation or residual calcium concentration in spiking cells could perform a differencing operation. The sensitivity model also requires a reasonable neural approximation to multiplication. In this case, the necessary clue is clear: if the firing rate of a neuron approximates the product of the firing rates of two other neurons, we have a suspect. Alternatively, if a neuron detects the coincidence of the spiking of two other neurons, it could implement multiplication in binary.

The ever-present dilemma throughout the history of neuroscience is whether the brain functions as a totally distributed system or a highly modular one. Noted historical proponents of the modular view included phrenologists like Franz Joseph Gall. Noted proponents of the distributionist view included Camillo Golgi, who believed in a sylvan reticulum. Whatever the final truth may be, we inhabit an era in which consensus opinion is swinging towards modularity. I expect that in another generation or so it will swing back – so it goes with the deep questions – but, for now, the age is on our side. Let's look for box diagrams!

Chapter 5

Bibliography

Bibliography

M N Abdelghani, T P Lillicrap, and D B Tweed. Sensitivity Derivatives for Flexible Sensorimotor Learning. *Neural computation*, 20(8):2085–2111, August 2008.

S Amari. Natural Gradient Works Efficiently in Learning. *Neural computation*, 10(2): 251–276, February 1998.

C G Atkeson. Efficient Robust Policy Optimization. Technical report, 2012.

F Attneave. Some informational aspects of visual perception. *Psychological Review*, 61 (3):183–193, May 1954.

H B Barlow. Possible principles underlying the transformation of sensory messages. *Sensory communication*, pages 217–234, 1961.

R E Bellman. *Dynamic Programming*. 1957.

C Brinkman, R Porter, and J Norman. Plasticity of motor behavior in monkeys with crossed forelimb nerves. *Science*, 1983.

R A Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 1986.

F Crick. The recent excitement about neural networks. *Nature*, 1989.

P Dayan. Goal-directed control and its antipodes. *Neural Networks*, 2009.

Kenji Doya. Reinforcement Learning in Continuous Time and Space. *Neural computation*, 12(1):219–245, January 2000.

T T Flash and N N Hogan. The coordination of arm movements: an experimentally confirmed mathematical model. *Journal of Neuroscience*, 5(7):1688–1703, July 1985.

R V Florian. Correct equations for the dynamics of the cart-pole system. *Center for Cognitive and Neural Studies (Coneural)*, 2007.

G Gaàl. Relationship of calculating the Jacobian matrices of nonlinear systems and population coding algorithms in neurobiology. *Physica D: Nonlinear Phenomena*, 1995.

E Gat. On Three-Layer Architectures. *Artificial Intelligence and Mobile Robots*, 1998.

A W Hantman and T M Jessell. Clarke's column neurons as the focus of a corticospinal corollary circuit. *Nature Neuroscience*, 2010.

S Haykin. Adaptive filter theory (ise). 2003.

W D Hillis and A Hart-Davis. The pattern on the stone. 1998.

G E Hinton and J L McClelland. Learning representations by recirculation. *Neural information processing ...*, 1988.

D Huh and E Todorov. Real-time motor control using recurrent neural networks. *In proceedings of the 2nd IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2009.

M I Jordan and D E Rumelhart. Forward Models: Supervised Learning with a Distal Teacher. *Cognitive Science*, 1992.

K Kavukcuoglu, M Ranzato, and Y LeCun. Fast inference in sparse coding algorithms with applications to object recognition. *Technical Report CBLL-TR-2008-12-01: Computational and Biological Learning Lab, Courant Institute, NYU.*, 2008.

M Kawato, K Furakawa, and R Suzuki. A Hierarchical Neural-Network Model for Control and Learning of Voluntary Movement. *Biological Cybernetics*, 1987.

J Z Kolter. Learning and control with inaccurate models. *PhD thesis, Stanford*, 2010.

J Z Kolter and A Y Ng. Policy search via the signed derivative. *Robotics: science and systems*, 2009.

J W Krakauer. Personal Communication. 2013.

K S Lashley. Basic Neural Mechanisms in Behavior. *Psychological Review*, 1930.

Y LeCun. A theoretical framework for backpropagation. *Proceedings of the 1988 Connectionist Models Summer School*, 1988.

L J Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 1992.

H R Maei, C Szepesvári, S Bhatnagar, D Precup, D Silver, and R S Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. *Advances in Neural Information Processing Systems*, 22:1204–1212, 2009.

P Mazzoni and J W Krakauer. An implicit plan overrides an explicit strategy during visuomotor adaptation. *The Journal of neuroscience*, 26(14):3642–3645, 2006.

L McBride and K Narendra. Optimization of time-varying systems. *Automatic Control*, 1965.

R C Miall, D J Weir, D M Wolpert, and J F Stein. Is the Cerebellum a Smith Predictor? *Journal of Motor Behavior*, 1993.

M L Minsky. Computation. 1967.

I Mordatch, E Todorov, and Z Popović. Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics (TOG)*, 31(4):43, 2012.

Y Nesterov. A method of solving a convex programming problem with convergence rate O (1/k2). *Soviet Mathematics Doklady*, 1983.

A Newell. Unified theories of cognition. 1994.

D Nguyen and B Widrow. The Truck Backer-Upper: An Example of Self-Learning in Neural Networks. *Proceedings of the International Joint Conference on Neural Networks*, 1989.

R M Sapolsky. Why Zebras Don't Get Ulcers: The Acclaimed Guide to Stress, Stress-Related Diseases, and Coping-Now Revised and Updated. 2004.

M Schmidt. minFunc. *http://www.di.ens.fr/~mschmidt/Software/minFunc.html*, 2013.

R Shadmehr, M A Smith, and J W Krakauer. Error Correction, Sensory Prediction, and Adaptation in Motor Control. *Annual Review of Neuroscience*, 2010.

C E Shannon. The bandwagon. *IRE Transactions on Information Theory*, 1956.

R F Stengel. *Optimal control and estimation*. Dover Publications, 1994.

G M Stratton. Some Preliminary Experiments on Vision Without Inversion of the Retinal Image, 1896.

Y Sugita. Global plasticity in adult visual cortex following reversal of visual input. *Nature*, 1996.

D Sussillo and O Barak. Opening the black box: Low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, 2013.

I Sutskever. Training Recurrent Neural Networks. *PhD thesis, University of Toronto*, 2013.

R S Sutton and A G Barto. Reinforcement Learning: An Introduction. 1998.

Y Tassa, T Erez, and E Todorov. Fast Model Predictive Control for Reactive Robot Swimming. *http://www.cs.washington.edu/homes/todorov/papers/MPCswimmer.pdf*, 2011.

R Tedrake. LQR-Trees: Feedback motion planning on sparse randomized trees. 2009.

E Theodorou, J Buchli, and S Schaal. Reinforcement learning of motor skills in high dimensions: A path integral approach. *Robotics and Automation*, 2010.

S Thrun, M Montemerlo, and et al. Stanley: The Robot that Won the DARPA Grand Challenge. *Journal of Field Robotics*, 2006.

E Todorov. Optimality principles in sensorimotor control. *Nature neuroscience*, 2004.

E Todorov. Recurrent neural networks trained in the presence of noise give rise to mixed muscle-movement representations. *Unpublished Manuscript available from http://homes.cs.washington.edu/ todorov/papers/mixed.pdf*, 2008.

Emanuel Todorov and Michael I Jordan. Optimal feedback control as a theory of motor coordination. *Nature neuroscience*, 5(11):1226–1235, November 2002.

R Vaidyanathan, C T Chen, and C D Jeong. A reflexive vehicle control architecture based on a neural model of the cockroach escape response. In *Proceedings of the . . .* , 2012.

J M Wang, D J Fleet, and A Hertzmann. Optimizing walking controllers. *ACM Transactions on Graphics (TOG)*, 2009.

P Wawrzyński. Real-time reinforcement learning by sequential Actor–Critics and experience replay. *Neural Networks*, 2009.

P Werbos. *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches*. Van Nostrand Reinhold Company, 1992.

B Widrow, N K Gupta, and S Maitra. Punish/reward: learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 1973.

N Wiener. *Cybernetics*. Or the Control and Communication in the Animal and the Machine. The MIT Press, 1961.

R J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement Learning*, 1992.

R J Williams and D Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1989.

D M Wolpert, J Diedrichsen, and J R Flanagan. Principles of sensorimotor learning. *Nature Reviews . . .* , 2011.

Chapter 6

Appendix on Lagrange Multipliers

6.1 A Simple Derivation of Lagrange Multipliers

Suppose we are restricted to lie on a constraint surface $s(\mathbf{x}) = 0$ and would like to find the extremum of a function $C(\mathbf{x})$ on the surface. Let the point $\mathbf{x}^*$ be one such extremum of $C(\mathbf{x})$ subject to the constraint. Choose a vector $\mathbf{v} \perp \nabla_{\mathbf{x}} s(\mathbf{x}^*)$. Then $\mathbf{v} \cdot \nabla_{\mathbf{x}} s(\mathbf{x}^*) = 0$. $\mathbf{v}$ constitutes a feasible direction of movement that would not change the value of the constraint to first order. If we check the value of the cost function along the direction of $\mathbf{v}$, we have

$$C(\mathbf{x}^* + \epsilon\mathbf{v}) \quad = \quad C(\mathbf{x}^*) + \epsilon\mathbf{v} \cdot \nabla_{\mathbf{x}} C(\mathbf{x}^*) + O(\epsilon^2).$$

By the assumption that $\mathbf{x}^*$ is an extremum, $C(\mathbf{x}^* + \epsilon\mathbf{v}) = C(\mathbf{x}^*) + O(\epsilon^2)$. So $\mathbf{v} \cdot \nabla_{\mathbf{x}} C(\mathbf{x}^*) = 0$, and $\mathbf{v} \perp \nabla_{\mathbf{x}} C(\mathbf{x}^*)$. Since this holds generically for all possible feasible directions $\mathbf{v}$, it must be the case that $\nabla_{\mathbf{x}} C(\mathbf{x}^*) \propto \nabla_{\mathbf{x}} s(\mathbf{x}^*)$. We call the constant of proportionality λ and finally conclude

$$\nabla_{\mathbf{x}} C(\mathbf{x}^*) = \lambda \nabla_{\mathbf{x}} s(\mathbf{x}^*).$$

We solve such a gradient equation while leaving the constant of proportionality undetermined. Once we have arrived at a solution, we must plug the result back into the constraint to determine the particular λ that is required.