SENSORIMOTOR LEARNING

CoSMo 2013

Kingston, Ontario

DOUGLAS TWEED

1 CONTROL

Here we will study how neural networks in the brain learn to control the body. Our focus is on the brain, but we will discuss the challenges facing *any* system that pursues its aims by interacting with the world. Therefore many of the concepts will apply also to other goal-directed systems, such as robots, immune responses, and gene networks.

State Dynamics

Any control problem starts with an agent of some sort embedded in a world which obeys rules of its own that the agent has to work with and can't change. For instance, the neural networks that steer your arm can't change the geometry of the arm or the laws of physics.

These unchangeable rules are summed up in a *state equation*, which describes how a vector of interest called the *state*, $\boldsymbol{x}$, evolves under the influence of a *command* vector $\boldsymbol{u}$ issued by the agent:

$$\dot{\boldsymbol{x}}(t) = \dot{\boldsymbol{x}}(\boldsymbol{x}(t), \boldsymbol{u}(t)) \tag{1}$$

That is, $\dot{\boldsymbol{x}}(t)$, the rate of change of $\boldsymbol{x}$ at time t, depends on the current values of $\boldsymbol{x}$ and $\boldsymbol{u}$. In arm control, for example, $\boldsymbol{x}$ might be the positions and velocities of all the arm joints, and $\boldsymbol{u}$ might be the firing rates of the motoneurons to the arm muscles. So the agent's job is to issue commands, $\boldsymbol{u}$, that achieve its aims, given the rules laid out by the state equation.

More generally, $\dot{\boldsymbol{x}}$ may also depend on other things besides $\boldsymbol{x}$ and $\boldsymbol{u}$. For instance your arm dynamics change if you pick up a heavy suitcase. We will ignore that complication, but it is easy to incorporate using the principles we will learn.

We will use the word *controller*, rather than *agent*, for the thing that issues the commands $\boldsymbol{u}$, to emphasize that for us it is usually a set of neurons rather than a whole organism. The controller receives information about $\boldsymbol{x}$, and about some target state $\boldsymbol{x}^*$, and uses it to compute $\boldsymbol{u}$. For instance $\boldsymbol{x}^*$ might be a location you want to reach to. The function the controller uses to compute $\boldsymbol{u}$'s from $\boldsymbol{x}$'s and $\boldsymbol{x}^*$'s is called its *policy*. We can expand formula (1) to spell out that $\boldsymbol{u}$ depends on $\boldsymbol{x}$ and $\boldsymbol{x}^*$:

$$\dot{\boldsymbol{x}}(t) = \dot{\boldsymbol{x}}(\boldsymbol{x}(t), \boldsymbol{u}(\boldsymbol{x}(t), \boldsymbol{x}^*(t))) \tag{2}$$

In these notes I will use bold lower-case letters to represent vectors, and I will add parentheses to refer to the functions that give rise to those vectors, e.g. $\boldsymbol{u}()$ is a policy, and $\boldsymbol{u}$ and $\boldsymbol{u}(\boldsymbol{x}, \boldsymbol{x}^*)$ are commands. For brevity I will usually write $\boldsymbol{u}$ rather than $\boldsymbol{u}(\boldsymbol{x}(t), \boldsymbol{x}^*(t))$, but it is understood that $\boldsymbol{u}$ depends on $\boldsymbol{x}$ and $\boldsymbol{x}^*$, which vary with time.

Biological Dynamics

Engineers study special cases of state dynamics that are easy to control. One is the *linear* form, $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u}$, but linear systems are rare in biology. A special case that does apply to biology is the *u-affine* form,

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{G}(\boldsymbol{x})\boldsymbol{u} \tag{3}$$

This equation is widely applicable because $\boldsymbol{f}()$ and $\boldsymbol{G}()$ can be any smooth functions.

Biological dynamics often involve a *configuration*, meaning a vector $\boldsymbol{q}$ with fewer elements than the state $\boldsymbol{x}$ and with the property that $\boldsymbol{q}$ and its time-derivatives determine $\boldsymbol{x}$. For example in reaching, $\boldsymbol{q}$ is all the angles of the arm joints, and $\boldsymbol{x}$ is these angles and their derivatives, $\boldsymbol{x} = (\boldsymbol{q}^{\mathrm{T}}, \dot{\boldsymbol{q}}^{\mathrm{T}})^{\mathrm{T}}$. If we can rewrite the state dynamics this way

$$\ddot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{G}(\boldsymbol{x})\boldsymbol{u} \tag{4}$$

then we say they are in *control-canonical form*.

Simulating Dynamics

Most differential equations, including most state equations, can be solved only by numerical integration. The simplest method is Euler's, where we solve $\dot{x} = f(x)$ by breaking time into discrete instants Δt apart, and setting

$$x(t + \Delta t) = x(t) + f(x(t))\,\Delta t \tag{5}$$

We will use this method in all our control simulations, with $\Delta t = 0.01$ s.

≫ run EULER_INTEGRATION.m

Desired Dynamics

A controller needs a policy that will achieve its aims, but what are those aims? Often they are to make the state $\boldsymbol{x}$ obey some desired dynamics, e.g. to snap to a target, or track a moving object.

Desired dynamics should be feasible. They shouldn't ask the controller to whip the hand to a target supersonically, but should specify an achievable motion. Computationally, this kind of path planning isn't trivial. In the brain, suitable paths are likely wired in by natural selection, and then improved by learning. We will consider that kind of learning in Section 5.

For now we will consider an engineering approach to path planning, which is to choose *linear* desired dynamics, e.g. in the arm example we might choose

$$\ddot{\boldsymbol{q}}^* = c_1\dot{\boldsymbol{q}} + c_0(\boldsymbol{q} - \boldsymbol{q}^*) \tag{6}$$

where the *c*'s are constants. How do we choose *c*'s that give the behavior we want, e.g. driving $\boldsymbol{q}$ to $\boldsymbol{q}^*$? We define the *Hurwitz polynomial*

$$s^2 - c_1 s^1 - c_0 s^0, \text{ i.e. } s^2 - c_1 s - c_0 \tag{7}$$

and choose our *c*'s so the real parts of all the roots of this polynomial are less than 0. Then the dynamics in (6) are said to be *Hurwitz*, and can be proven to drive $\boldsymbol{q}$ to $\boldsymbol{q}^*$.

Having chosen our desired dynamics, how do we find a policy that will make them come true, i.e. make $\ddot{\boldsymbol{q}} = \ddot{\boldsymbol{q}}^*$? If the state dynamics have control-canonical form (4), and the matrix $\boldsymbol{G}(\boldsymbol{x})$ has a right-inverse, $\boldsymbol{G}(\boldsymbol{x})^+$, then a suitable policy is

$$\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{G}(\boldsymbol{x})^+[\ddot{\boldsymbol{q}}^*(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{x})] \tag{8}$$

To verify that this policy works, substitute the right-hand side for $\boldsymbol{u}$ in (4) and simplify to get $\ddot{\boldsymbol{q}} = \ddot{\boldsymbol{q}}^*$. This

policy, with a Hurwitz $\ddot{\boldsymbol{q}}^*$, is called *feedback linearization*.

≫ run FEEDBACK_LINEARIZATION.m

Policy (8) assumes the controller knows the state dynamics, as defined by $\boldsymbol{f}()$ and $\boldsymbol{G}()$. Control networks in the brain probably do have approximations to their $\boldsymbol{f}()$ and $\boldsymbol{G}()$ wired in at birth, thanks to natural selection. But as the body grows, its dynamics change. So neural controllers must *learn* $\boldsymbol{f}()$ and $\boldsymbol{G}()$.

2 ELEMENTS OF LEARNING

Neural controllers must learn their state dynamics and policies. To explore the principles in a simple setting, we will consider a task where a learner deduces a *target function* $\boldsymbol{y}()$ based on examples of $\boldsymbol{x}$'s and $\boldsymbol{y}(\boldsymbol{x})$'s. This task is called *supervised* learning; the $\boldsymbol{x}$'s are *inputs*; the $\boldsymbol{y}$'s are *desired outputs*, and each ordered pair $(\boldsymbol{x}, \boldsymbol{y})$ is an *example*. Usually the learner can't deduce $\boldsymbol{y}()$ precisely but seeks an estimate $\hat{\boldsymbol{y}}()$ that *approximates* $\boldsymbol{y}()$.

Models

Any learner has a *model*, which is the set of all the functions $\hat{\boldsymbol{y}}()$ it is willing to consider as approximations to $\boldsymbol{y}()$. For instance it might have a *linear* model, which permits only linear functions, of the form

$$\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{x} \tag{9}$$

Here $\boldsymbol{W}$ is a matrix of numbers called *parameters* or *weights*, which the controller adjusts to improve its estimates. So learning comes down to finding the $\boldsymbol{W}$ that makes $\hat{\boldsymbol{y}}()$ as close as possible to $\boldsymbol{y}()$.

More generally the model may have the form

$$\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{W}) \tag{10}$$

where $\hat{\boldsymbol{y}}()$ may be nonlinear in $\boldsymbol{x}$ or $\boldsymbol{W}$ or both. In the brain, the parameters are (probably mainly) the weights (i.e. the strengths) of the synapses.

Risk

The aim of learning is to compute the $\boldsymbol{W}$ that gives us the best approximation. But what do we mean by *best*? One approach that seems well suited to the brain is to say the best approximation is the one that minimizes the *least-squares risk*, which is the expected value of the squared error in our approximation,

$$R(\boldsymbol{W}) = \mathrm{E}(\left\|\hat{\boldsymbol{y}}(\boldsymbol{x},\boldsymbol{W}) - \boldsymbol{y}\right\|^2) \tag{11}$$

i.e. the average squared error over *all possible* examples ($\boldsymbol{x}$, $\boldsymbol{y}$).

Optimization

Learning is a matter of finding the weights $\boldsymbol{W}$ that optimize (i.e. minimize) the risk, R. How might the brain do this? Here we will look at the main methods of optimization from math and computer science. Then in Sections 3 and 4 we will apply them to neural networks. For simplicity we will focus for now on the case where $\boldsymbol{W}$ is a row vector, $\boldsymbol{w}$.

Optimization methods are classed according to the derivatives of R they use: none at all, first derivatives only, or first and second derivatives.

Derivative-free Optimization

If we can compute $R()$ but not its derivatives then the best we can do is try different $\boldsymbol{w}$'s and remember which worked best. We make an initial guess $\boldsymbol{w}$, compute $R(\boldsymbol{w})$, then try another weight $\boldsymbol{w}$' close to $\boldsymbol{w}$. If $R(\boldsymbol{w}') < R(\boldsymbol{w})$ then $\boldsymbol{w}$' becomes our new $\boldsymbol{w}$; otherwise we reject $\boldsymbol{w}$'. Then we repeat, choosing a new $\boldsymbol{w}$' near our cur-

rent $\boldsymbol{w}$ and so on. As we keep doing this, $R(\boldsymbol{w})$ gradually decreases.

Gradient Descent

If we know the first derivative of R with respect to $\boldsymbol{w}$ — written $\nabla_{\boldsymbol{w}}R$ or simply $R_{\boldsymbol{w}}$ or ∇R and called the *gradient* — then we can move toward a minimum by *gradient descent*. That is, we choose some $\boldsymbol{w}$ as a starting point, then step to

$$\boldsymbol{w}_{new} = \boldsymbol{w} - \eta \nabla R(\boldsymbol{w})^{\mathsf{T}} \tag{12}$$

(where η is some positive number), and repeat. $\nabla R(\boldsymbol{w})^{\mathsf{T}}$ points in the direction of steepest increase of R at the point $\boldsymbol{w}$, and so by stepping in the opposite direction, we *de*crease our risk.

Unfortunately this method doesn't tell us how *far* to step, i.e. how big to make η. If we choose η too small then we may approach the minimum extremely slowly. If we make η too large then we may overstep the minimum and increase our risk. Sometimes we can choose good step sizes using the second derivative of R, written $\nabla^2 R$ and called the *Hessian.*

Newton's Method

If we know the Hessian then we can often find minima quickly by *Newton's method*. In this approach we exploit the fact that any smooth function, such as R, in any small region near a minimum, is shaped roughly like the bottom of a paraboloid, i.e. like a quadratic surface $Q(\boldsymbol{w}) = a + \boldsymbol{w}\boldsymbol{b} + \boldsymbol{w}\boldsymbol{C}\boldsymbol{w}^{\mathsf{T}}$, where a is a scalar, $\boldsymbol{b}$ a column vector, and $\boldsymbol{C}$ a symmetric, positive-semidefinite matrix.

Clearly $\nabla Q(\boldsymbol{w}) = \boldsymbol{b} + 2\boldsymbol{C}\boldsymbol{w}^{\mathsf{T}}$, $\nabla^2 Q = 2\boldsymbol{C}$, and therefore

$$\nabla Q(\boldsymbol{w}) = \boldsymbol{b} + \nabla^2 Q \boldsymbol{w}^{\mathsf{T}} \tag{13}$$

We are at the minimum if ∇Q is zero. It follows that, given any $\boldsymbol{w}$, the minimum is at $\boldsymbol{w} + \Delta\boldsymbol{w}$ if

$$\left[\nabla^2 Q\right]\Delta\boldsymbol{w}^{\mathsf{T}} = -\nabla Q(\boldsymbol{w}) \tag{14}$$

because then $\nabla Q(\boldsymbol{w} + \Delta\boldsymbol{w}) = \boldsymbol{b} + \nabla^2 Q(\boldsymbol{w} + \Delta\boldsymbol{w})^{\mathsf{T}}$ [by (13)] $= \boldsymbol{b} + \nabla^2 Q\boldsymbol{w}^{\mathsf{T}} - \nabla Q(\boldsymbol{w})$ [by (14)] $= \boldsymbol{b} + \nabla^2 Q\boldsymbol{w}^{\mathsf{T}} - \boldsymbol{b} - \nabla^2 Q\boldsymbol{w}^{\mathsf{T}}$ [by (13)] $= \boldsymbol{0}$.

This formula motivates Newton's method of minimizing R, which is to choose a $\boldsymbol{w}$, step to $\boldsymbol{w}_{new} = \boldsymbol{w} + \Delta\boldsymbol{w}$, where $\Delta\boldsymbol{w}$ is a solution to

$$\left[\nabla^2 R\right]\Delta\boldsymbol{w}^{\mathsf{T}} = -\nabla R(\boldsymbol{w}) \tag{15}$$

and repeat.

If R is a perfect, concave-up paraboloid then a $\Delta\boldsymbol{w}$ fitting (15) will take us to the minimum in a single step. If R is *roughly* parabolic near $\boldsymbol{w}$ then we will need a series of steps, though fewer than with gradient descent. But if R isn't even roughly parabolic near $\boldsymbol{w}$ — if $\boldsymbol{w}$ is far from a minimum — then Newton's method may fling us into the wild blue yonder.

Another issue is that we may need a lot of effort and memory to compute the Hessian, e.g. if $\boldsymbol{w}$ has 1000 elements then $\nabla^2 R$ has a million.

Next we will see how these 3 classes of optimization methods may be implemented in the brain.

3 ONLINE LEARNING

The brain learns *online*, meaning it receives a stream of examples ($\boldsymbol{x}_s$, $\boldsymbol{y}_s$), one after another, and adjusts its weights after each one. We will assume the examples are drawn from a probability distribution (unknown to the learning algorithm). So $\boldsymbol{x}$ and $\boldsymbol{y}$ are random variables. For convenience we will assume they have means of zero.

We will start with methods that approximate the relation between $\boldsymbol{x}$ and $\boldsymbol{y}$ with linear functions, $\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{x}$. These methods are worth studying because, as we will see, they generalize to learn not just linear but any smooth function.

Again we will focus on the case where $\boldsymbol{W}$ is a row vector, $\boldsymbol{w}$. Our goal is to find the $\boldsymbol{w}$ that minimizes the risk,

$$R = \mathrm{E}(\|\boldsymbol{w}\boldsymbol{x} - y\|^2) \tag{16}$$

For now we will assume our neurons are linear: each receives a column vector of inputs $\boldsymbol{x}$ and multiplies it by a row vector of weights, $\boldsymbol{w}$. For instance if the neuron has 1000 incoming synapses then $\boldsymbol{x}$ has 1000 elements Each of the 1000 incoming signals x_i gets multiplied by the weight of its synapse, w_i, and the sum of the 1000 products is the cell's firing rate, or signal, $\hat{y} = \boldsymbol{w}\boldsymbol{x}$. We call it $\hat{y}$ because the cell is trying to estimate the desired output y.

Weight Perturbation

This is a derivative-free method. A neuron starts with a random weight vector $\boldsymbol{w}_0$, and perturbs it, trying out

nearby $\boldsymbol{w}$'s to see if they work better. As each example input $\boldsymbol{x}_s$ arrives, our neuron computes its estimate $\hat{y}_s = \boldsymbol{w}_{s-1}\boldsymbol{x}_s$ and its error $\tilde{y}_s = \hat{y}_s - y_s$. Then it chooses a small random perturbation $\boldsymbol{w}_{pert}$, computes a new estimate $\hat{y}_s' = (\boldsymbol{w}_{s-1} + \boldsymbol{w}_{pert})\boldsymbol{x}_s$ and error $\tilde{y}_s'$. If the new error is smaller than the first then the neuron accepts the perturbed weights, i.e. $\boldsymbol{w}_s = \boldsymbol{w}_{s-1} + \boldsymbol{w}_{pert}$. Otherwise it adjusts in the opposite direction, $\boldsymbol{w}_s = \boldsymbol{w}_{s-1} - \boldsymbol{w}_{pert.}$

≫ run WEIGHT_PERTURBATION.m

Node Perturbation

This is a faster derivative-free method, where the neuron perturbs its signal (its firing rate) rather than its synapses. When input $\boldsymbol{x}_s$ arrives, the neuron fires at rate $\hat{y}_s = \boldsymbol{w}_{s-1}\boldsymbol{x}_s$, and computes its error. Then it alters its firing rate slightly, to $\hat{y}_s' = \hat{y}_s + \hat{y}_{pert}$, and computes its new error. Based on those 2 errors, it computes an appropriate adjustment to $\boldsymbol{w}$. That computation makes node perturbation more complicated that weight perturbation, but faster.

≫ run NODE_PERTURBATION.m

Risk Derivatives

To apply gradient descent or Newton's method we need the derivatives of the risk R. Clearly $\nabla R = \nabla \mathrm{E}((\boldsymbol{w}\boldsymbol{x} - y)^{\mathrm{T}}(\boldsymbol{w}\boldsymbol{x} - y)) = \mathrm{E}(2\boldsymbol{x}(\boldsymbol{w}\boldsymbol{x} - y))$ [because E is linear and so we can switch E and ∇ and apply the chain rule] $= \mathrm{E}(2\boldsymbol{x}(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{w}^{\mathrm{T}} - y)) = 2\mathrm{E}(\boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{w}^{\mathrm{T}} - \boldsymbol{x}y)$, i.e.

$$\nabla R = 2[\mathrm{E}(\boldsymbol{x}\boldsymbol{x}^{\mathrm{T}})\boldsymbol{w}^{\mathrm{T}} - \mathrm{E}(\boldsymbol{x}y)] \tag{17}$$

And by similar reasoning,

$$\nabla^2 R = 2\mathrm{E}(\boldsymbol{x}\boldsymbol{x}^{\mathsf{T}}) \tag{18}$$

This matrix $\mathrm{E}(\boldsymbol{x}\boldsymbol{x}^{\mathsf{T}})$ is the *covariance* of $\boldsymbol{x}$.

The expected values in (17) and (18) are usually unknown, but we can estimate them based on the current example, $\boldsymbol{x}_s$ and y_s. To begin with, we can estimate the covariance and cross-covariance this way:

$$\hat{\mathrm{E}}(\boldsymbol{x}\boldsymbol{x}^{\mathsf{T}}) = \boldsymbol{x}_s \boldsymbol{x}_s^{\mathsf{T}}, \quad \hat{\mathrm{E}}(\boldsymbol{x}y) = \boldsymbol{x}_s y_s \tag{19}$$

And therefore

$$\hat{\nabla} R_s = 2\left(\boldsymbol{x}_s \boldsymbol{x}_s{}^{\mathsf{T}} \boldsymbol{w}^{\mathsf{T}} - \boldsymbol{x}_s y_s\right) \tag{20}$$

$$\hat{\nabla}^2 R_s = 2\boldsymbol{x}_s \boldsymbol{x}_s{}^{\mathsf{T}} \tag{21}$$

Online Gradient Descent

To find the formula for gradient descent in this online linear setting, we plug (20) into (12) to get

$$\boldsymbol{w}_s = \boldsymbol{w}_{s-1} - \eta[\boldsymbol{w}_{s-1}\boldsymbol{x}_s - y_s]\boldsymbol{x}_s^{\mathsf{T}} \tag{22}$$

In the case where the output is a vector, $\boldsymbol{y}$, rather than a scalar, the formula is the same except that $\boldsymbol{w}$ becomes an n_y-by-n_x matrix $\boldsymbol{W}$. This formula is also called *least-mean square* learning, or *LMS*.

$$\boldsymbol{W}_s = \boldsymbol{W}_{s-1} - \eta[\boldsymbol{W}_{s-1}\boldsymbol{x}_s - \boldsymbol{y}_s]\boldsymbol{x}_s^{\mathsf{T}} \tag{23}$$

For a more-compact expression, we define an error vector

$$\tilde{\boldsymbol{y}}_s = \hat{\boldsymbol{y}}_s - \boldsymbol{y}_s = \boldsymbol{W}_{s-1}\boldsymbol{x}_s - \boldsymbol{y}_s \tag{24}$$

and write LMS as

$$\Delta \boldsymbol{W} = -\eta \tilde{\boldsymbol{y}} \boldsymbol{x}^{\mathsf{T}} \tag{25}$$

LMS will drive $\tilde{\boldsymbol{y}}$ to $\mathbf{0}$, so long as η is not too large. But if η is too small then the convergence is unbearably slow. So η must be chosen carefully.

≫ run LMS.m

Online Newton

To apply Newton's method, from (15), we have to find a $\Delta \boldsymbol{w}$ such that $[\hat{\nabla}^2 R_s]\Delta \boldsymbol{w}^{\mathsf{T}} = -\hat{\nabla} R_s(\boldsymbol{w})$, i.e. $[\boldsymbol{x}_s \boldsymbol{x}_s^{\mathsf{T}}]\Delta \boldsymbol{w}^{\mathsf{T}} = \boldsymbol{x}_s \boldsymbol{x}_s^{\mathsf{T}} \boldsymbol{w}^{\mathsf{T}} - \boldsymbol{x}_s y_s$. A little algebra will confirm that $\Delta \boldsymbol{w} = -[\boldsymbol{w}\boldsymbol{x}_s - \boldsymbol{y}_s][\boldsymbol{x}_s^{\mathsf{T}} \boldsymbol{x}_s]^{-1} \boldsymbol{x}_s^{\mathsf{T}}$ does the job. Hence we have this formula for online Newton, also known as *normalized least-mean-square* learning, *NLMS*:

$$\Delta \boldsymbol{W} = -\tilde{\boldsymbol{y}} \boldsymbol{x}^{\mathsf{T}} / (\boldsymbol{x}^{\mathsf{T}} \boldsymbol{x}) \tag{26}$$

(when $\boldsymbol{x} = \mathbf{0}$ we set $\Delta \boldsymbol{W} = \mathbf{0}$). NLMS learns faster than LMS, and doesn't require us to choose an η.

≫ run NLMS.m

A drawback of NLMS is that it doesn't converge as well as LMS to a steady $\boldsymbol{W}$ in cases where only rough fits are possible. If there is a single $\boldsymbol{W}^*$ that yields a close fit to all the data then NLMS will find it, but if there is no such all-purpose $\boldsymbol{W}^*$ then NLMS will keep adjusting $\boldsymbol{W}$ forever, always minimizing the error for just the current input, never finding a $\boldsymbol{W}$ that is a good global compro-

mise. It helps to introduce a damping factor η between 0 and 1 into the learning rule, like this:

$$\Delta \boldsymbol{W} = -\eta \tilde{\boldsymbol{y}} \boldsymbol{x}^{+} \tag{27}$$

to make it less aggressive in adjusting $\boldsymbol{W}$ to suit the current input.

Recursive Least Squares

NLMS learns faster than LMS but not optimally fast. For optimal speed, we need a more complex algorithm called *recursive least squares*, RLS, also known as an online *Gaussian process*. We won't derive it here, but you can see its equations in the code file RLS.m.

≫ run RLS.m

4 LEARNING NONLINEARITIES

So far, all our methods have learned *linear* approximations, $\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{x}$, but the brain needs algorithms that work with nonlinear models

$$\hat{\boldsymbol{y}} = \hat{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{W}) \tag{28}$$

Backpropagation

The backprop algorithm trains layered networks of nonlinear neurons. Networks of this form are capable of *universal approximation*, i.e. you can approximate any smooth function as closely as you like if you have a big enough network. Backprop adjusts the network weights by gradient descent.

≫ run BACKPROPAGATION.m

Linear-in-the-parameters Learning

To learn nonlinear functions $\boldsymbol{y}(\boldsymbol{x})$, we needn't make $\hat{\boldsymbol{y}}$ nonlinear in both $\boldsymbol{x}$ and $\boldsymbol{W}$, as in (28). We can make $\hat{\boldsymbol{y}}$ nonlinear in $\boldsymbol{x}$ but *linear* in $\boldsymbol{W}$,

$$\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{\varphi}(\boldsymbol{x}) \tag{29}$$

This way, we can learn nonlinear functions of $\boldsymbol{x}$ but use linear learning algorithms. This approach is called *linear-in-the-parameters* learning.

The vector $\boldsymbol{\varphi}(\boldsymbol{x})$ in (29) is called the *feature vector*. Its elements $\varphi_i(\boldsymbol{x})$, called *features*, are functions of $\boldsymbol{x}$, usually nonlinear. Common choices are Gaussians, polyno-

mials, and hyperbolic tangents. Like layered networks, models of the form (29) are universal approximators.

By analogy with (24), our error is

$$\tilde{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{\varphi}(\boldsymbol{x}) - \boldsymbol{y} \tag{30}$$

For linear-in-the-parameters learning we simply run any of the algorithms from Section 3 using $\boldsymbol{\varphi}(\boldsymbol{x})$ in place of $\boldsymbol{x}$.

For example, suppose we want to learn the function $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}$ over the domain $x_1, x_2 \in \{-1, 1\}$, as in Backpropagation.m, but now using RLS with Gaussian features, i.e. features of the form $\varphi_i(\boldsymbol{x}) = \exp(-\gamma(\boldsymbol{x} - \boldsymbol{x}_i)^2)$. The vector $\boldsymbol{x}_i$ is called the *center* of the Gaussian. We have to guess where to put these centers, and what value of γ will work best, and how many features we need. The resulting algorithm is much faster than backprop.

≫ run RLS_GAUSSIAN.m

How might linear-in-the-parameters learning work in the brain? It could run on a layered network with linear neurons in the output layer. The upstream neurons are nonlinear, and the signals coming from the second-last layer of cells serve as features $\varphi_i(\boldsymbol{x})$. Only the synapses from this second-last layer to the output layer are adjusted by learning, because only they are linearly related to the network's output. All upstream synapses are frozen. They may be set by natural selection to yield good features $\varphi_i(\boldsymbol{x})$, but they don't change with learning, just as the widths and centers of the Gaussians don't change in RLS_Gaussian.m. So linear-in-the-parameters networks are full of synapses that may be far from optimal but can't be improved.

The Curse

With linear-in-the-parameters methods we can't improve our features during learning, so we have to choose good ones at the outset. Can we just choose a huge number of them, and hope there will be some good ones in the mix? In fact that *is* part of the method. But a brute-force approach — just choosing a lot of features randomly — has its limits, thanks to a theorem called the *curse of dimensionality*, which says the number of random features we need grows exponentially with the dimensionality of the input $\boldsymbol{x}$.

Linear versus Nonlinear

Linear-in-the-parameters methods learn faster than nonlinear ones and they may be more robust because they hinge on fewer choices. With RLS, for instance, we choose only the number and type of features, e.g. Gaussians of a certain width. But with backprop we have to choose the type of neuron, the number of layers, the number of neurons in each layer, the η's for all the neurons, and the initial values of the weights. If we choose wrong then learning fails.

On the other hand, linear-in-the-parameters networks have a lot of non-optimal synapses they can't improve whereas nonlinear methods adjust all the synapses. Nonlinear methods still suffer the curse of dimensionality — the number of adjustable parameters they need grows exponentially with the dimensions of $\boldsymbol{x}$ — but they make better use of the parameters they have.

In light of all this, the brain may mix linear and nonlinear methods — fast linear learning for output-layer synapses, and slower backprop or perturbation upstream.

5 OPTIMAL CONTROL

In optimal control we try to find desired dynamics and policies that minimize certain costs. Typically we have some non-negative scalar quantity we want to keep small, such as our rate of fuel consumption or the distance to a target, and we call that quantity the *cost rate*, r. This r may depend on $\boldsymbol{x}$, $\boldsymbol{u}$, and other things such as the location of a target, but for simplicity we will assume the target is always $\mathbf{0}$ and r is determined by $\boldsymbol{x}$ or $\boldsymbol{u}$ or both — $r(\boldsymbol{x}, \boldsymbol{u})$.

The thing we want to minimize is the *cost*, J, which is the time-integral of r,

$$J = \int_t^\infty r(\boldsymbol{x}(\tau), \boldsymbol{u}(\boldsymbol{x}(\tau)))\, d\tau \tag{31}$$

Cost-to-go

The cost (31) varies as a function of the initial state $\boldsymbol{x}(t)$. When we want to emphasize its dependence on $\boldsymbol{x}$, we call it the *cost-to-go*, and write

$$J^n(\boldsymbol{x}(t)) = \int_t^\infty r(\boldsymbol{x}(\tau), \boldsymbol{u}^n(\boldsymbol{x}(\tau)))\, d\tau \tag{32}$$

In other words $J^n(\boldsymbol{x}(t))$ is the total cost we will accumulate from now on if we are currently (i.e. at time t) in state $\boldsymbol{x}$ and we choose our commands using the policy $\boldsymbol{u}^n()$. The superscript n's emphasize that the cost-to-go depends on the policy. Any one system has many pos-

sible policies, and (32) shows the cost-to-go for one of them, $\boldsymbol{u}^n()$.

Our aim is to find the *optimal policy* $\boldsymbol{u}^*()$, which is the one with the smallest cost-to-go $J(\boldsymbol{x})$ for all $\boldsymbol{x}$ in state space. This smallest J is the *optimal cost-to-go*

$$J^*(\boldsymbol{x}(t)) = \int_t^\infty r(\boldsymbol{x}(\tau), \boldsymbol{u}^*(\boldsymbol{x}(\tau)))\, d\tau \tag{33}$$

Finding $\boldsymbol{u}^*()$ is challenging because the cost is an integral through time, and it is hard to work out how a command at any one moment will affect the cost later on. But we can simplify things with 2 concepts: the principle of optimality and Hamiltonians.

Principle of Optimality

This says that every piece of an optimal trajectory is itself optimal. For any stretch of time, t to $t + \Delta t$, and any state $\boldsymbol{x}(t)$, the optimal policy $\boldsymbol{u}^*()$ always yields a smaller value than does any other policy $\boldsymbol{u}()$ for this integral:

$$\begin{aligned} &\int_t^{t+\Delta t} \nabla J^*(\boldsymbol{x}(\tau))\, \dot{\boldsymbol{x}}(\boldsymbol{x}(\tau), \boldsymbol{u}(\boldsymbol{x}(\tau))) + r(\boldsymbol{x}(\tau), \boldsymbol{u}(\boldsymbol{x}(\tau)))\, d\tau \\ &= \int_t^{t+\Delta t} \nabla J^* \dot{\boldsymbol{x}} + r\, d\tau \\ &= \int_t^{t+\Delta t} \dot{J}^*\, d\tau + \int_t^{t+\Delta t} r\, d\tau \end{aligned} \tag{34}$$

To see why, note that in the last line of (34), the first integral is the decrease in J^* achieved by the policy $\boldsymbol{u}()$ between times t and $t + \Delta t$, and the second integral is the price paid — the cost that $\boldsymbol{u}()$ accumulated to get this reduction in J^*. Clearly, if $\boldsymbol{u}()$ achieved a smaller sum than $\boldsymbol{u}^*()$, then we could use $\boldsymbol{u}()$ from time t to Δt, and $\boldsymbol{u}^*()$ thereafter, and so accumulate less cost from

time t to ∞ than we would with $\boldsymbol{u}^*()$ alone. But this is a contradiction because $\boldsymbol{u}^*()$ is optimal. Hence $\boldsymbol{u}^*()$ minimizes (34).

Hamiltonians

As we have seen, $\boldsymbol{u}^*()$ minimizes the integral (34) over any time interval Δt, no matter how brief. That means it must minimize the integrand at each moment, i.e. the optimal command $\boldsymbol{u}^*$ is the one that minimizes

$$\nabla J^*(\boldsymbol{x})\ \dot{\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{u}) + r(\boldsymbol{x},\boldsymbol{u}) \tag{35}$$

The quantity (35) is called the *Hamiltonian* of the system, $H^*(\boldsymbol{x}, \boldsymbol{u})$.

Moreover, differentiating (33) yields $\dot{J}^* = -r$ (where the minus sign appears because we differentiate with respect to the left rather than the right side of the integration domain $[t, \infty)$). Adding r to both sides and expanding terms, we get

$$\nabla J^*(\boldsymbol{x})\ \dot{\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{u}^*(\boldsymbol{x})) + r(\boldsymbol{x},\boldsymbol{u}^*(\boldsymbol{x})) = 0 \tag{36}$$

i.e. $H^*(\boldsymbol{x}, \boldsymbol{u}^*) = 0$. From (35) and (36) we see that $\boldsymbol{u}^*$ both minimizes and zeroes the Hamiltonian. So for all $\boldsymbol{x}$,

$$\min_{\boldsymbol{u}\in U}\{\nabla J^*(\boldsymbol{x})\ \dot{\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{u}) + r(\boldsymbol{x},\boldsymbol{u})\} = 0 \tag{37}$$

where U is the set of admissible commands.

Finally, it can be shown that no other function can substitute for $J^*()$ in (37) and still make it true; i.e. $J^*()$ is the *unique* solution to

$$\min_{\boldsymbol{u}\in U}\{\nabla J(\boldsymbol{x})\ \dot{\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{u}) + r(\boldsymbol{x},\boldsymbol{u})\} = 0 \tag{38}$$

which is called the *Hamilton-Jacobi-Bellman equation*.

So if we can find the $J()$ that solves the HJB equation then we have the optimal cost-to-go $J^*()$, and we get $\boldsymbol{u}^*()$ as well because it is defined by (37). Unfortunately, in most problems the HJB equation can't be solved analytically. We have to look for *approximate* solutions.

Solving HJB Equations by Learning

There are many approaches, but most are versions of the following scheme. First we choose a policy $\boldsymbol{u}()$ and a scalar function $\hat{J}()$ to be our initial estimates of $\boldsymbol{u}^*()$ and $J^*()$. This initial $\boldsymbol{u}()$ may be far from $\boldsymbol{u}^*()$, but must at least have a finite cost-to-go. Often a suitable $\boldsymbol{u}()$ can be found by feedback linearization.

This $\boldsymbol{u}()$ and $\hat{J}()$ imply an estimate of the Hamiltonian,

$$\hat{H}(\boldsymbol{x},\boldsymbol{u}) = \nabla\hat{J}(\boldsymbol{x})\ \dot{\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{u}(\boldsymbol{x})) + r(\boldsymbol{x},\boldsymbol{u}(\boldsymbol{x})) \qquad (39)$$

Our $\boldsymbol{u}()$, $\hat{J}()$ and $\hat{H}()$ won't have the properties of the real $\boldsymbol{u}^*()$, $J^*()$ and $H^*()$ as laid out in (37), i.e. our $\boldsymbol{u}()$ won't yield commands $\boldsymbol{u}$ that minimize $\hat{H}(\boldsymbol{x},\boldsymbol{u})$, and the minimum of $\hat{H}(\boldsymbol{x},\boldsymbol{u})$ won't be 0 for all $\boldsymbol{x}$. But we can gradually adjust $\boldsymbol{u}()$ and $\hat{J}()$ to get closer and closer to those properties. That is, we alternate 2 operations: Step 1, adjust $\hat{J}()$ to bring $\hat{H}$ closer to 0, i.e. to shrink $\hat{H}^2$; Step 2, adjust $\boldsymbol{u}()$ to lower $\hat{H}$.

For example, we may choose an estimate $\hat{J}(\boldsymbol{x})$ of the form $\boldsymbol{w}\boldsymbol{\varphi}(\boldsymbol{x})$ where $\boldsymbol{w}$ is a row-vector of weights and $\boldsymbol{\varphi}(\boldsymbol{x})$ is a feature vector. Then

$$\hat{H} = \boldsymbol{w}\nabla\varphi\dot{\boldsymbol{x}} + r = w\dot{\varphi} + r \qquad (40)$$

and

$$\partial \hat{H}^2 / \partial \boldsymbol{w} = 2\hat{H}\dot{\phi} \qquad (41)$$

So in Step 1 we can adjust $\boldsymbol{w}$ to shrink $\hat{H}^2$ by the LMS formula, $\Delta \boldsymbol{w} = -\eta \hat{H}\dot{\phi}^{\mathrm{T}}$. Similarly, we choose a parametrized $\boldsymbol{u}() \text{ — } \boldsymbol{u}(\boldsymbol{x}, \boldsymbol{W})$ — so in Step 2 we can adjust $\boldsymbol{W}$ to lower $\hat{H}$ by gradient descent, $\Delta \boldsymbol{W} = -\eta_u [\partial \hat{H} / \partial \boldsymbol{W}]^{\mathrm{T}}$. There is an example in the code file HG_TO_1DOF.m.

≫ run HG_TO_1DOF.m