THEORETICAL PHYSIOLOGY

PSL432

University of Toronto

DOUGLAS TWEED

Course outline

This course deals with the mathematical principles of control and learning in the sensorimotor systems of the brain.

Course director:	Douglas Tweed MSB 3221 Office hours by appointment douglas.tweed@utoronto.ca 416-912-7497
Teaching assistant:	Ali Mojdeh Office hours by appointment ali.mojdeh@mail.utoronto.ca
Lectures:	Tuesdays 2–4 p.m. MSB 3227 Wifi 12345678

Materials

There is no text book. All topics will be covered in class and summarized in these handout notes and computer code posted online. Please read the assigned sections of the notes for each class before the class begins, and come prepared to ask questions about the material, and to help answer other students' questions.

Evaluation

There will be no tests or exams. All course marks will be based on 3 take-home assignments, worth 25%, 40%, and 35%. The assignments will be posted on Jan 28, Feb 25, and Mar 17, and will be due 2 weeks later, i.e. one minute before midnight on Feb 11, Mar 10, and Mar 31. The penalty for late submission will be 5% (of the total marks for that assignment) per day late. Assignments may include calculations and proofs, but will mainly involve writing computer simulations of sensorimotor control and learning.

Simulations

You will write all your simulations in the Matlab programming language. You can get Student Matlab for free at the website https://www.mathworks.com/academia/ tah-portal/university-of-toronto-676468.html. Any version of Matlab from 2017 on will do, and you need *only* Matlab, not Simulink or any toolboxes.

Starting on Jan 14, each class will end with a simulation workshop, where you can apply the day's new concepts on your own computer. Please bring a laptop with Matlab installed on it so you can take part.

Schedule

Date	Topic
Jan 7	Control Systems
Jan 14	Control Systems
Jan 21	Control Systems
Jan 28*	Supervised Learning
Feb 4	Supervised Learning
Feb 11	Supervised Learning
Feb 18	*Reading week*
Feb 25*	Sensory Maps
Mar 3	Reinforcement Learning
Mar 10	Reinforcement Learning
Mar 17*	Reinforcement Learning
Mar 24	Regularization
Mar 31	Regularization

* Days when problem sets will be assigned. You will have 2 weeks for each, i.e. they will be due one minute before midnight on Feb 11, Mar 10, and Mar 31.

1 Control systems

In this course we will study how sensorimotor systems in the brain learn to control the body. The mathematical principles of learning and control have become clearer in recent years thanks to research in artificial intelligence and robotics, so we will borrow concepts from these disciplines to help us understand the brain, and conversely, we will consider how ideas from neuroscience may benefit artificial systems.

State dynamics

Any sensorimotor task starts with an *agent* of some sort embedded in an *environment*. We write $\boldsymbol{s}_t$ for the *state* of the environment at time t, and $\boldsymbol{a}_t$ for the agent's *action* at this time (where both $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$ are represented mathematically by column vectors).

For example, the agent might be the saccadic system in the brain, which rotates the eyes quickly to aim them at an object of interest. In that case $\boldsymbol{s}_t$ might include the current positions of the eyes and the location of the visual target. And $\boldsymbol{a}_t$ might be the motor command — the firing rates of the motoneurons to the muscles that move the eyes.

In this course, we assume that time passes in discrete steps of some fixed size Δt, because that approach is convenient for computer simulations. With each step, the state takes a new value,

(1) $$\boldsymbol{s}_{t+\Delta t} = \boldsymbol{s}_t + \Delta \boldsymbol{s}_t$$

The state-change, $\Delta \boldsymbol{s}_t$, depends on the current $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$ according to rules called the *environment* (or *state*) *dynamics*. Those dynamics may be stochastic, and so in general we describe them with probabilities,

$$p(\Delta \boldsymbol{s}_t \mid \boldsymbol{s}_t, \boldsymbol{a}_t) \tag{2}$$

That is, the conditional probability that the state-change will take the value $\Delta \boldsymbol{s}_t$, given the current $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$, is set by the distribution p.

We will assume the state dynamics are *Markov*, meaning that these conditional probabilities would not change if we knew any or all *earlier* states and actions, in addition to the current ones, $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$. That is, the history before time t contains no further information that would let us predict $\Delta \boldsymbol{s}_t$ better than we can with just $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$.

We define $\boldsymbol{f}$, the *state-dynamics function*, to be the average of p, divided by Δt,

$$\mathrm{E}[\Delta \boldsymbol{s}_t] = \Delta t\, \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{3}$$

In the special case where the dynamics are deterministic, we omit the expectation:

$$\boldsymbol{s}_{t+\Delta t} = \boldsymbol{s}_t + \Delta \boldsymbol{s}_t = \boldsymbol{s}_t + \Delta t\, \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t) \tag{4}$$

Simulating dynamics

In our equations, time passes in discrete steps. But real time is continuous, and real processes obey *differential* equations, such as $\mathrm{d}\boldsymbol{s}/\mathrm{d}t = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{a})$. We can approximate such systems in our discrete-time framework using *Euler integration*, where we replace $\mathrm{d}\boldsymbol{s}/\mathrm{d}t$ by $\Delta \boldsymbol{s}_t / \Delta t$, as in

(3) or (4). This approximation will usually be more accurate when Δt is smaller, but small Δt's are computationally expensive, because it takes a lot of them to cover, say, 1 s of time. In this course, a good compromise will be to choose Δt in the range 0.1 to 0.001 s. You can find an example in the code file Euler_integration.m, posted online (and see *Getting Started with Matlab*, near the end of this chapter).

›› run EULER_INTEGRATION.M

Policy

The agent's job is to choose actions, $\boldsymbol{a}$, that achieve its aims, given the rules laid out by the state dynamics. In this course we will usually assume that the agent, based on its sensory inputs, memories, and intentions, knows $\boldsymbol{s}_t$ in full at every time t. That is, we consider situations where *state feedback* is available, not ones where the state is only partially observable.

The function the agent uses to compute $\boldsymbol{a}$'s from $\boldsymbol{s}$'s is called its *policy*, $\boldsymbol{\mu}$,

(5) $$\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t)$$

Rewards

An agent needs a policy that will achieve its aims, but what are these aims? We assume the agent receives or computes, at each time step, a signal called a *reward*, r_t, usually determined by the current state and action,

(6) $$r_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t)$$

A high reward indicates something pleasant, such as food, warmth, a stabilized retinal image, or having some visual target perfectly centered on the fovea, whereas a lower reward is less pleasant. The reward at any moment is a *scalar*: it may be computed from the vectors $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$, but it expresses the overall pleasantness of the situation in a single number. The agent's job is to choose actions that yield high rewards.

For example, suppose the agent wants to hold $\boldsymbol{s}$ near a target state $\boldsymbol{s} = \mathbf{0}$. We can express that wish mathematically by defining a reward that is maximal when $\boldsymbol{s} = \mathbf{0}$, and is otherwise much lower, such as

(7) $$r_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = -\tanh(\boldsymbol{s}_t^{\mathsf{T}} \boldsymbol{B}\, \boldsymbol{s}_t)$$

where $\boldsymbol{B}$ is a diagonal matrix of non-negative elements. This r_t attains its maximum, namely 0, when $\boldsymbol{s}$ equals its target value of $\mathbf{0}$. But if $\boldsymbol{s}$ is even a little off target (and the elements of $\boldsymbol{B}$ are at least fairly large) then r_t is close to its minimum of –1.

It may seem odd that the reward (7) is defined with a negative sign and has a maximal value of 0 — i.e. the best reward we can ever get is *nothing*. But we will see that rewards of this type are common in sensorimotor systems — many such systems try to eliminate some bad thing, such as being off-target, and their highest reward is to zero that bad thing.

Equation (7) also shows that r_t doesn't always depend on both $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$, but I have written $r(\boldsymbol{s}_t, \boldsymbol{a}_t)$ nonetheless because in general, rewards do depend on both.

Returns

I have introduced the idea of rewards as a step toward expressing aims in a mathematically precise way, but many practical aims can't be defined simply in terms of maximizing a reward at each moment. Instead, most purposive behavior is naturally expressed in terms of maximizing the *time-integral* of a reward, called the *return*, R.

For example, suppose we want to drive a mechanical system such as an arm or leg to a target state as quickly as possible and hold it there — a task called *time-optimal* control. If the target state is $\mathbf{0}$ then (7) is an apt reward function, but we should *not* aim to maximize this reward at each moment. If we *do*, we will drive $\boldsymbol{s}$ to $\mathbf{0}$ at high speed, so as to raise r_t quickly, but then we will have so much momentum that we will overshoot the target, and r_t will plummet. Essentially, if we maximize r_t at each moment then we won't hit the brakes in a timely fashion. Policies of this type, which maximize immediate reward at each instant with no regard for future consequences, are called *myopic*.

So we don't want a myopic policy that maximizes immediate reward, we want a policy that cares about the future, and so tries to maximize the discrete-time integral of the reward from the present moment, $t = 0$, to some future time T,

$$R = \Delta t \, \Sigma_{t=0 \ldots T} \, r_t \tag{8}$$

where T may be ∞. This integral formulation accurately expresses the aim of time-optimal control because clearly the way to maximize R is to get $\boldsymbol{s}$ to $\mathbf{0}$ as quickly as possible and hold it there.

Vestibulo-ocular reflex

An example of a sensorimotor system is the *vestibulo-ocular reflex*, or *VOR*, which senses head motion and counterrotates the eyes to keep the retinal images stable. Without your VOR, your vision would blur whenever you moved, like a photo taken with a moving camera.

For simplicity we will consider just one eye, rotating only horizontally, in which case the state $\boldsymbol{s}_t$ consists of horizontal eye position $\boldsymbol{q}_t$ and current horizontal head velocity $\boldsymbol{h}_t$, the latter measured by sensors called semi-circular canals located in the inner ears, in cavities in the skull bones called the *vestibula*. We write

(9) $$\boldsymbol{s}_t = [\boldsymbol{q}_t; \boldsymbol{h}_t]$$

where the semicolon means we stack $\boldsymbol{q}_t$ and $\boldsymbol{h}_t$ vertically, one on top of the other, to form a *column* vector $\boldsymbol{s}_t$.

The variables $\boldsymbol{q}_t$ and $\boldsymbol{h}_t$ have just one element each, and so could be regarded as scalars (i.e. numbers), but we will view them as one-element vectors and write them in vector notation, in boldface, because in most systems, the position and input variables are vectorial.

Throughout the course, we will write $\boldsymbol{q}$ for the *configuration* of a system, i.e. its position, location, or posture, as opposed to its velocity, acceleration, etc. The elements of $\boldsymbol{q}$ are the system's *degrees of freedom*.

What are the state dynamics of the VOR? Eye position $\boldsymbol{q}$ evolves (to a good approximation) deterministically:

(10) $$\Delta\boldsymbol{q}_t = \Delta t\,(\boldsymbol{a}_t - \kappa\boldsymbol{q}_t) / \rho$$

Here κ and ρ are constants that reflect the mechanical properties — stiffness and viscosity — of the eye muscles. Realistic values are $\kappa = 250$ and $\rho = 50$, if t is measured in seconds, $\boldsymbol{q}$ in radians, and $\boldsymbol{a}$ in mean spike rate per motoneuron per second.

As for head velocity $\boldsymbol{h}_t$, we will measure it in radians/s and treat it as something random, which the agent cannot predict at all, but must simply read in from its sensors. In reality you *can* predict your head motion to some extent, as it is partly under your control, but it may also be affected by unpredictable factors, as when you are riding a bucking bronco. For now, it is simpler to treat $\boldsymbol{h}_t$ as completely unpredictable.

What are the aims of the VOR? It tries to *minimize* the velocity with which the visual image is slipping across the retina, so we can define its reward as the *negative* of the squared slip velocity,

$$(11) \qquad r_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t) = -\left\| (\Delta \boldsymbol{q}_t / \Delta t) + \boldsymbol{h}_t \right\|^2$$

(though other choices, such as the negative of the *non*-squared magnitude of the slip velocity, would also be reasonable).

The VOR's policy is a neural network in the brainstem and cerebellum. It receives information about $\boldsymbol{s}_t$ and uses it to compute an action $\boldsymbol{a}_t$, which is another one-element vector, the mean firing of the motoneurons. (Or more precisely, it is the difference between the mean firing rates of the motoneurons to the rightward and leftward-pulling eye muscles.)

What is the best policy? Using (10), we see that r_t can be maximized (that is, zeroed) if

(12) $$\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t) = \kappa \boldsymbol{q}_t - \rho \boldsymbol{h}_t$$

To apply this policy, the agent must know not just $\boldsymbol{s}_t$ but also the stiffness and viscosity coefficients of the muscles, κ and ρ. No sensory signals can report those values in any straightforward way, but by observing eye motions evoked by a wide variety of actions, $\boldsymbol{a}$, the agent can gradually deduce them. In other words, the agent can learn estimates of κ and ρ.

To emphasize that it uses approximations rather than exact, true values, we write the policy this way,

(13) $$\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t) = \langle \kappa \rangle \boldsymbol{q}_t - \langle \rho \rangle \boldsymbol{h}_t$$

where corner brackets indicate estimates. In any system with estimates of environment properties, such as $\langle \kappa \rangle$ and $\langle \rho \rangle$, we call the set of estimates a *model* of the environment.

›› run VOR.m

Saccades

We have deduced the ideal policy for the VOR, but for most sensorimotor systems it is harder to choose optimal actions. To see some of the issues, we will examine the system that drives *saccades*, the rapid eye movements that rotate the gaze lines to a visual target. Again we consider horizontal movements of one eye.

For this task, the state is

(14) $$\boldsymbol{s}_t = [\boldsymbol{q}_t; \boldsymbol{q}^*_t]$$

where $\boldsymbol{q}_t$ is horizontal eye position as in (9), and $\boldsymbol{q}^*_t$ is *desired* horizontal eye position, the position that will point the fovea at the visual target. Eye position $\boldsymbol{q}$ evolves according to (10), as it did in the VOR, and with the same κ and ρ, because both systems use the same eyeballs and muscles.

Target position $\boldsymbol{q}^*$ we treat as random. In reality, the saccadic system must analyze the retinal images to detect objects and choose one as the visual target, but for now we will assume the choice has already been made in circuits upstream, and the saccadic system receives a one-element vector $\boldsymbol{q}^*_t$.

For this first look at saccades, we will take as our reward the negative squared difference between the actual and desired eye positions,

$$r_t = -\left\| \boldsymbol{q}_t - \boldsymbol{q}^*_t \right\|^2 \tag{15}$$

(In fact, slightly more complicated rewards are more realistic for saccades, but (15) will do for now.)

The policy is implemented by neurons mainly in the midbrain and brainstem. We assume these neurons always know the current $\boldsymbol{q}_t$ and $\boldsymbol{q}^*_t$. But because time passes in discrete steps, they have to wait Δt time units before they learn the *next* state and select the next action. In other words, our discrete Δt mirrors the fact that real agents, even in continuous time, receive their feedback with a *delay*. For the saccadic system we set Δt = 0.01 s because that much delay is biologically reasonable (though values as small as 0.001 s may also be realistic).

Another issue, crucial for saccades, is that actions are *bounded*. Motoneurons can't fire billions of spikes per second, to snap the eye to its target in a nanosecond.

Real motoneurons can't exceed about 500 spikes/s, and most saccades take 20–100 ms.

So what is the best policy for saccades? We might try to find a policy by solving for the action $\boldsymbol{a}_t$ that zeroes r_t, as we did with the VOR, but we won't succeed, because nothing in (15) — the formula for r_t — is a function of $\boldsymbol{a}_t$. Specifically, (10) tells us that $\boldsymbol{a}_t$ (together with other variables) determines $\Delta\boldsymbol{q}_t$, not $\boldsymbol{q}_t$.

We might think of solving for the $\boldsymbol{a}_t$ that produces a $\Delta\boldsymbol{q}_t$ such that $\boldsymbol{q}_{t+\Delta t} = \boldsymbol{q}^*_t$. But that won't work either: because $\boldsymbol{a}_t$ is bounded, it is almost never possible to make $\boldsymbol{q}_{t+\Delta t} = \boldsymbol{q}^*_t$, i.e. to reach the target in a single 0.01-s time step.

Another approach is to define some sensible *desired dynamics*. To keep the formulas compact, we will write $\boldsymbol{q}^{(1)}_t$ for the discrete-time *velocity* $\Delta\boldsymbol{q}_t/\Delta t$, $\boldsymbol{q}^{(2)}_t$ for the *acceleration* $\Delta\boldsymbol{q}^{(1)}_t/\Delta t$, $\boldsymbol{q}^{(3)}_t$ for the *jerk* $\Delta\boldsymbol{q}^{(2)}_t/\Delta t$, and so on. And for $\boldsymbol{q}_t$ we may also write $\boldsymbol{q}^{(0)}_t$.

Using this notation, we might define a desired eye velocity for saccade trajectories,

(16) $$\boldsymbol{q}^{(1)*}_t = \alpha\,(\boldsymbol{q}_t - \boldsymbol{q}^*_t)$$

where α is a negative constant. Then we can solve (10) for the $\boldsymbol{a}_t$ that makes $\boldsymbol{q}^{(1)}_t = \boldsymbol{q}^{(1)*}_t$, namely

(17) $$\boldsymbol{a}_t = \langle\kappa\rangle\boldsymbol{q}_t + \langle\rho\rangle\alpha(\boldsymbol{q}_t - \boldsymbol{q}_t^*)$$

This policy will drive the eye to its target, zeroing the reward (15), so long as $\langle\kappa\rangle$ and $\langle\rho\rangle$ are accurate and α is not so large that it causes overshoot, as you can check in the posted code file.

```
›› run SACCADIC_LINEAR.m
```

But there are many possible choices of desired dynamics, and (16) is not particularly good. For one thing, saccades should be fast, but (16) is slow. It makes $\boldsymbol{q}^{(1)*}{}_t$ a *linear* function of the *motor error,* $\boldsymbol{q}_t - \boldsymbol{q}_t^*$, which means that as the error shrinks, the velocity shrinks proportionally. If initial error is, say, 0.4 radians, then the eye may start toward the target at high speed, but a little later, when the error has shrunk to 0.2, velocity will be halved.

Better to choose desired dynamics that maintain speed until the target is reached, such as

(18) $$\boldsymbol{q}^{(1)*}{}_t = \alpha\,\mathrm{sgn}(\boldsymbol{q}_t - \boldsymbol{q}_t^*)$$

where the sgn function (pronounced *signum*) yields 1 when its argument is positive, –1 when it is negative, and 0 when it is 0. The resulting policy is

(19) $$\boldsymbol{a}_t = \langle\kappa\rangle\boldsymbol{q}_t + \langle\rho\rangle\alpha\,\mathrm{sgn}(\boldsymbol{q}_t - \boldsymbol{q}_t^*)$$

This policy is a simple case of *sliding-mode* control, which we will define in greater generality later.

›› run SACCADIC_SLIDING.m

As you explore this code, you will find that (19) doesn't actually work very well — it causes *chatter*, or rapid oscillations in $\boldsymbol{q}$ and $\boldsymbol{a}$ (*why?*) — but the following softened version is better,

(20) $$\boldsymbol{a}_t = \langle\kappa\rangle\boldsymbol{q}_t + \langle\rho\rangle\alpha\tanh(\beta(\boldsymbol{q}_t - \boldsymbol{q}_t^*))$$

In fact, (20) is close to what the real saccadic system does, except that the real system uses its $\langle\kappa\rangle$ and $\langle\rho\rangle$ not only as in (20), but also to update an internal estimate of

$\boldsymbol{q}_t$, by running a kind of neural simulation of the state dynamics (10):

$$(21) \qquad \langle \boldsymbol{q}_{t+\Delta t} \rangle = \langle \boldsymbol{q}_t \rangle + \Delta t\, (\boldsymbol{a}_t - \langle \kappa \rangle \langle \boldsymbol{q}_t \rangle) / \langle \rho \rangle$$

It then uses this running estimate instead of the real $\boldsymbol{q}_t$ in its policy,

$$(22) \qquad \boldsymbol{a}_t = \langle \kappa \rangle \langle \boldsymbol{q}_t \rangle + \langle \rho \rangle \alpha \tanh(\beta (\langle \boldsymbol{q}_t \rangle - \boldsymbol{q}_t^*))$$

This way, the policy neurons get (approximate) information about eye position faster than if they waited for visual or proprioceptive signals. This mechanism is called *internal feedback* because the agent learns $\boldsymbol{q}_t$ from its own model rather than from the senses. Internal feedback is what allows the saccadic system to achieve a feedback delay Δt as small as 0.01. Visual feedback takes longer — about 0.1 s.

›› run SACCADIC_INTERNAL.m

Higher-order dynamics

The VOR and saccades both have (to a good approximation) *first-order* state dynamics, meaning that no time-derivatives higher than the first — velocity — appear in the dynamics. But most sensorimotor systems are at least second-order.

To see the implications, we will start not with any real sensorimotor dynamics but with the simplest second-order system, namely a frictionless bead sliding on a straight, horizontal wire, driven by a varying force which we will regard as an action $\boldsymbol{a}$. The bead's location $\boldsymbol{q}$ obeys Newton's second law, $d^2\boldsymbol{q}/dt^2 = \boldsymbol{a}$ (assuming the bead has mass 1 kg, and force $\boldsymbol{a}$ is measured in newtons). The state must now include both position and velocity, because both contribute to the future evolution of the system,

(23) $$\boldsymbol{s}_t = [\boldsymbol{q}_t; \boldsymbol{q}^{(1)}{}_t]$$

and the state dynamics (in discrete time) are

(24) $$\Delta\boldsymbol{s}_t = [\Delta\boldsymbol{q}_t; \Delta\boldsymbol{q}^{(1)}{}_t] = \Delta t\,[\boldsymbol{q}^{(1)}{}_t; \boldsymbol{a}_t]$$

Notice that $\boldsymbol{a}_t$ has no effect on $\Delta\boldsymbol{q}_t$, and therefore none on $\boldsymbol{q}_{t+\Delta t}$. It may affect $\boldsymbol{q}_{t+2\Delta t}$ and later $\boldsymbol{q}$'s, but $\boldsymbol{q}_{t+\Delta t}$ is already determined by the current velocity $\boldsymbol{q}^{(1)}{}_t$. In other words, the system has *inertia*, and this fact can complicate control.

Suppose we want to drive our bead-on-a-wire to state $\boldsymbol{s} = \mathbf{0}$, assuming $|\boldsymbol{a}| \le 5$, and taking (7) as our reward. As with saccades, no $\boldsymbol{a}_t$ will zero this reward immediately, but we can choose desired dynamics that take $\boldsymbol{s}$ to $\mathbf{0}$ eventually.

For saccades, it was easy to choose reasonable linear dynamics, because the state dynamics were first-order. For an *n*th-order system, how do we choose constant coefficients α_i so the linear dynamics,

(25) $$\boldsymbol{q}^{(n)*} = \alpha_{n-1}\boldsymbol{q}^{(n-1)} + \ldots + \alpha_1\boldsymbol{q}^{(1)} + \alpha_0\boldsymbol{q}^{(0)}$$

give the behavior we want, e.g. driving $\boldsymbol{q}$ to $\mathbf{0}$? We define the *Hurwitz polynomial*

(26) $$x^n - \alpha_{n-1}x^{n-1} - \ldots - \alpha_1 x^1 - \alpha_0$$

(replacing the time-derivatives $\boldsymbol{q}^{(i)}$ from (25) with powers x^i of a new variable x), and choose our α's so this polynomial's roots all have real parts less than 0. Then if we use these α's in (25), we get what are called *Hurwitz* dynamics, which (it can be shown) drive $\boldsymbol{s}$ to $\mathbf{0}$, so long as Δt is not too large.

If our goal is not $\mathbf{0}$ then we make appropriate adjustments, e.g. if we want to take $\boldsymbol{q}$ to some nonzero stationary target $\boldsymbol{q}^*$ then we set

$$(27) \quad \boldsymbol{q}^{(n)}{*} = \alpha_{n-1}\boldsymbol{q}^{(n-1)} + \ldots + \alpha_1\boldsymbol{q}^{(1)} + \alpha_0(\boldsymbol{q}^{(0)} - \boldsymbol{q}^*)$$

For the bead on the wire, we might choose the Hurwitz dynamics

$$(28) \quad \boldsymbol{q}^{(2)}{*} = \alpha_1\boldsymbol{q}^{(1)} + \alpha_0\boldsymbol{q}^{(0)} = -6\boldsymbol{q}^{(1)} - 9\boldsymbol{q}$$

which imply the policy

$$(29) \quad \boldsymbol{a}_t = -6\boldsymbol{q}^{(1)}{}_t - 9\boldsymbol{q}_t$$

Defining a policy this way, based on Hurwitz dynamics, is called *feedback linearization*.

But as with saccades, linear dynamics are slow. We can speed things up using *sliding-mode* control, meaning we choose Hurwitz dynamics for $\boldsymbol{q}^{(n-1)}$, but faster, nonlinear dynamics for $\boldsymbol{q}^{(n)}$. For example, given a stationary target $\boldsymbol{q}^*$, we set

$$(30) \quad \boldsymbol{q}^{(n-1)}{*} = \alpha_{n-2}\boldsymbol{q}^{(n-2)} + \ldots + \alpha_1\boldsymbol{q}^{(1)} + \alpha_0(\boldsymbol{q}^{(0)} - \boldsymbol{q}^*)$$

and

$$(31) \quad \boldsymbol{q}^{(n)}{*} = \alpha_{n-1}\,\mathrm{sgn}(\boldsymbol{q}^{(n-1)} - \boldsymbol{q}^{(n-1)}{*})$$

Here α_0 to α_{n-2} are chosen to make (30) Hurwitz, and α_{n-1} is a negative number. Given (31), $\boldsymbol{q}^{(n-1)}$ moves rapidly toward $\boldsymbol{q}^{(n-1)}{*}$, reaching it quickly, and thereafter the state dynamics obey (30), with the result that $\boldsymbol{q} \to \boldsymbol{q}^*$ exponentially.

For the bead on the wire, we might choose

(32) $$\boldsymbol{q}^{(1)*} = \alpha_0(\boldsymbol{q} - \boldsymbol{q}^*) = -6\boldsymbol{q}$$

and

(33) $$\boldsymbol{q}^{(2)*} = \alpha_1 \operatorname{sgn}(\boldsymbol{q}^{(1)} - \boldsymbol{q}^{(1)*}) = -5\operatorname{sgn}(\boldsymbol{q}^{(1)} + 6\boldsymbol{q})$$

which imply the policy

(34) $$\boldsymbol{a}_t = -5\operatorname{sgn}(\boldsymbol{q}^{(1)}{}_t + 6\boldsymbol{q}_t)$$

But even (34) is not the time-optimal policy — the one that gets $\boldsymbol{s}$ to **0** as quickly as possible, maximizing return (8) given reward (7). In this simple case of a frictionless bead on a wire, we can deduce the time-optimal policy analytically (see posted code), but in general, time-optimal control is hard.

```
›› run INERTIAL_TIME_OPT.m
```

Learning

We have managed to find sensible policies for the VOR, saccades, and inertial control. But we did it by examining equations and reasoning about them. How can such policies be acquired by sensorimotor networks in the brain? They can't be hardwired in by the genome, because they depend on mechanical coefficients like κ and ρ, which change as the body grows. They must be at least partly learned.

Also, we found our policies by *specific* reasoning, tailored to the VOR, saccades, and a bead on a wire. And in fact our reasoning worked only because those tasks were very simple. The brain has more likely evolved one or a few *general* mechanisms that can learn policies for a wide range of difficult tasks. In this course we will explore how such mechanisms can be embodied in the neural networks of the brain.

Getting started with Matlab

In Matlab, you write code (m-files) in the *editor window*, and when you run your code, you may see printed outputs and error messages in the *command window*. You can also use the command window as a calculator (e.g. type `5 + 7` and get `12`), and you can write and run one-line programs there.

The following words or key-presses are useful in the command window.

`Ctrl-C` will halt a running piece of Matlab code.

`help` will provide information about Matlab functions, e.g. `help rand` will reveal how to use Matlab's random-number function, `rand`.

If you don't know the name of the function you want, then write `lookfor`; e.g. `lookfor random` will bring up a lot of documentation where the word "random" appears.

If there is a bug in your code then error messages will appear in the command window.

In the editor window, omitting the semicolon `;` after a statement will cause Matlab to display its output in the command window, which may help you debug your code, e.g. if you write `c = a + b`, without a semicolon, then the value of `c` will appear in the command window.

Start a line with % to make it a comment (i.e. ignored by the computer but useful for humans reading your code).

If a variable takes the value `NaN` (not a number) or `inf` (infinite) then there is a problem in your code.

Write `A(2, 7)` for the element in the second row and seventh column of a matrix `A`. `A(2:4, 7:12)` is the 3-by-6 submatrix of `A` containing its rows 2 through 4 and columns 7 through 12. `A(:, 7:12)` is the submatrix of `A` containing all its rows and the columns 7 through 12.

Write `A′` for the transpose of the matrix `A`.

`A*B` is the matrix product of `A` and `B`.

`A.*B` is the elementwise product.

Simulation workshop

1. Save Euler_integration.m under a different name and use the new file to experiment with various changes.

(a) Change `dur`, `Dt`, and the initial `s`.

(b) Right now the state dynamics are $\boldsymbol{s}^{(1)}{}_t$ (i.e. $\Delta \boldsymbol{s}_t / \Delta t$) = $\boldsymbol{s}_t$. Make the program more flexible by introducing a variable α and setting the equation to $\boldsymbol{s}^{(1)}{}_t = \alpha \boldsymbol{s}_t$.

(c) Plot the exact, continuous-time solution to the corresponding differential equation $\mathrm{d}\boldsymbol{s}/\mathrm{d}t = \alpha \boldsymbol{s}$ with a dashed red line in the same panel as the discrete-time, Euler approximation.

(d) Change other aspects of the graphing by altering the code following "`% plot`".

2. To get practice with states and configurations, write an m-file that uses Euler integration to express and solve (approximately) the second-order differential equation $d^2\boldsymbol{q}/dt^2 = -\alpha\boldsymbol{q}$, with initial conditions $\boldsymbol{q} = 0$ and $d\boldsymbol{q}/dt = \alpha^{1/2}$. Also plot the exact solution as a dashed red line. Try different time steps.

3. In Saccadic_linear.m, try larger and smaller values for α, the desired-dynamics coefficient. Does anything odd happen when α is very large? Why?

4. Write an m-file that uses feedback linearization to control a system with state dynamics $d^2\boldsymbol{q}/dt^2 = \boldsymbol{a} - \alpha\boldsymbol{q}$, for a small constant α. Make the agent drive $\boldsymbol{q}$ to a target $\boldsymbol{q}^*$ that jumps every 500 ms, as in Saccadic_linear.m.

Other resources

Hanneton S et al. (1997) Does the brain use sliding variables for the control of movements? *Biological Cybernetics* 77: 381-393

Khalil HK (2002) *Nonlinear Systems*. Prentice Hall

Slotine JJE, Li W (1991) *Applied Nonlinear Control*. Prentice Hall

2 Supervised learning

Agents must learn their policies based on information from their sensors. To explore the principles in a simple setting, we will first consider tasks where a learner tries to deduce a *target function* $\boldsymbol{\tau}$ from examples of $\boldsymbol{x}$'s and $\boldsymbol{\tau}(\boldsymbol{x})$'s.

This task is called *supervised* learning; the $\boldsymbol{x}$'s are called *inputs*; the $\boldsymbol{\tau}(\boldsymbol{x})$'s are *target outputs*, *desired* outputs, *labels*, or *supervisor* or *teacher* signals; and each ordered pair $(\boldsymbol{x}, \boldsymbol{\tau}(\boldsymbol{x}))$ received by the learner is an *example*. Usually a learner can't deduce $\boldsymbol{\tau}$ precisely but seeks an estimate $\langle\boldsymbol{\tau}\rangle$ that *approximates* $\boldsymbol{\tau}$.

The other main type of learning, besides supervised, is *reinforcement* learning, where an agent learns a *policy*, based not on teacher signals telling it what its actions should be, but on reward signals. Reinforcement learning is important for sensorimotor systems, and will be examined later. But supervised learning is also crucial, because auxiliary networks, learning from teacher signals, can help policy networks learn their tasks.

Error and loss

To formulate a supervised-learning task, we define the *error* vector

(35) $$\boldsymbol{e}(\boldsymbol{x}) = \langle\boldsymbol{\tau}\rangle(\boldsymbol{x}) - \boldsymbol{\tau}(\boldsymbol{x})$$

Then we choose a *loss* function, L — a scalar function of $\boldsymbol{e}$, usually non-negative, with a single, global minimum at $\boldsymbol{e} = \mathbf{0}$, and no other minima. An example is the *squared error,*

(36) $$L = \boldsymbol{e}^{\mathsf{T}}\boldsymbol{e}$$

Then the aim of learning is to adjust $\langle \boldsymbol{\tau} \rangle$ to minimize the expected (i.e. average) loss, E[L], averaged across all possible inputs $\boldsymbol{x}$, taking into account that some $\boldsymbol{x}$'s may be more common than others.

Neural networks

The learners we study in this course are layered networks of computational units called *neurons* — simplified models of the real neural networks in the brain. We write n_l for the number of layers, and $\boldsymbol{y}\{l\}$ for the *signal* in layer l, i.e. the vector of the firing rates of all the neurons in that layer. Each signal (apart from the first layer's) depends on the signal in the previous, upstream layer, by the formula

(37) $$\boldsymbol{y}\{l\} = \boldsymbol{\varphi}(\boldsymbol{W}\{l\}\boldsymbol{y}\{l-1\} + \boldsymbol{b}\{l\})$$

Here $\boldsymbol{W}\{l\}$ is the matrix of synaptic weights from layer $l - 1$ to layer l; $\boldsymbol{b}\{l\}$ is the *bias* vector of layer l (representing baseline activities of the cells in the layer); and $\boldsymbol{\varphi}$ is the *activation function*. The term in parentheses is called the *pre-activation potential,*

(38) $$\boldsymbol{v}\{l\} = \boldsymbol{W}\{l\}\boldsymbol{y}\{l-1\} + \boldsymbol{b}\{l\}$$

For our activation function, we will mainly use the *rectified-linear-unit*, or *relu*, function,

(39) $$\boldsymbol{y}\{l\} = \boldsymbol{\varphi}(\boldsymbol{v}\{l\}) = \max(0, \boldsymbol{v}\{l\})$$

in layers 2 through $n_l - 1$, and the *identity* function, $\boldsymbol{y}\{n_l\} = \boldsymbol{v}\{n_l\}$, in the final layer. No activation function is applied in the first layer of the network either, so we have also $\boldsymbol{y}\{1\} = \boldsymbol{v}\{1\}$.

Another popular activation function is the *hyperbolic tangent*,

$$\boldsymbol{y}\{l\} = \tanh(\boldsymbol{v}\{l\}) \tag{40}$$

which has some advantages, e.g. it is bounded between –1 and 1, and so tanh-neuron signals can never fly off to infinity, as may happen with relu.

In any layered network, we regard the first-layer signal, $\boldsymbol{y}\{1\}$, as the network's *input*, $\boldsymbol{x}$. The network's *output*, is usually its final-layer signal, $\boldsymbol{y}\{n_l\}$, though sometimes it is convenient to make $\langle\boldsymbol{\tau}\rangle(\boldsymbol{x})$ some function of $\boldsymbol{y}\{n_l\}$. Often we write simply $\boldsymbol{y}$ for $\langle\boldsymbol{\tau}\rangle(\boldsymbol{x})$, and $\boldsymbol{y}^*$ for the desired output $\boldsymbol{\tau}(\boldsymbol{x})$, in which case (35) becomes

$$\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{y}^* \tag{41}$$

So long as they have at least 3 layers, networks with relu or tanh activation functions (or indeed with any of a wide variety of other nonlinear activation functions) are *universal approximators*, meaning we can approximate any continuous function $\boldsymbol{\tau}$ arbitrarily well with a network of this form if we have enough neurons and we find appropriate $\boldsymbol{W}$'s and $\boldsymbol{b}$'s.

Gradient descent

Learning is a matter of finding weights and biases that yield a small E[L]. Finding the very best $\boldsymbol{W}$'s and $\boldsymbol{b}$'s is computationally intractable — too slow and expensive

— but it is feasible to guess $\boldsymbol{W}$ and $\boldsymbol{b}$ and then improve those guesses by *gradient descent*.

To explain this idea, it is convenient to arrange all the adjustable parameters — all the $\boldsymbol{W}\{l\}$ and $\boldsymbol{b}\{l\}$ — together in a single vector, $\boldsymbol{\theta}$. For any one network, the set of all its possible $\boldsymbol{\theta}$'s is called its *parameter space*. $\mathrm{E}[L]$ is a function of $\boldsymbol{\theta}$, because when a network has good parameters then $\mathrm{E}[L]$ is small, and when it has bad parameters then $\mathrm{E}[L]$ is large.

The gradient $\partial\mathrm{E}[L]/\partial\boldsymbol{\theta}$ at any locus in parameter space is a vector pointing in the direction of steepest increase of $\mathrm{E}[L]$. Therefore we can *de*crease $\mathrm{E}[L]$ by adjusting $\boldsymbol{\theta}$ in the opposite direction, setting

(42) $$\boldsymbol{\theta}_{t+\Delta t} = \boldsymbol{\theta}_t - \eta\, \partial\mathrm{E}[L] / \partial\boldsymbol{\theta}$$

where η is some small, positive number. Then we compute $\partial\mathrm{E}[L]/\partial\boldsymbol{\theta}$ at this new spot in parameter space, apply (42) again, and repeat over and over, gradually improving $\boldsymbol{\theta}$. (How do we choose η? We will address that issue in the section *Faster Learning*).

We can picture the graph of $\mathrm{E}[L]$ versus $\boldsymbol{\theta}$ as a high-dimensional surface, the $\mathrm{E}[L]$-*landscape*. Then gradient descent is like walking down the slopes of that landscape in hopes of finding a low basin.

›› run GRADIENT_DESCENT_QUADRATIC.m

Usually we can't compute $\partial\mathrm{E}[L]/\partial\boldsymbol{\theta}$, but we can estimate it by computing $\partial L_t/\partial\boldsymbol{\theta}$, where L_t is the loss for the current example. At each time point t, the network receives an example $(\boldsymbol{x}_t, \boldsymbol{\tau}(\boldsymbol{x}_t))$, and computes its estimate $\langle\boldsymbol{\tau}\rangle(\boldsymbol{x}_t)$ and its error $\boldsymbol{e}_t = \boldsymbol{\tau}(\boldsymbol{x}_t) - \langle\boldsymbol{\tau}\rangle(\boldsymbol{x}_t)$. From $\boldsymbol{e}_t$ it can compute $\partial L_t/\partial\boldsymbol{\theta}$, which is an estimate of $\partial\mathrm{E}[L]/\partial\boldsymbol{\theta}$. The estimate is *unbiased*, meaning it equals $\partial \mathrm{E}[L]/\partial \boldsymbol{\theta}$ on average, but of course it does vary around this mean, and that variation is called *gradient noise*.

Backprop

The *error-backpropagation* algorithm, or *backprop*, computes $\partial L_t/\partial \boldsymbol{\theta}$ for all the adjustable parameters $\boldsymbol{\theta}$ in a layered network. As a step toward that goal, it computes the derivatives of L_t with respect to all the network's pre-activation potentials, $\boldsymbol{v}\{l\}$. Dropping the subscript t's to avoid clutter, we see by the chain rule that

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{v}\{l\}} &= \frac{\partial L}{\partial \boldsymbol{v}\{l+1\}} \circ \frac{\partial \boldsymbol{v}\{l+1\}}{\partial \boldsymbol{y}\{l\}} \circ \frac{\mathrm{d}\boldsymbol{y}\{l\}}{\mathrm{d}\boldsymbol{v}\{l\}} \\ &= \frac{\partial L}{\partial \boldsymbol{v}\{l+1\}} \circ \boldsymbol{W}\{l+1\} \circ \frac{\mathrm{d}\boldsymbol{y}\{l\}}{\mathrm{d}\boldsymbol{v}\{l\}}\end{aligned} \tag{43}$$

This equation tells us how to find $\partial L/\partial \mathbf{v}$ for layer l given $\partial L/\partial \mathbf{v}$ for layer $l + 1$. To get started, we need $\partial L/\partial \mathbf{v}$ for the final layer. Usually that is easy to compute, though the details depend on the formula for L. If L is the squared-error loss, as in (36), and the final-layer activation function is the identity, then

$$\frac{\partial L}{\partial \boldsymbol{v}\{n_l\}} = 2\boldsymbol{e}^{\mathsf{T}} \frac{\mathrm{d}\boldsymbol{e}}{\mathrm{d}\boldsymbol{v}\{n_l\}} = 2\boldsymbol{e}^{\mathsf{T}} \frac{\mathrm{d}\boldsymbol{y}\{n_l\}}{\mathrm{d}\boldsymbol{v}\{n_l\}} = 2\boldsymbol{e}^{\mathsf{T}} \tag{44}$$

So backprop starts at the output end with (44) and works backward layer by layer through the network with (43) to get all the $\partial L/\partial \mathbf{v}\{l\}$.

This same procedure also yields the derivatives of L with respect to the biases $\boldsymbol{b}\{l\}$, because

(45) $$\frac{\partial L}{\partial \boldsymbol{b}\{l\}} = \frac{\partial L}{\partial \boldsymbol{v}\{l\}} \circ \frac{\partial \boldsymbol{v}\{l\}}{\partial \boldsymbol{b}\{l\}} = \frac{\partial L}{\partial \boldsymbol{v}\{l\}}$$

To get the derivative of L with respect to the *weights*, we again apply the chain rule,

(46) $$\frac{\partial L}{\partial \boldsymbol{W}\{l\}} = \frac{\partial L}{\partial \boldsymbol{v}\{l\}} \circ \frac{\partial \boldsymbol{v}\{l\}}{\partial \boldsymbol{W}\{l\}} = \boldsymbol{y}\{l-1\}\frac{\partial L}{\partial \boldsymbol{v}\{l\}}$$

Given these derivatives, we can adjust the weights and biases by gradient descent, $\Delta \boldsymbol{W} = -\eta(\partial L/\partial \boldsymbol{W})^{\mathsf{T}}$, $\Delta \boldsymbol{b} = -\eta$ $(\partial L/\partial \boldsymbol{b})^{\mathsf{T}}$.

›› run BACKPROPAGATION.m

Finding other gradients by backprop

We have seen that backprop can compute the gradient of L with respect to many network variables, including the input vector, $\boldsymbol{y}\{1\}$ — also called $\boldsymbol{v}\{1\}$ or $\boldsymbol{x}$. But given any vector $\boldsymbol{\alpha}$ of the same dimensionality as $\boldsymbol{y}\{n_l\}$, we can replace L by $\boldsymbol{\alpha}^{\mathsf{T}}\boldsymbol{y}\{n_l\}$ in (43) and (44). It follows that, if we begin backprop with $\boldsymbol{\alpha}$ instead of $2\boldsymbol{e}$, we get $\boldsymbol{\alpha}^{\mathsf{T}}\,\partial\boldsymbol{y}\{n_l\}/\partial\boldsymbol{x}$ at the input end. In other words, we can compute the gradient of any network's output with respect to its input, multiplied by an arbitrary vector $\boldsymbol{\alpha}$, by backpropagating $\boldsymbol{\alpha}$. In brief,

(47) $$\text{backpropagating } \boldsymbol{\alpha} \rightarrow \boldsymbol{\alpha}^{\mathsf{T}}\partial\boldsymbol{y}\{n_l\}/\partial\boldsymbol{x}$$

This principle will be useful for policy-learning in Chapter 4.

Feedback alignment

Very likely, backprop can't operate in the brain, because of (43). That equation contains the term $\boldsymbol{W}\{l + 1\}$, which means that the feedback circuits that compute the parameter adjustments must know the weight matrices $\boldsymbol{W}$. That is, neurons in these feedback pathways, whose signals represent $\partial L/\partial \mathbf{v}\{l\}$, must know the weights of vast numbers of synapses on *other* neurons, in the forward path — the neurons whose signals are $\boldsymbol{y}\{l\}$. Despite decades of research on inter-neuronal communication, we have no known mechanism that could convey this synaptic data with anywhere near the speed and capacity required for backprop.

But another mechanism, very close to backprop, works almost as well and *is* biologically feasible. If the $\boldsymbol{W}$'s in (43) are replaced by *random*, *fixed* matrices then there is no need to communicate synaptic weights from cell to cell, and yet (it can be shown) the forward-path weight matrices $\boldsymbol{W}\{l\}$ evolve to resemble the fixed matrices in the feedback circuits, so that in the end it is as if those feedback matrices had been set equal to the $\boldsymbol{W}\{l\}$'s, as in backprop.

›› run FEEDBACK_ALIGNMENT.m

Minibatches

Backprop (and feedback alignment) networks learn much faster if they receive, at each time t, not a single example $(\boldsymbol{x}_t, \boldsymbol{\tau}(\boldsymbol{x}_t))$, but a set of n_m examples called a *minibatch*, with n_m usually in the range 100–256. One reason minibatches help is that with more examples we can compute better estimates of $\partial \mathrm{E}[L]/\partial \boldsymbol{\theta}$, reducing gradient noise. Another reason is that we can arrange all the data vectors of the minibatch side by side in a single matrix, and computers can process that matrix of, say, 100 examples much faster than they can 100 individual examples.

When we use a minibatch of n_m examples, we are running n_m different inputs through the same network, so we get n_m different sets of $\boldsymbol{v}$'s and $\boldsymbol{y}$'s, and n_m different outputs, errors, and losses, but all n_m runs share the same $\boldsymbol{W}$'s and $\boldsymbol{b}$'s. We will write L_m for the loss in example m of the minibatch, $v\{l\}_{i,m}$ for the pre-activation potential of neuron i of layer l in example m, and analogous things for other variables. In this setting, L means the average of all n_m losses in the minibatch. Backprop equations (44), (43), (45), and (46) then become

(48) $$\frac{\partial L_m}{\partial v\{n_l\}_{i,m}} = 2\,e_{i,m}\,\frac{dy\{n_l\}_{i,m}}{dv\{n_l\}_{i,m}} = 2\,e_{i,m}$$

(49) $$\frac{\partial L_m}{\partial v\{l\}_{i,m}} = \left(\Sigma_j \frac{\partial L_m}{\partial v\{l+1\}_{j,m}} W\{l+1\}_{j,i}\right)\frac{dy\{l\}_{i,m}}{dv\{l\}_{i,m}}$$

(50) $$\frac{\partial L}{\partial b\{l\}_i} = \left(\frac{1}{n_m}\right)\Sigma_m \frac{\partial L_m}{\partial v\{l\}_{i,m}}$$

(51) $$\frac{\partial L}{\partial W\{l\}_{i,k}} = \left(\frac{1}{n_m}\right)\Sigma_m\, y\{l-1\}_{k,m}\frac{\partial L_m}{\partial v\{l\}_{i,m}}$$

Brains likely can't use minibatches, but in most of our simulations we will use them nonetheless, for speed.

We get much the same result we would have obtained without minibatches, but faster.

›› run BACKPROPAGATION_MINIBATCH.m

Initialization

In many problems, backprop-based learning will succeed or fail depending on how we set the initial values of the weights and biases. One rule of thumb, which does *not* always work, is that in a relu network, the weights should start out small and the biases mainly positive, say with a mean near 0.1. That way, most neurons in the network will yield nonzero signals given most inputs $\boldsymbol{x}$, whereas with negative $\boldsymbol{b}$'s, or large $\boldsymbol{W}$'s that overpowered the $\boldsymbol{b}$'s, many neurons would be in the flat part of the relu function — a problem, as neurons can't learn when they are in the flats because then the derivative $\partial \boldsymbol{y}/\partial \boldsymbol{v}$, which drives learning in (43), is $\mathbf{0}$.

Activation functions like relu that are zero over some stretch of their domain are said to show *hard-zero saturation*. One might expect better learning with other activation functions, such as the hyperbolic tangent, that are flat nowhere, but in practice, relu networks usually learn better, maybe because the relu function, being so simple, simplifies the E[L]-landscape. Also, most common activation functions *without* hard-zero saturation, such as tanh, are *almost* flat over parts of their domains, and so they don't really avoid the problem. Real neurons in the brain show hard-zero saturation, as their firing rates can't be negative.

Faster learning

Speed of learning hinges on the rate constant η in (42). If η is too small, then $\boldsymbol{\theta}$ improves very slowly. If η is too large, it may overstep a minimum and *increase* E[L].

Annealing

Often it helps to *anneal* η, shrinking it as time goes by so the network learns rapidly at first and then slows down to home in on an optimal $\boldsymbol{\theta}$. But choosing the annealing schedule (the rate at which η shrinks) is a matter of trial and error, as is choosing the initial η.

Momentum

Backprop usually learns faster with *momentum*, meaning that the parameter adjustment $\Delta\boldsymbol{\theta}$ is not exactly $-\eta$ times the gradient, but also depends on the $\Delta\boldsymbol{\theta}$ in the previous time step:

$$\Delta\boldsymbol{\theta}_t = \beta\Delta\boldsymbol{\theta}_{t-\Delta t} - \eta\,\partial L/\partial\boldsymbol{\theta} \tag{52}$$

where β is a number between 0 and 1, usually 0.9 or larger. The result is that if the negative-gradient $-\partial L/\partial\boldsymbol{\theta}$ *consistently* points in some direction over several time steps, then $\boldsymbol{\theta}$ will build up speed in that direction, whereas if the negative-gradient flicks back and forth between opposite directions from time step to time step then that back-and-forth motion will be averaged out and ignored.

This averaging is crucial because the E[L]-landscape is often poorly *conditioned*, meaning its ridges, saddles, valleys, and basins are much steeper in some dimensions than in others. Imagine a deep river valley that snakes its way downhill very slowly, i.e. with a shallow grade, but that has very steep sides. In other words, the land slopes steeply down to the river on either side, but slopes downhill much more gradually in the direction of the river's flow. In a valley like this, it is usually a bad idea to step exactly down the gradient: if you are standing on either side-slope, the negative-gradient will point straight down that slope, at almost a right angle to the riverbank, and will be huge, so unless η in (42) is tiny, $\Delta\boldsymbol{\theta}$ will also be huge, and so $\boldsymbol{\theta}$ will tend to jump back and forth *across* the valley, $\Delta\boldsymbol{\theta}_t$ taking it from one side to the other, and $\Delta\boldsymbol{\theta}_{t+\Delta t}$ taking it back again, rather than snaking steadily downhill along the valley floor.

On the other hand, if η *is* tiny, then $\boldsymbol{\theta}$ will descend the steep side in an orderly way, but then its progress along the valley floor will be very slow.

So no single η is appropriate for the steep and the shallow dimensions. But with (52), the back-and-forth jumps across the valley will tend to cancel each other out, and any motion *along* the valley will compound, so $\boldsymbol{\theta}$ improves more efficiently.

›› run MOMENTUM_QUADRATIC.m

Nesterov momentum

In (52), we applied the momentum *after* computing the gradient, i.e. we received the input $\boldsymbol{x}_t$, computed the gradient, and then adjusted $\boldsymbol{\theta}$. Another approach, often faster, is to adjust $\boldsymbol{\theta}$ to an interim value, $\boldsymbol{\theta}'$, by applying the momentum, *then* compute the gradient at $\boldsymbol{\theta}'$, and move down that gradient:

$$\begin{aligned}\boldsymbol{\theta}' &= \boldsymbol{\theta}_{t-\Delta t} + \beta\,\Delta\boldsymbol{\theta}_{t-\Delta t}\\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}' - \eta\partial L/\partial\boldsymbol{\theta}\,(\boldsymbol{\theta}')\\ \Delta\boldsymbol{\theta}_t &= \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-\Delta t}\end{aligned} \tag{53}$$

The rationale is, if we are already committed to taking a big momentum step, then we should check the lay of the land after that step, not before.

›› run NESTEROV_QUADRATIC.m

RMSprop

We have seen that E[L]-landscapes have steep and shallow dimensions, and that no single η can work well for all of them. Therefore it is useful to define a separate learning-rate factor for each dimension, i.e. for each parameter θ_i. One approach is to choose a single, scalar η as usual, but then divide each $\Delta\theta_i$ by $|\partial L_t/\partial\theta_i|$, replacing (42) by

$$\theta_{t+\Delta t,\,i} = \theta_{t,\,i} - \eta\;\partial L_t/\partial\theta_i \,/\, \left(\left|\partial L_t/\partial\theta_i\right| + \varepsilon\right) \tag{54}$$

where ε is some tiny number, added to prevent us ever dividing by 0. With (54), the sizes of the adjustments are tightly controlled, as $|\Delta\theta_i|$ is always either 0 or very close to η. No θ_i can take huge steps down a steep dimension and fly out of control, or take minuscule steps along a nearly-flat dimension and stagnate.

But because of gradient noise, it is better to scale $\Delta\theta_i$ not by the current $|\partial L_t/\partial\theta_i|$ alone, but by a running average of recent $|\partial L_t/\partial\theta_i|$. In the method called RMSprop, we introduce a vector $\boldsymbol{\theta}^{\text{rms}}$, where the superscript means *root-mean square*, we initialize it to **0**, we update it by

$$\theta^{\text{rms}}{}_{t,i} = \beta\,\theta^{\text{rms}}{}_{t-\Delta t,i} + (1-\beta)\,(\partial L_t/\partial\theta_i)^2 \tag{55}$$

where $\theta^{\text{rms}}{}_{t,i}$ is the i-th element of $\boldsymbol{\theta}^{\text{rms}}$ at time t, and we use it to adjust $\boldsymbol{\theta}$,

(56) $$\theta_{t+\Delta t,\,i} = \theta_{t,\,i} - \eta\left(\partial L_t/\partial\theta_i\right)/\left(\sqrt{\theta^{\text{rms}}{}_{t,\,i}} + \varepsilon\right)$$

In (55), setting $\beta = 0.9$ usually strikes a nice balance between too little and too much memory. If we set $\beta = 0$ (that is, we scale $\Delta\boldsymbol{\theta}$ based only on the current mini-batch) then we overreact to noise, but if we set β much higher than 0.9 then we keep obsolete information about gradients in a region of $\boldsymbol{\theta}$-space we have already left behind.

›› run RMSPROP_QUADRATIC.m

Adam

The most popular way of adjusting parameters in AI right now is probably the method of *adaptive moments*, or *adam*, which maintains running estimates of the mean, or *first moment* of $\partial L/\partial\boldsymbol{\theta}$,

(57) $$m^1{}_{t,i} = \beta_1 m^1{}_{t-\Delta t,i} + (1-\beta_1)\,\partial L_t/\partial\theta_i$$

and the *second moment*,

(58) $$m^2{}_{t,i} = \beta_2 m^2{}_{t-\Delta t,i} + (1-\beta_2)(\partial L_t/\partial\theta_i)^2$$

To get started, we have to initialize $\boldsymbol{m}^1$ and $\boldsymbol{m}^2$, and for lack of any better guess, we set them both to $\mathbf{0}$, with the result that our running estimates in (57) and (58) are biased toward $\mathbf{0}$. Those biases may not hurt us, because they fade with time, as we process more and more examples and the initial values have less and less effect. But just to be safe, we can bias-correct. It can be shown that the following corrected estimates are unbiased:

(59)
$$\boldsymbol{m}^{1\prime}_{t} = \boldsymbol{m}^{1}_{t} / (1 - \beta_1^{t/\Delta t})$$
$$\boldsymbol{m}^{2\prime}_{t} = \boldsymbol{m}^{2}_{t} / (1 - \beta_2^{t/\Delta t})$$

where $\beta_1^{t/\Delta t}$ and $\beta_2^{t/\Delta t}$ are the β's from (57) and (58) raised to the power $t/\Delta t$.

Finally, we adjust the parameters,

(60)
$$\theta_{t+\Delta t,\,i} = \theta_{t,\,i} - \eta\, m^{1\prime}_{t,\,i} \,/ \left(\sqrt{m^{2\prime}_{t,\,i}} + \varepsilon\right)$$

With adam, $m^1_{t,i}$ in the numerator acts like momentum, cancelling out components of the gradient that reverse direction from one time step to the next, while $m^2_{t,i}$ in the denominator acts like RMSprop, bounding the sizes of the adjustments, $\Delta\theta_i$. Usually we set $\varepsilon = 10^{-8}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and choose η by trial and error.

›› run ADAM_QUADRATIC.m

›› run COMPARE_DESCENT_QUADRATIC.m

›› run COMPARE_DESCENT_COMPLEX.m

Batch normalization

Even with adam, a backprop network may still learn slowly because its signal-elements are correlated, i.e. the individual elements $y\{l\}_i$ in any one signal vector $\boldsymbol{y}\{l\}$ are not independent, but correlate with each other across examples. Roughly speaking, the network has trouble sorting out which elements contribute what to E[L], when those elements are related to each other.

›› run CORRELATED_ELEMENTS.m

It is possible to decorrelate signal elements by techniques such as principal components analysis, but all such methods are expensive. A cheaper approach is partial decorrelation: we can often improve learning simply by ensuring that each element of each signal $\boldsymbol{y}\{l\}$ has a mean of 0 and a standard deviation of 1.

In this method, called *batch normalization* or *batchnorm*, we normalize each layer's signal $\boldsymbol{y}\{l\}$: we calculate the mean $\mu\{l\}_i$ and standard deviation $\sigma\{l\}_i$ of each $y\{l\}_i$ across the current minibatch, then subtract off the means and divide by the standard deviations to create a *normalized signal*,

(61) $$y'\{l\}_{i,m} = (y\{l\}_{i,m} - \mu\{l\}_i) / \sigma\{l\}_i$$

where $y\{l\}_{i,m}$ is the i-th element of example m in the minibatch. (There is another version of batchnorm which normalizes the pre-activation potential $\boldsymbol{v}\{l\}$ rather than $\boldsymbol{y}\{l\}$, but it is more complicated and usually doesn't work any better.)

To backprop through this batchnorm network, we must compute $\partial L_m/\partial \boldsymbol{y}$ from $\partial L_m/\partial \boldsymbol{y}'$,

(62) $$\begin{aligned}\partial L/\partial y_{i,m} = (&n_m \partial L_m/\partial y'_{i,m} - \\ &y'_{i,m}\Sigma_n y'_{i,n}\partial L_n/\partial y'_{i,n} - \Sigma_n \partial L_n/\partial y'_{i,n}) \\ &/\, n_m(\sigma_i + \varepsilon)\end{aligned}$$

where n_m is the number of examples in the minibatch, and ε is a small number introduced to prevent us dividing by 0.

Batchnorm sometimes improves learning dramatically, though in other cases it makes it worse. It often works better with tanh than with relu activation functions, because with relu, zeroing the mean pre-activation potential pushes it into hard-zero saturation, i.e. the flat part of the relu curve.

›› run BATCHNORM.m

Brains likely can't use batchnorm, because they likely can't use minibatches, but variants of batchnorm may be feasible. For example, each neuron may monitor its own activity through time, and alter itself to keep its mean firing rate and variance near some target values.

Comparing algorithms

To assess learning algorithms, and judge their biological feasibility, we have to consider how much energy they consume, how long they take to learn, and how much memory they need.

Energy

Computer scientists measure the complexity of their algorithms by counting the number of floating-point operations, or *flops*. For example, multiplying or adding two floating-point variables counts as one flop. Usually, we will assume that methods that involve a lot of flops in a computer also involve a lot of computational effort in a brain.

On the other hand, there are operations that are hard in computers but easy in brains, or vice versa, so we try to identify these. Moreover, the energy cost of an algorithm in the brain depends not only on its flop count but also on how many neurons it needs, as we will discuss below.

Time

In a computer, the running time of an algorithm depends mainly on the number of flops, because computers work mainly *serially*, doing just a few operations at a time in the central processing unit (CPU) or several thousand at a time in the graphical processing unit (GPU). But in the brain, vast arrays of neurons do *billions* of operations in parallel, so learning time depends more on the number of examples required than on the flops.

Memory

A crucial difference between brains and computers is that brains represent data in at least two different forms: in neuron parameters such as biases and synaptic weights, or in signals (firing rates) of neurons. Synapses are smaller than neurons and more numerous, so they are a metabolically cheap, high-capacity memory store. But they can't send their information to other neurons or even other places in the same neuron — there is no known way for messages about the strengths of synapses to travel from one place to another. Information stored in neural firing, on the other hand, is *transmissible* — it can travel to other cells along axons. Therefore in our algorithms we have to consider which pieces of information are transmitted from place to place, so we can deduce both the synaptic and the transmissible memory requirements, or in other words how many synapses and how many neurons we need.

Scaling

Sensorimotor tasks are often *high-dimensional*, meaning the state vectors have many elements. For example, visual inputs to the brain run on the optic nerves, with more than a million axons each, and so visual input is a vector of more than 2 million elements.

We are interested in how complexity, time, and memory *scale* with dimensionality. If the number of flops needed for a computation increases linearly with the dimensionality n of its input, then we say the computation is O(n), or *on the order of* n. If it grows with the second power of n then it is O(n^2). For example, computing the dot product of two n-element vectors takes n scalar multiplications, and about as many additions, and so the dot product is an O(n) operation. Multiplying an n-element vector by an n-by-n matrix is O(n^2). Multiplying two n-by-n matrices is O(n^3). O(n^3) operations are slow and expensive when n is large.

Other resources

Goodfellow I, Bengio Y, Courville A (2016) *Deep Learning*. MIT Press.

Kingma DP, Ba JL (2015) Adam: a method for stochastic optimization. arXiv:1412.6980

Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv:1502:03167

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. *Nature* 323

3 Sensory maps

With backprop or feedback alignment, neural networks can learn visual tasks, such as recognizing objects in images. To feed an image into a network, we encode it as a vector, e.g. if the image is greyscale, with *n* pixels, we string those greyscale values into an *n*-element vector and feed that into the network as its input, $\boldsymbol{x}$. But this vector does not convey the *geometry* of the image: that each vector-element represents a pixel in a 2-D array, and each pixel is spatially related to every other pixel — right, left, above, or below it, and close by or far away. This geometry matters, because most things we want to recognize visually are *contiguous* objects, which are defined by their *boundaries*, which in images show up as *edges*, i.e. sharp differences in brightness or color between *neighboring* pixels.

How can we convey the geometric relations pf the image pixels to a neural network? In this chapter we explore one approach that is used in the brain and in computer vision.

Receptive fields

In the human eye, light is detected by photoreceptors (rods and cones) in the retina. From those receptors, signals pass through a series of layers — bipolar cells and retinal ganglion cells in the eye, and then, in the brain, the lateral geniculate nuclei, primary visual cortex, and so on. Any one bipolar cell gets input from only a small fraction of the photoreceptors, and those input receptors are not scattered about the retina but are all close together in a contiguous patch called the *receptive field* of the bipolar cell.

The bipolar cell distinguishes different input patterns, responding more to some patterns of activity among its input receptors than to others. And because those receptors are all neighbors, the cell distinguishes *local* features of the image. For instance, many bipolar cells respond to small spots of light or dark against contrasting backgrounds. Owing to their input connections, then, the cells detect geometric features, namely spots, that are useful for identifying objects.

Feature maps

Moreover, neighboring bipolar cells have neighboring receptive fields, and therefore the sheet of bipolar cells is a *map* of visual space, just as the sheet of receptors is. The loci in this second map — the individual bipolar cells — do not represent the brightness of points in space as the receptors do. Rather, they represent spots of contrast. We say the bipolars form a *feature map*, each cell signaling the presence or absence of an image feature, in this case a dark or light spot on a contrasting background, in a region of visual space.

Because the bipolar cells form a map, just as the receptors do, the brain can apply the same principle again at the next layer: any one retinal ganglion cell gets input from only a small contiguous patch of bipolar cells, and neighboring ganglion cells get input from neighboring patches. So the ganglion cells form a new feature map. And so on through the lateral geniculate nuclei, primary visual cortex, and beyond.

Layer by layer, the features become more and more abstract. So-called *simple cells* in primary visual cortex combine information from neighboring spot-detectors in the lateral geniculate to detect straight, oriented edges. Farther downstream, other maps combine neighboring edges into contours; and maybe neighboring contours into outlines of shapes, and shapes into scenes, though the details are less clearly understood the deeper we go into the visual system.

In neuroscience, these visual areas are called *retinotopic* maps, and analogous terms exist for other senses. Primary somatosensory cortex, which gets information from touch receptors, is a *somatotopic* map, where neighboring cells get information from neighboring patches of skin. And some auditory areas are *tonotopic* maps, where neighboring cells represent tones of similar frequencies.

Given any one retinal ganglion cell, its *input field* is the set of bipolar cells that project to it, whereas its *receptive field* is the set of photoreceptors that communicate with it, via the bipolars. And similarly for any layer of any sensory system: a cell's receptive field is a patch of receptors, and its input field is a patch of cells in the previous, upstream map. Or more generally, a cell may have several input fields, if it receives projections from more than one map, e.g. direct projections from cells two or more layers back, or feedback from maps farther downstream.

Connection matrices

The next file of sample code constructs a network where all layers except the final one are maps. It defines a map *size* for each map layer, i.e. how many pixels wide the map is (the code creates only square maps, so any map's height is the same as its width). All maps share the same *field-fraction*, which sets the width of the input fields as a fraction of the width of the up-stream map, e.g. if the field-fraction is 0.25 and the layer-2 map is 32 pixels wide, then each cell in layer 3 has an input field in layer 2 that is $32 \times 0.25 = 8$ pixels wide (and tall).

For each cell in each map layer after the first (the input) layer, the posted code file chooses an input field, located so that neighboring cells have neighboring fields. For each layer *l*, the locations and sizes of all its cells' fields are stored in a *connection matrix*, $\boldsymbol{C}\{l\}$, of ones and zeros. For example, cell 35 in layer 3 receives a projection (an axon branch) from cell 140 in layer 2, and so $\boldsymbol{C}\{3\}_{35,\,140} = 1$, but that same cell 35 in layer 3 receives *no* projection from cell 130 in layer 2, and so $\boldsymbol{C}\{3\}_{35,\,130} = 0$. As the layer-3 map has 16 by 16 pixels, and the layer-2 map has 32 by 32, $\boldsymbol{C}\{3\}$ has $16^2 = 256$ rows (one for each neuron in layer 3) and $32^2 = 1024$ columns (one for each neuron in layer 2). As each cell in layer 3 has an input field 8 pixels wide (and tall), it follows that each row in C{3} has exactly $8^2 = 64$ entries that are equal to 1, and the other $1024 – 64 = 960$ elements are zeros. In short, $\boldsymbol{C}\{3\}$ is structured so that each cell in layer 3 gets input from only 64 cells in layer 2, and those 64 cells form a contiguous 8-by-8 patch — an input field. The code plots the whole connection matrix $\boldsymbol{C}\{3\}$, to show that most connections are absent (matrix entries are 0s) and only a few exist (entries are 1s). It also plots row 35 of the matrix reshaped into a 32-by-32 map, to show the geometry of the input field of cell 35.

The same code file ensures that the network's weight matrices agree with the connection matrices: the synaptic weight from any one neuron to any other can be nonzero only if there is a connection between those two cells, i.e. if $\boldsymbol{C}\{l\}_{i,j} = 0$ then $\boldsymbol{W}\{l\}_{i,j} = 0$. Of course the converse is not true: $\boldsymbol{W}\{l\}_{i,j}$ may happen to be 0 even if $\boldsymbol{C}\{l\}_{i,j} = 1$.

Because $\boldsymbol{C}$ and $\boldsymbol{W}$ matrices between map layers have few nonzero elements, they are said to be *sparse*. The code shows how they can be converted into Matlab's "sparse" data type, which can make operations on them run faster. In contrast, networks like those in Chapter 2, where each cell in each layer projects to every cell in the next layer, are called *dense* or *fully-connected* networks. In computer vision, most neural networks have several layers of maps, with sparse weight matrices, followed by a layer or two with dense connections.

›› run MAP_NETWORK.m

Reading handwritten numerals

One way to test computer-vision algorithms is to see how well they learn to recognize handwritten numerals in the MNIST (Modified National Institute of Standards) data set, put together by Yann LeCun and colleagues. MNIST contains a *training set* of 60,000 examples, each a 28-by-28-pixel image of one of the digits "0" through "9", and a *test set* of 10,000 additional examples, which we present to the network only *after* it has finished learning, to assess how well it recognizes examples it did not see during its training. For each of the 70,000 example images, MNIST also provides a *label*, or in other words a desired-output vector $\boldsymbol{y}^*$, which reveals what numeral is drawn in the image. Of course the learning network does not receive the labels as part of its inputs, but it does use them to compute its errors, as in equation (41), and we use them to assess whether the network's answers are correct.

When neural networks are trained on classification tasks, such as recognizing handwritten digits in the MNIST data set, the desired-output vectors (the labels) are usually *one-hot*, meaning there is always one element of the vector equal to 1 and the others are all 0s. For example, if the net is learning to recognize the 10 digits "0" through "9", then the desired output $\boldsymbol{y}^*$ is typically a 10-element vector, set to [1; 0; 0; 0; 0; 0; 0; 0; 0; 0] when the digit is a "0", to [0; 1; 0; 0; 0; 0; 0; 0; 0; 0] when the digit is a "1", and so on.

In this task, therefore, the actual output $\boldsymbol{y}$ of the network must also be a 10-element vector, where each element of $\boldsymbol{y}$ learns to signal the presence of one of the object classes, "0" through "9". We say the network has given the correct answer when the largest element of $\boldsymbol{y}$ is the one with the correct index, e.g. if the digit is a "5" then $y^*_6 = 1$ and all other $y^*_i = 0$, and we say the network has answered correctly if $y_6 > y_i$ for all $i \neq 6$.

When $\boldsymbol{y}^*$ is one-hot then it is a good idea to restrict the elements of $\boldsymbol{y}$ to the range [0, 1] as well, e.g. by sending the final-layer signals through a rectified tanh function, $\boldsymbol{y} = \max(0, \tanh(\boldsymbol{y}\{n_l\}))$.

On the course website you can find the MNIST data set, in the file MNIST.mat, and also MNIST_key.m, which illustrates the format of those data.

```
›› run MNIST_KEY.m
```

Convolution

At present, most computer-vision networks use maps with additional mechanisms which we will consider briefly here but won't cover in detail because they are biologically implausible. Most importantly, these networks use *convolution*, meaning that all the neurons in any one map have the *same* incoming synaptic weights. For example, if those neurons have 2-by-2 input fields, and one of the neurons has weights 2, –3, 1, 5 for its upper-left, upper-right, lower-left, and lower-right input cells, respectively, then every other neuron in the map will also have those same weights at those same loci within its field (but the fields for different neurons will still have different locations within the upstream map, as in Map_network.m).

This arrangement, called *weight-sharing*, is not biological. Real neurons in real sensory maps each have their own synapses and no mechanism to keep their weights equal to those of other neurons, though of course there may be some mechanism, still waiting to be discovered, by which real neural maps could approximate weight sharing.

The set of shared weights that a map applies to all its input fields is called the map's *kernel* or *filter*. So weight sharing forces a map to apply a single filter to all its fields. That constraint can be valuable because if a filter is useful in one part of visual space, then it is likely also useful elsewhere. If it is useful to detect vertical edges in the upper-left part of visual space then it is likely also useful to detect them in the lower-right and everywhere else, because objects with vertical edges may appear anywhere.

Weight-sharing drastically reduces the number of adjustable parameters. In a 20-by-20 map *without* weight-sharing, where each of the 400 neurons has an 8-by-8 input field, there are 400 × 8 × 8 = 25,600 incoming weights. But with weight-sharing there are only 8 × 8 = 64. That is too few to learn almost any visual task, and therefore convolution nets have many maps, usually 5 to 100 of them, in each layer. This way, they get reasonable numbers of parameters, though to do it they need a lot of maps, with a lot of neurons, i.e. convolution nets have a high ratio of neurons to adjustable parameters.

Other resources

LeCun Y et al. (1998) Gradient-based learning applied to document identification. *Proceedings of the IEEE* (*a description of convolution nets*)

4 Reinforcement learning

Sensorimotor agents can’t shape their policies by supervised learning alone, because supervised learning requires teacher signals, $\boldsymbol{\tau}(\boldsymbol{x})$, that say what the current output should have been. There are no such signals for most sensorimotor tasks, as sensory feedback usually can’t tell an agent what its action should have been; it can only report the consequences of the action, such as the resulting state change $\Delta \boldsymbol{s}_t$ and reward r_t. The agent must improve its policy based on that feedback — a process called *reinforcement learning*.

As in Chapter 1, equation (8), the aim is to maximize some *return*, the time-integral of a reward to some final time T,

(63) $$R = \Delta t \, \Sigma_{t=0 \ldots T} \, r_t$$

More generally, we may start at some time t' other than 0, and we may aim to optimize a *discounted* return,

(64) $$R_{t'} = \Delta t \, \Sigma_{t=t' \ldots T} \, \gamma^{t-t'} r_t$$

where γ is the *discounting factor*, between 0 and 1, say 0.99. So if $\gamma < 1$ then we care less and less about rewards further and further in the future. One motivation is that we are less and less certain about those future rewards. Another is that discounting helps us avoid infinite returns when $T = \infty$.

Models

There are two main types of reinforcement learning: model-based and model-free. In *model-based* methods, the agent learns a *model*, or in other words an internal simulation, of the state dynamics of its environment, and uses that model to improve its policy.

But if the state dynamics are complex, the agent may have trouble learning an accurate model. For this reason, a lot of recent work in AI has instead used *model-free* methods, where the agent learns not the full state dynamics but something simpler, an *action-value function*, which represents the expected long-term outcome that will follow if the agent takes a given action in a given state.

Model-free methods have yielded impressive results in machine learning, but the brain clearly uses models for at least some purposes: we can predict the sensory consequences of our actions, and plan by running scenarios in our minds. We will study both approaches, beginning with model-free.

Action-value functions

Many algorithms for reinforcement learning are based on *action-value*- or Q-functions. A Q-function takes as inputs a state $\boldsymbol{s}$ and an action $\boldsymbol{a}$, and yields as output the long-term result — the expected return — achieved by taking action $\boldsymbol{a}$ when in state $\boldsymbol{s}$.

Of course, that expected return depends not only on the action $\boldsymbol{a}$, but also on every action thereafter, till time T. So Q makes sense only if we specify some policy $\boldsymbol{\mu}$ for all those subsequent actions. Moreover, the expected return also depends on how many time steps we have left till time T. Therefore Q-functions are easiest to define if $T = \infty$, because then every t is equally far from T, and so we can define Q in terms of $\boldsymbol{s}$ and $\boldsymbol{a}$ alone, regardless of what the present time is. Given a policy $\boldsymbol{\mu}$, we define $Q^{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{a})$ to be the expected return, from now on, if we are presently (at time t') in state $\boldsymbol{s}$, and we take action $\boldsymbol{a}$, and from then on we choose our actions using policy $\boldsymbol{\mu}$.

(65)
$$Q^{\boldsymbol{\mu}}(\boldsymbol{s},\boldsymbol{a}) = r(\boldsymbol{s},\boldsymbol{a}) + \mathrm{E}_p\left[\Sigma_{t=t'+\Delta t\ldots\infty}\gamma^{t-t'}r(\boldsymbol{s}_t,\boldsymbol{\mu}(\boldsymbol{s}_t))\right]$$

We put the expectation in this formula because, if the state dynamics are stochastic, then the return is a random variable that depends on the distribution p, as in (2). Notice that in (65), the action $\boldsymbol{a}$ need *not* fit the policy $\boldsymbol{\mu}$ — $Q^{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{a})$ still makes sense when $\boldsymbol{a} \neq \boldsymbol{\mu}(\boldsymbol{s})$.

Policy gradients

One powerful method of model-free reinforcement learning is to train a neural network $\langle Q\rangle$ to estimate $Q^{\boldsymbol{\mu}}$, where $\boldsymbol{\mu}$ is the current policy, implemented by another network, separate from $\langle Q\rangle$, and then improve the policy network by adjusting its parameters $\boldsymbol{\theta}^{\boldsymbol{\mu}}$ down the gradient of $\langle Q\rangle$. By the chain rule, we have

(66)
$$\partial\langle Q\rangle/\partial\boldsymbol{\theta}^{\boldsymbol{\mu}} = \partial\langle Q\rangle/\partial\boldsymbol{a} \circ \partial\boldsymbol{a}/\partial\boldsymbol{\theta}^{\boldsymbol{\mu}}$$

To compute the $\partial\langle Q\rangle/\partial\boldsymbol{a}$ term, we backpropagate the single-element vector [1] through the $\langle Q\rangle$ network — just as we backpropagated the vector $\boldsymbol{\alpha}$ in (47). Then we get $\partial\langle Q\rangle/\partial\boldsymbol{\theta}^{\boldsymbol{\mu}}$ by backpropagating $\partial\langle Q\rangle/\partial\boldsymbol{a}$ through the $\boldsymbol{\mu}$-network.

This method is called *deep deterministic policy gradient*, or *DDPG* — “deep” because it uses multilayered networks for $\langle Q \rangle$ and $\boldsymbol{\mu}$, and “deterministic” because it assumes the policy is deterministic (though the state dynamics may be stochastic). The next 6 sections describe concepts we need to understand DDPG.

Bellman equations

In DDPG, the neural network $\langle Q \rangle$ learns $Q^{\boldsymbol{\mu}}$. And because the policy $\boldsymbol{\mu}$ is also learning, and therefore changing, it follows that $Q^{\boldsymbol{\mu}}$ is changing as well, which means $\langle Q \rangle$ must keep readjusting itself, so as to remain always a good approximation to the current $Q^{\boldsymbol{\mu}}$.

How can $\langle Q \rangle$ learn $Q^{\boldsymbol{\mu}}$? Not by supervised learning, because there are no teacher signals that know the true value of $Q^{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{a})$. But useful error signals can be created using the *Bellman equation*,

$$Q^{\boldsymbol{\mu}}(\boldsymbol{s}_t, \boldsymbol{a}_t) = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma^{\Delta t} \mathrm{E}[Q^{\boldsymbol{\mu}}(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+\Delta t}))] \tag{67}$$

This equation follows from the definition of $Q^{\boldsymbol{\mu}}$ in (65): the left-hand side of (67) is the expected return from time t onward, given an initial state $\boldsymbol{s}_t$ and action $\boldsymbol{a}_t$, and the right-hand side is the same thing: the reward r at time t given $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$, plus the discounted, expected return from time $t + \Delta t$ onward given policy $\boldsymbol{\mu}$.

This equation implies a *Bellman error* signal that can be used to train estimates of $Q^{\boldsymbol{\mu}}$, namely

$$\begin{aligned} e^Q &= \langle Q \rangle (\boldsymbol{s}_t, \boldsymbol{a}_t) \\ &\quad - r(\boldsymbol{s}_t, \boldsymbol{a}_t) - \gamma^{\Delta t} \langle Q \rangle (\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+\Delta t})) \end{aligned} \tag{68}$$

In other words, we don't know the true value of $Q^{\boldsymbol{\mu}}(\boldsymbol{s}_t, \boldsymbol{a}_t)$, but we know it must relate to the future value $Q^{\boldsymbol{\mu}}(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{a}_{t+\Delta t})$ by equation (67). This e^Q is a reasonable error signal because if $\langle Q\rangle = Q^{\boldsymbol{\mu}}$ then $e^Q = 0$ on average, and (it can be shown) if $e^Q = 0$ on average for all inputs $(\boldsymbol{s}_t, \boldsymbol{a}_t)$ then $\langle Q\rangle = Q^{\boldsymbol{\mu}} + c$ for some constant c, and that constant is immaterial because the learning rule (66) uses the *gradient* of $\langle Q\rangle$, which doesn't depend on c. This technique of training $\langle Q\rangle$ to agree with its own future values is called *bootstrapping*.

Target networks

How do we use a Bellman error such as (68) to train the network $\langle Q\rangle$? The obvious approach is to compute the gradient of e^Q-squared with respect to $\langle Q\rangle$'s parameters, $\boldsymbol{\theta}^{\langle Q\rangle}$, and adjust $\boldsymbol{\theta}^{\langle Q\rangle}$ down that gradient. That approach does work, but it can get complicated because $\langle Q\rangle$ appears *twice* in the formula (68) for e^Q, and with different inputs in its two appearances. As a result, $\partial e^Q/\partial\boldsymbol{\theta}^{\langle Q\rangle}$ is more complicated than if $\langle Q\rangle$ had appeared just once, and so the E[L]-landscape is more convoluted, and learning is slower.

A simpler approach that often works better is to pretend that $\langle Q\rangle$ appears just once in the Bellman error. We replace (68) by

$$\begin{aligned} e^Q &= \langle Q\rangle(\boldsymbol{s}_t, \boldsymbol{a}_t) \\ &\quad - r(\boldsymbol{s}_t, \boldsymbol{a}_t) - \gamma^{\Delta t} Q'(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+\Delta t})) \end{aligned} \tag{69}$$

where Q' is a *target network*, separate from $\langle Q\rangle$ but with the same structure: the same numbers of layers and neurons, the same activation functions, and the same *initial* weights and biases.

With (69), $\langle Q \rangle$ appears only once in e^Q, and so it can learn in a supervised way, by backpropagating e^Q through $\langle Q \rangle$. As learning proceeds, we adjust Q', nudging its parameters toward those of $\langle Q \rangle$,

$$\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^{Q'} + \tau(\boldsymbol{\theta}^{\langle Q \rangle} - \boldsymbol{\theta}^{Q'}) \tag{70}$$

where τ is a positive number. Usually τ is small, say 0.001. Otherwise $\langle Q \rangle$ may fail to learn, because its error signal e^Q, in (69), is driving it toward a target function Q' that is changing too quickly.

For the same reason, to avoid driving $\langle Q \rangle$ toward a goal that is changing too quickly, we also use a target network in place of $\boldsymbol{\mu}$ in (69). We create a network $\boldsymbol{\mu}'$, separate from $\boldsymbol{\mu}$ but with the same structure and initialization, and use it to compute e^Q, replacing (69) by

$$\begin{aligned} e^Q = {} & \langle Q \rangle(\boldsymbol{s}_t, \boldsymbol{a}_t) \\ & - r(\boldsymbol{s}_t, \boldsymbol{a}_t) - \gamma^{\Delta t} Q'(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}'(\boldsymbol{s}_{t+\Delta t})) \end{aligned} \tag{71}$$

and updating $\boldsymbol{\mu}'$ by an equation analogous to (70).

Replay buffers

To train $\langle Q \rangle$, we need plenty of examples of e^Q, for many different states and actions. So it is useful to store examples in memory, in a matrix called a *replay buffer*. At each time step, we store a new column vector in the buffer,

$$[\boldsymbol{s}_t; \boldsymbol{a}_t; r(\boldsymbol{s}_t, \boldsymbol{a}_t); \boldsymbol{s}_{t+\Delta t}] \tag{72}$$

Gradually, the buffer fills with many such examples of the data that we need to compute e^Q using (71). At each time step (or more or less frequently), we randomly select a minibatch of examples from the buffer, and use those to compute e^Q and adjust $\boldsymbol{\theta}^{\langle Q \rangle}$, so $\langle Q \rangle$ learns from a large, varied mix of data. When the buffer is full, the agent continues to update it, making room for the newest column vector (72) by removing the oldest one.

Off-policy learning

As time goes by, the agent's policy, $\boldsymbol{\mu}$, changes though learning. Therefore the columns of the replay buffer will contain data generated by a variety of obsolete policies the agent is no longer using. If we select a column (72) from the buffer for our current minibatch, then it will contain an $\boldsymbol{s}_t$ together with an $\boldsymbol{a}_t$ that is probably *not* the action the *current* policy would choose if it were in state $\boldsymbol{s}_t$:

$$\boldsymbol{a}_t \neq \boldsymbol{\mu}(\boldsymbol{s}_t) \tag{73}$$

But a nice feature of e^Q, as defined in (68), (69), and (71), is that it is still useful if $\boldsymbol{a}_t \neq \boldsymbol{\mu}(\boldsymbol{s}_t)$. As we saw in (65) and the text right after it, $Q^{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{a})$ is defined for *any* $\boldsymbol{a}$, not just $\boldsymbol{\mu}(\boldsymbol{s})$, and so we can train $\langle Q \rangle$ using any $\boldsymbol{a}$'s. This type of learning, based on actions that don't fit the current $\boldsymbol{\mu}$, is called *off-policy* learning.

Off-policy learning is convenient because it lets us reuse old data stored in a replay buffer. Some other reinforcement-learning algorithms (not covered here) need *on*-policy data, and therefore can't use replay buffers, at least not without additional operations to try to correct for the policy mismatch.

Rollouts

With off-policy learning, we can in principle learn Q^{μ} with data from any policy at all, but it is better to train on data close to those of the current policy, $\boldsymbol{\mu}$ itself. One reason is that a $\langle Q \rangle$ network can almost never match the true $Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})$ perfectly for all possible inputs $(\boldsymbol{s}, \boldsymbol{a})$. At best, it *approximates* $Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})$, and the approximation will be closer and more useful if its domain is smaller and well chosen, i.e. if we don't try to match $Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})$ over all $(\boldsymbol{s}, \boldsymbol{a})$ but instead focus on the crucial subset, namely states that will actually be visited by an agent using policy $\boldsymbol{\mu}$, and actions that will be chosen by $\boldsymbol{\mu}$ when in those states.

Therefore we train $\langle Q \rangle$ based on data collected during *rollouts*, which are episodes where the agent executes its task using its current policy $\boldsymbol{\mu}$, e.g. if the agent is learning to reach for target objects then each rollout is one reach to one target. This method works well, and also looks very much like biological learning, where an animal learns a task using sense data it collects while practicing that task.

Exploration

So the agent learns $\langle Q \rangle$ based on rollouts using its current policy, $\boldsymbol{\mu}$, but usually it adds a little *noise*, to explore alternative actions that are close to $\boldsymbol{\mu}$'s but possibly better. That is, the actions taken by the agent during rollouts are

$$\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t) + \boldsymbol{v}_t \tag{74}$$

where $\boldsymbol{v}_t$ is a random vector, usually small and with a mean of zero. The *amount* of exploration — the size of the random vectors $\boldsymbol{v}_t$ in (74) — is chosen by trial and error.

Time-optimal control

In the posted sample code, we run DDPG on time-optimal tasks: learning to move various mechanical systems to a target configuration quickly and accurately, as described in Chapter 1. This is a challenging set of tasks, and of practical importance for the brain, which often has to move the eyes, head, or limbs rapidly from one posture to another.

In each task, the environment is a mechanical system with state dynamics $\Delta\boldsymbol{s}_t = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t)$. The dynamics are second-order, like the laws of mechanics. That is, the state vector $\boldsymbol{s}_t$, of dimensionality n_s, consists of two subvectors, the *configuration* $\boldsymbol{q}_t$ and the *velocity* $\boldsymbol{q}^{(1)}{}_t$, each of dimensionality $n_{\boldsymbol{q}} = n_s/2$, and the state dynamics take the form

(75) $$\Delta\boldsymbol{s}_t = \Delta[\boldsymbol{q}_t; \boldsymbol{q}^{(1)}{}_t] = \Delta t\ [\boldsymbol{q}^{(1)}{}_t; \boldsymbol{\alpha}(\boldsymbol{s}_t, \boldsymbol{a}_t)]$$

where the function $\boldsymbol{\alpha}$ is the acceleration. In other words, the current action $\boldsymbol{a}_t$ affects the change in only the second part of the state vector; the change in the first part, $\boldsymbol{q}_t$, is determined by $\boldsymbol{q}^{(1)}{}_t$. To get a varied set of acceleration functions, we compute $\boldsymbol{\alpha}(\boldsymbol{s}_t, \boldsymbol{a}_t)$ using randomly initialized neural nets.

For these tasks, we want actions to be bounded in the range –1 to 1, so in the sample code we set

(76) $$\boldsymbol{a}_t = \tanh(\boldsymbol{\mu}(\boldsymbol{s}_t))$$

That is, the action $\boldsymbol{a}_t$ is not the signal $\boldsymbol{y}\{n_l\}$ in the final layer of the $\boldsymbol{\mu}$ network, but rather the hyperbolic tangent of that signal.

The reward is

$$r_t = -\tanh(\boldsymbol{s}_t^{\mathsf{T}} \boldsymbol{B} \boldsymbol{s}_t) \tag{77}$$

where $\boldsymbol{B}$ is a diagonal matrix of non-negative elements. As we saw in Chapter 1, this form of r function encourages the agent to move quickly to the target state, $\boldsymbol{s} = \boldsymbol{0}$.

Task-relevance

Almost always in the real world, only some aspects of your environment matter for your task. If you are walking through a park to school, you are surrounded by sights and sounds — of birds, squirrels, fluttering leaves — that make little difference to your progress. Your success depends on learning to attend to relevant events and filter out the rest. To recreate this challenge in our sample code, we define the $\boldsymbol{B}$ matrix in (77) so that r_t depends on only some elements of the state $\boldsymbol{s}_t$.

In the sample code DDPG.m, r_t depends only on $\boldsymbol{q}_t$, not on $\boldsymbol{q}^{(1)}{}_t$. We define n_r to be the number of elements of $\boldsymbol{q}_t$ that affect r_t, and then we set the first n_r elements on the diagonal of $\boldsymbol{B}$ to 10, and all other elements to 0.

But even if the *current* reward, r_t, depends on only the first n_r elements of $\boldsymbol{q}_t$, it is still possible that many more elements of $\boldsymbol{q}_t$ and $\boldsymbol{q}^{(1)}{}_t$ affect *future* values of $q_1, q_2, \dots, q_{nr}$, and therefore affect *future* r's and the return, R. We define n_R to be the number of elements of $\boldsymbol{s}_t$ that affect R, and we structure the $\boldsymbol{\alpha}$ net of (75) to ensure that no other state element has any influence on R.

Pseudocode for DDPG

```
for rollout = 1 to $n_{\text{rollouts}}$
  choose a random initial state $\boldsymbol{s}_0$
  for $t$ = 0 to $T$ in steps of $\Delta t$
    $\boldsymbol{a}_t = \tanh(\boldsymbol{\mu}(\boldsymbol{s}_t)) + \boldsymbol{\nu}_t$
    $r_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t)$
    $\boldsymbol{s}_{t+\Delta t} = \boldsymbol{s}_t + \Delta t\, \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t)$
    add $[\boldsymbol{s}_t; \boldsymbol{a}_t; r_t; \boldsymbol{s}_{t+\Delta t}]$ to the buffer
    choose a minibatch $[\boldsymbol{s}_i; \boldsymbol{a}_i; r_i; \boldsymbol{s}'_i]$ from the buffer
    for all $i$ in the minibatch
      $e^Q_i = \langle Q\rangle(\boldsymbol{s}_i, \boldsymbol{a}_i) - r_i - Q'(\boldsymbol{s}'_i, \boldsymbol{\mu}'(\boldsymbol{s}'_i))$
      backprop $e^Q_i$ through $\langle Q\rangle$ to adjust $\boldsymbol{\theta}^{\langle Q\rangle}$
      compute $\langle Q\rangle(\boldsymbol{s}_i, \boldsymbol{a}_i)$
      backprop [1] through $\langle Q\rangle$ to get $\partial\langle Q\rangle/\partial\boldsymbol{a}_i$
      backprop $\partial\langle Q\rangle/\partial\boldsymbol{a}_i$ through $\boldsymbol{\mu}$ to adjust $\boldsymbol{\theta}^{\mu}$
    end
    $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^{Q'} + \tau(\boldsymbol{\theta}^{\langle Q\rangle} - \boldsymbol{\theta}^{Q'})$
    $\boldsymbol{\theta}^{\mu'} \leftarrow \boldsymbol{\theta}^{\mu'} + \tau(\boldsymbol{\theta}^{\mu} - \boldsymbol{\theta}^{\mu'})$
  end
end
```

›› run DDPG.m

Costate equation

Here we consider *episodic*, or in other words *finite-time* or *finite-horizon* tasks, where the aim is to maximize the return R of a movement from time 0 to a finite end time T. In this setting, people often confuse big-R return with little-r reward, so we will use the symbol J for the return, and we will sometimes call it the *objective* — a generic term meaning “the thing we want to optimize”. So we have

(78) $$J = \Delta t \, \Sigma_{t=0 \ldots T} r_t = \Delta t \, \Sigma_{t=0 \ldots T} r(\boldsymbol{s}_t, \boldsymbol{\mu}(\boldsymbol{s}_t))$$

The policy $\boldsymbol{\mu}$ is again a multilayer network with parameters $\boldsymbol{\theta}^{\boldsymbol{\mu}}$, and our aim is to adjust $\boldsymbol{\theta}^{\boldsymbol{\mu}}$ to maximize the average return E[*J*] over some repertoire of motions, e.g. reaches from a variety of initial states.

To make those adjustments, we perform a lot of motions, and after each one we compute the gradient of its return *J* with respect to each of its actions $\boldsymbol{a}_t$, from $t = 0$ to *T*. To find those gradients, we note that $\boldsymbol{a}_t$ can affect *J* in two ways, by altering r_t and $\boldsymbol{s}_{t+\Delta t}$, and therefore by the chain rule,

(79) $$\partial J / \partial \boldsymbol{a}_t = \Delta t \left[\partial r_t / \partial \boldsymbol{a}_t + \partial J / \partial \boldsymbol{s}_{t+\Delta t} \circ \partial \boldsymbol{f} / \partial \boldsymbol{a}_t \right]$$

This formula shows that we need $\partial J/\partial \boldsymbol{s}_{t+\Delta t}$ to get $\partial J/\partial \boldsymbol{a}_t$. To find the $\partial J/\partial \boldsymbol{s}_t$, we again apply the chain rule,

(80) $$\begin{aligned} \partial J / \partial \boldsymbol{s}_t &= \Delta t [\partial r_t / \partial \boldsymbol{s}_t + \partial r_t / \partial \boldsymbol{a}_t \circ \partial \boldsymbol{\mu} / \partial \boldsymbol{s}_t] + \\ &\quad \partial J / \partial \boldsymbol{s}_{t+\Delta t} \circ \mathrm{d}\boldsymbol{s}_{t+\Delta t} / \mathrm{d}\boldsymbol{s}_t \\ &= \Delta t [\partial r_t / \partial \boldsymbol{s}_t + \partial r_t / \partial \boldsymbol{a}_t \circ \partial \boldsymbol{\mu} / \partial \boldsymbol{s}_t] + \\ &\quad \partial J / \partial \boldsymbol{s}_{t+\Delta t} [\boldsymbol{I} + \Delta t (\partial \boldsymbol{f} / \partial \boldsymbol{s}_t + \partial \boldsymbol{f} / \partial \boldsymbol{a}_t \circ \partial \boldsymbol{\mu} / \partial \boldsymbol{s}_t)] \\ &= \partial J / \partial \boldsymbol{s}_{t+\Delta t} + \Delta t [\partial r_t / \partial \boldsymbol{s}_t + \partial r_t / \partial \boldsymbol{a}_t \circ \partial \boldsymbol{\mu} / \partial \boldsymbol{s}_t + \\ &\quad \partial J / \partial \boldsymbol{s}_{t+\Delta t} (\partial \boldsymbol{f} / \partial \boldsymbol{s}_t + \partial \boldsymbol{f} / \partial \boldsymbol{a}_t \circ \partial \boldsymbol{\mu} / \partial \boldsymbol{s}_t)] \end{aligned}$$

Hence if we know (or can estimate) the functions $\boldsymbol{f}$, *r*, and $\boldsymbol{\mu}$, we can compute the final $\partial J/\partial \boldsymbol{s}$,

(81) $$\partial J / \partial \boldsymbol{s}_T = \Delta t [\partial r_T / \partial \boldsymbol{s}_T + \partial r_T / \partial \boldsymbol{a}_T \circ \partial \boldsymbol{\mu} / \partial \boldsymbol{s}_T]$$

and then sweep back in time, using (80) to compute all the $\partial J/\partial \boldsymbol{s}_t$ in turn down to $\partial J/\partial \boldsymbol{s}_{\Delta t}$. The $\partial J/\partial \boldsymbol{s}_t$ are called *costates*, and (80) is the *costate equation*.

We plug these $\partial J/\partial \boldsymbol{s}_t$ into (79) to find the $\partial J/\partial \boldsymbol{a}_t$, and use those derivatives to improve the policy, adjusting $\boldsymbol{\theta}^{\mu}$ down the gradient

(82) $$\partial J/\partial \boldsymbol{\theta}^{\mu} = \Sigma_{t=0\ldots T} \partial J/\partial \boldsymbol{a}_t \circ \partial \boldsymbol{a}_t/\partial \boldsymbol{\theta}^{\mu}$$

Learning *f* and *r*

To apply the costate equation (80), the agent must estimate the state dynamics $\boldsymbol{f}$ and the reward function r. It can learn $\boldsymbol{f}$ by backprop based on the error signal

(83) $$\boldsymbol{e}^{f} = \Delta t \langle \boldsymbol{f} \rangle (\boldsymbol{s}_t, \boldsymbol{a}_t) - \Delta \boldsymbol{s}_t$$

and r based on

(84) $$\boldsymbol{e}^{r} = \langle r \rangle (\boldsymbol{s}_t, \boldsymbol{a}_t) - r_t$$

As in DDPG, we train our networks based on minibatches drawn from a replay buffer.

Training *μ*

We train the policy based on *imaginary* rollouts: starting from a minibatch of initial states drawn from the buffer, we run the rollouts forward in time, computing rewards and new states at each time step using $\langle r \rangle$ and $\langle \boldsymbol{f} \rangle$. (It would be cheating to use minibatches for *real* rollouts, because an agent can't make multiple real movements at once, but for imaginary rollouts, minibatches are allowed). We use (80) and (79), with $\langle \boldsymbol{f} \rangle$ and $\langle r \rangle$ in place of $\boldsymbol{f}$ and r, to compute $\partial J_i/\partial \boldsymbol{a}_{ti}$ for each time point t and each rollout i in the minibatch. With those derivatives we compute $\partial J/\partial \boldsymbol{\theta}^{\mu}$, where J is the sum of all the returns J_i in the minibatch. And we use $\partial J/\partial \boldsymbol{\theta}^{\mu}$ to adjust $\boldsymbol{\mu}$.

Pseudocode for Costate Policy Gradient

```
for rollout = 1 to $n_{\text{rollouts}}$

  // Run a real rollout to gather data for the buffer
  choose a random initial state $\boldsymbol{s}_0$
  for $t = 0$ to $T$ in steps of $\Delta t$
    $\boldsymbol{a}_t = \tanh(\boldsymbol{\mu}(\boldsymbol{s}_t))$
    $r_t = r(\boldsymbol{s}_t, \boldsymbol{a}_t)$
    $\boldsymbol{s}_{t+\Delta t} = \boldsymbol{s}_t + \Delta t\, \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{a}_t)$
    add $[\boldsymbol{s}_t; \boldsymbol{a}_t; r_t; \Delta \boldsymbol{s}_t]$ to the buffer
  end

  // Train $\langle \boldsymbol{f} \rangle$ and $\langle r \rangle$
  choose a minibatch $[\boldsymbol{s}_i; \boldsymbol{a}_i; r_i; \Delta \boldsymbol{s}_i]$ from the buffer
  for all $i$ in the minibatch
    $\boldsymbol{e}^{\boldsymbol{f}}_i = \Delta t \langle \boldsymbol{f} \rangle(\boldsymbol{s}_i, \boldsymbol{a}_i) - \Delta \boldsymbol{s}_i$
    backprop $\boldsymbol{e}^{\boldsymbol{f}}_i$ through $\langle \boldsymbol{f} \rangle$ to adjust $\boldsymbol{\theta}^{\langle \boldsymbol{f} \rangle}$
    $e^r_i = \langle r \rangle(\boldsymbol{s}_i, \boldsymbol{a}_i) - r_i$
    backprop $e^r_i$ through $\langle r \rangle$ to adjust $\boldsymbol{\theta}^{\langle r \rangle}$
  end

  // Train $\boldsymbol{\mu}$ with an imaginary rollout
  choose a minibatch of initial states $\boldsymbol{s}_0$ from the buffer
  for $t = 0$ to $T$ in steps of $\Delta t$
    $\boldsymbol{a}_t = \tanh(\boldsymbol{\mu}(\boldsymbol{s}_t))$
    $\boldsymbol{s}_{t+\Delta t} = \boldsymbol{s}_t + \Delta t \langle \boldsymbol{f} \rangle(\boldsymbol{s}_t, \boldsymbol{a}_t)$
    store $[\boldsymbol{s}_t; \boldsymbol{a}_t]$
  end
  compute $\langle r \rangle(\boldsymbol{s}_T, \boldsymbol{a}_T)$
  backprop through $\langle r \rangle$ to get $\langle \nabla r_T \rangle$
  $\langle \partial J / \partial \boldsymbol{a}_T \rangle = \Delta t \langle \partial r_T / \partial \boldsymbol{a}_T \rangle$
  compute $\langle \partial J / \partial \boldsymbol{s}_T \rangle$ using (81)
  …
  set $\partial J/\partial \boldsymbol{\theta}^{\mu} = \mathbf{0}$
  for $t = T - \Delta t$ to 0 in steps of $-\Delta t$
    compute $\langle r\rangle(\boldsymbol{s}_t, \boldsymbol{a}_t)$
    backprop through $\langle r\rangle$ to get $\langle \nabla r_t\rangle$
    backprop $\langle \partial J/\partial \boldsymbol{s}_{t+\Delta t}\rangle$ through $\langle \boldsymbol{f}\rangle$ to get $\langle \partial J/\partial \boldsymbol{s}_{t+\Delta t}\, \nabla \boldsymbol{f}\rangle$
    compute $\langle \partial J/\partial \boldsymbol{a}_t\rangle$ using (79)
    backprop $\langle \partial J/\partial \boldsymbol{a}_t\rangle$ through $\boldsymbol{\mu}$
    add the result to $\partial J/\partial \boldsymbol{\theta}^{\mu}$
    compute $\langle \partial J/\partial \boldsymbol{s}_t\rangle$ using (80)
  end

  adjust $\boldsymbol{\mu}$ based on $\partial J/\partial \boldsymbol{\theta}^{\mu}$

end
```

›› run COSTATE_POLICY_GRADIENT.m

Other resources

Kirk DE (1998) *Optimal Control Theory: an Introduction*. Dover Publications (*discusses the costate equation, though in continuous time*)

Lillicrap TP et al. (2016) Continuous control with deep reinforcement learning. arXiv:1509:02971 (*presents DDPG*)

Silver D et al. (2014) Deterministic policy gradient algorithms. *Proceedings of the 31st International Conference on Machine Learning. Journal of Machine Learning Research Workshop & Conference Proceedings* 32. (*describes some math underlying DDPG*)

Sutton RS, Barto AG (2018) *Reinforcement Learning: An Introduction*.

5 Regularization

When we learn a function relating the ***x***’s and ***y****’s in a training set of examples (a *sample*), we want that relation to generalize, i.e. we want it to hold true for other ***x***'s and ***y****’s besides those in our sample. It is a serious challenge: in many learning problems it is easy enough to produce a network that predicts ***y****’s from ***x***’s very accurately *when the net has already seen those specific **x**’s and **y***’s during its training*; but faced with a new ***x*** it has not seen before, the net’s accuracy often drops markedly. This situation — good accuracy on the training set but poor on other data — is called *overfitting*. Methods of reducing overfit are called *regularization*.

The central problem is that for any training set, there will always be many different functions that fit the data points in the set equally well but make incompatible predictions about points not in the set. Suppose we have a learning problem where our entire training set $\{(\boldsymbol{x}_t, \boldsymbol{y}^*_t)\}$ consists of just 3 examples: (0.1, 0), (0.2, 0), (0.8, 0), i.e. our ***x***’s take the values 0.1, 0.2, and 0.8, and our ***y****’s are all 0s. Our target function may be the zero function, which takes all ***x***’s to 0. But then again the target may be some other function that happens to equal 0 when ***x*** is 0.1, 0.2, or 0.8 but is far from 0 elsewhere. If we guess wrong then we will make large errors whenever ***x*** is not 0.1, 0.2, or 0.8.

Ensembles

One way of coping is to train several networks — an *ensemble* — and have them vote on answers. We might train 10 networks and then present each novel ***x*** to all

10 of them, taking the average of their predictions as our guess. Mistakes made by the different nets often cancel out to some extent, so their averaged guesses are more accurate than their individual ones.

Capacity

Another coping method is to shrink our network, or otherwise restrict it in some way, to reduce its *capacity*, or in other words the size of its *hypothesis space*, which is the set of all functions that network could ever possibly compute, given any possible setting of its weights and biases. The rationale is that overfitting occurs because our hypothesis space contains functions which are not the desired function but nonetheless fit the training data. If the hypothesis space were smaller then there might be fewer of these false fits. And in practice we can indeed reduce overfitting by using networks with fewer adjustable parameters.

But if we make the hypothesis space too small then it won't contain *any* functions that are a decent match to the desired one. Ideally, we want the smallest hypothesis space that nevertheless contains all the functions we will ever have to learn. For instance, if we know, somehow, that the target function is linear then we might choose a hypothesis space consisting only of linear functions. But we rarely have that much prior knowledge about our targets.

What we can do in practice is monitor our network's performance on both training data and a separate set of non-training examples called the *validation set*, distinct from both the training set and the test set. If our network can't learn to do a good job even on the training data then its capacity is likely too low. If it does much better on training than on validation data then it is over-fitting, and might benefit from a smaller hypothesis space, or from other regularization methods we will discuss in a moment.

Data Augmentation

If we can manage it, the best antidote to overfitting is to get more training data. Consider again the case where we had just 3 training examples, all compatible with the zero function. If we had a million examples instead of just 3, with a million ***x***'s evenly packed over the range from 0 to 1, and the ***y****'s all 0, then we could be more certain that our target really was the zero function, at least over the range [0, 1]. In other words, overfitting is reduced when the training set is large.

Unfortunately, even sets that seem quite big can still be too small to prevent overfitting, e.g. the MNIST database of handwritten digits contains a training set of 60,000 examples, but when we use that set to train a network to recognize handwritten digits, we get serious overfitting. On any big problem we want truly vast numbers of examples.

If we can't get enough examples then it is often useful to make fake ones, provided we take care that our fake examples really do resemble real ones our net may face after its training is done. This technique is called *data augmentation*.

For example the MNIST training set includes about 6000 examples of handwritten "5"s. We can take those

6000 images and tweak each one in various ways, shifting it slightly horizontally or vertically, stretching or compressing it, rotating or shearing it, and so get vast numbers of new examples, all of which are plausible "5"s which a human might actually write. Applying these transformations (for all the digits, not just the "5"s) greatly improves the trained network's generalization. And we can do even better by augmenting even more, e.g. the tweaking I just described included only affine transformations (linear transformations plus shifts), and we can do better by adding certain kinds of nonlinear warping.

It is important that the transformations we use to augment our data produce new examples that are plausible. It would be a bad idea, for instance, to create fake MNIST examples by mirror-reflecting real ones upside-down, because a handwritten "5" reflected upside-down would not resemble any "5" that a real human is likely to write; it would look more like a strange "2". And an upside-down "6" would look like a "9". Used as training data, these fakes would confuse rather than aid the network. Data augmentation should alter our training examples so they resemble the huge set of real variations the net will encounter after its learning is done.

That said, it is often possible to reduce overfitting at least a little by tampering with training data in ways that are not so obviously guaranteed to produce plausible fakes. An example is a method called *dropout*.

Dropout

Here we add noise to a network by randomly silencing many of its hidden neurons, and sometimes input neurons as well. We might stipulate that each hidden neuron has a 50% chance of switching off for any one example. As each example arrives it will evoke a response, and drive learning, in some random set of hidden neurons, about 50% of the total, and the other 50% will have zero activity. Then in the test phase, after training is done, we let all the neurons respond to each example, and we halve the weights in the network to compensate for the doubled numbers of active cells.

This method improves generalization, though the reasons are not entirely clear. It may be a form of ensemble learning: in a sense each example trains a different network, because a different set of neurons is switched on at each moment, and then in the test phase we blend the outputs of all those networks.

Or it may be a kind of augmentation, creating novel training examples by contaminating them with noise. Why might noise help? We can think of it as breaking up spurious regularities. For example, it might happen by sheer coincidence that all our training images of “5”s have a pure black pixel at one consistent locus, and so our network might learn to trust that pixel as an identifier of “5”s. But tested on other “5”s, outside the training set, which don’t have a black pixel at that spot, the network will fail. The likelihood of any one, specific coincidence of this type may be tiny, but there are vast numbers of such coincidences that might arise, and in practice they do exist in all finite training sets. Noise can disrupt these spurious features.

›› run DROPOUT.m

Simplicity

It is often useful to assume that our target functions are simple, in the sense that their graphs are smooth rather than wiggly. If multiple functions in our hypothesis space fit our training data fairly well then we count the smooth, simple functions as more probable than the jagged, complicated ones. In other words, we assume the world is simple, as the philosopher William of Occam urged in the 14th century.

It is not clear why that is a useful assumption, but in practice it does reduce overfitting. Part of the reason may be that data are often distorted by noise, which makes the relation between $\boldsymbol{x}$ and $\boldsymbol{y}^*$ look more complicated than it really is. In other words, complexity can be a figment of noise, and should be viewed with skepticism, so we do well to favor simple solutions. We will see some examples shortly.

Simplicity can be defined in different ways. The most common approach is called *Tikhonov regularization*, where we replace our old, least-squares loss (36) with

$$L = \|\boldsymbol{y} - \boldsymbol{y}^*\|^2 + \lambda \Sigma_l \|\boldsymbol{W}\{l\}\|^2 \quad (85)$$

where $\lambda > 0$ and $\|\boldsymbol{W}\{l\}\|^2$ is the sum of the squared elements of $\boldsymbol{W}\{l\}$. So now we are minimizing not just our approximation errors but also the sum of the squared weights, which is a measure of complexity in the sense that larger weights yield wigglier functions while smaller weights yield flatter ones.

To see the advantages of Tikhonov regularization, run the m-file Occam.m with different settings of its variables *pdt, ns*, *nsd, pda,* and *lam*. Occam defines a target function $\boldsymbol{y}^* = \boldsymbol{\tau}(\boldsymbol{x})$ — different each time it is run — which is a polynomial of degree *pdt*. It chooses *n_ex* examples of $\boldsymbol{x}$'s and $\boldsymbol{y}^*$'s, adds Gaussian noise with standard deviation *nsd* to the $\boldsymbol{y}^*$'s, and presents this corrupted sample to a learning algorithm. The algorithm doesn't know the target function or *pdt,* but finds the polynomial approximation of degree *pda* that minimizes the expected Tikhonov-regularized loss (85) with *lam* playing the role of λ. You will see that when the data are few and noisy, and *pda* is large and *lam* is 0, then your approximations will be poor between the data points, but you can improve them with regularization, either by shrinking *pda* (so as to permit only lower-degree, i.e. simpler, polynomials) or by increasing *lam*.

```
›› run OCCAM.m
```

Occam_multitrial.m runs large numbers of trials like those in Occam.m, to confirm that on average we get better generalization with a simpler hypothesis space or Tikhonov regularization.

```
›› run OCCAM_MULTITRIAL.m
```

Tikhonov regularization is easy to implement in a network. It involves descending the gradient of (85) instead of (36), but the only difference between the two is the second addend in (85), namely $\lambda \Sigma_l \|\boldsymbol{W}\{l\}\|^2$, whose derivative with respect to any one weight, $W\{l\}_{i,j}$, is $2\lambda W\{l\}_{i,j}$, a scalar multiple of the weight itself. To descend this new gradient, then, all we do is add $-2\lambda \boldsymbol{W}\{l\}$ to the gradient computed by backprop, i.e. learn by backprop as usual, and also have each weight shrink slightly toward 0 at each time step, at a rate proportional to its own size. This mechanism is called *weight decay*.

Autoencoders

One way to simplify and regularize networks is to reduce the dimensionalities of the signals they process. Biological data are often very high-dimensional, but they are also usually compressible. For instance, your visual information is coded by the quarter-billion receptor cells (rods and cones) in your retinas, but that information is then processed for transmission to visual cortex on the mere 2.5 million axons of your optic nerves, i.e. the original signal, of a quarter-billion dimensions, is condensed 100-fold. In general, reducing dimensionality lets an agent learn faster and with smaller networks.

One straightforward and perhaps brainlike approach is to use an *autoencoder*, a neural network that creates low-dimensional representations of data by learning to make its output as similar as possible to its input. For instance, we might build an autoencoder to create low-dimensional representations of MNIST images of handwritten digits. Its first layer would have 784 neurons, to represent the greyscales values of the 784 pixels in an MNIST image. The other 4 layers might have neuron counts of 300, 30, 300, and 784 respectively — the crucial thing here is the tight middle layer, or *bottleneck*. If the network learns by backprop to make the activities in its final layer resemble those in its first layer, then that means it has squeezed a lot of information about the original, 784-pixel image through a bottleneck of just 30 neurons.

Why use 5 layers in this example? Why not just 3, with neuron counts of 784, 30, 784? In practice it works better to compress the data in stages, maybe because it breaks a difficult compression problem into easier parts.

›› run AUTOENCODER.m

Other resources

Goodfellow I, Bengio Y, Courville A (2016) *Deep Learning*. MIT Press.

Srivastava N et al. (2014) Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15: 1929–1958

Glossary

action. In reinforcement learning, the agent's output, $\boldsymbol{a}$.

action-value function. Given the current state $\boldsymbol{s}$, an action $\boldsymbol{a}$, and a policy $\boldsymbol{\mu}$, the action-value $Q^{\mu}(\boldsymbol{s}, \boldsymbol{a})$ is the expected return (i.e. the time-integral of the rewards we will accumulate from now to some final time T) if we start in state $\boldsymbol{s}$, we choose action $\boldsymbol{a}$ (which need not agree with policy $\boldsymbol{\mu}$) at the first time point, and from then on we apply policy $\boldsymbol{\mu}$.

adam. a formula for updating network weights and biases based on present and past gradients computed by backprop.

adversarial training. a method where a learning network improves by training on challenging examples provided by a second, "adversary" net. The adversary has full knowledge of the learner's structure and parameters, and uses it to create examples that are likely to fool the learner, i.e. it identifies the learner's weaknesses and provides special training to overcome them.

agent. in reinforcement learning, the entity that performs actions to achieve goals; called the controller in control theory.

Bellman equation. Given any deterministic policy $\boldsymbol{\mu}$, the (one-step) Bellman equation is $Q^{\mu}(\boldsymbol{s}_t, \boldsymbol{a}_t) = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma^{\Delta t}\, \mathrm{E}[Q^{\mu}(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+\Delta t}))]$, where Q^{μ} is the action-value function, γ is the discounting factor, and $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$ are the state and action at time t. The *multistep* or n-*step* Bellman equation is $Q^{\mu}(\boldsymbol{s}_t, \boldsymbol{a}_t) = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \mathrm{E}[\ \Sigma_{i=1..n-1}\, \gamma^{i\Delta t}\, r(\boldsymbol{s}_{t+i\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+i\Delta t})) + \gamma^{n\Delta t}\, Q^{\mu}(\boldsymbol{s}_{t+n\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+n\Delta t}))\ ]$.

Bellman error. an error signal derived from a Bellman equation, used for training estimates of Q^{μ}. For instance, the one-step Bellman Q-equation expresses a property of the Q-function, $Q^{\mu}(\boldsymbol{s}_t, \boldsymbol{a}_t) = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma^{\Delta t}$ $\mathrm{E}[Q^{\mu}(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+\Delta t}))]$, which can be turned into an error signal to train estimates of Q^{μ}, namely $e^Q = \langle Q\rangle(\boldsymbol{s}_t, \boldsymbol{a}_t) - r(\boldsymbol{s}_t, \boldsymbol{a}_t) - \gamma^{\Delta t} \langle Q\rangle(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+\Delta t}))$.

Bellman optimality equation. the Bellman equation for the optimal policy $\boldsymbol{\mu}^*$, e.g. the one-step version is $Q^*(\boldsymbol{s}_t, \boldsymbol{a}_t) = r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma^{\Delta t}\, \mathrm{E}[\min_{\boldsymbol{a}'} Q^*(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{a}')\ ]$, where Q^* is the *optimal* action-value function.

bootstrap. train an estimate of the action-value, Q^{μ}, or a related function such as the return or value, using a Bellman error.

conditioning. Any smooth function is bowl-shaped near its minima, i.e. it approximates a quadratic function, of the form $\alpha + \boldsymbol{\beta}^{\mathsf{T}}\boldsymbol{\theta} + \boldsymbol{\theta}^{\mathsf{T}}\boldsymbol{B}\boldsymbol{\theta}$, where $\boldsymbol{\theta}$ is the function argument, α is a scalar, $\boldsymbol{\beta}$ is a vector, and $\boldsymbol{B}$ is a positive-semidefinite matrix. If the eigenvalues of $\boldsymbol{B}$ differ greatly in size (i.e. if the bowl is much narrower and steeper in some dimensions than in others) then gradient-descent methods take a long time to reach the minimum, because no one learning rate constant η is well suited for both the shallow and the steep dimensions. In that case, the function and the learning problem are said be *poorly conditioned*.

controller. an agent, in the terminology of control theory.

control signal. in control-theory terminology, the output, often called $\boldsymbol{u}$, of a controller (i.e. an agent) — analogous to the action, $\boldsymbol{a}$, in reinforcement learning.

convex optimization. finding the optimum of a convex function, i.e. one with only a single optimum, rather than many local optima. More precisely, a convex function may have multiple optima, but they form a connected set and all have the same value, i.e. none is better than any other. Examples of convex optimization methods are least-mean squares learning (LMS), normalized least-mean squares learning (NLMS), recursive least squares (RLS), and backprop in the final layer of a network. An example of non-convex learning is backprop through a network of 3 or more layers. Convex optimization is usually faster, and easier to prove theorems about, than non-convex optimization.

convolution network. a network where neurons form maps, i.e. each neuron in any one layer l receives input from only a small contiguous patch of neurons in layer $l - 1$, and neighboring neurons in layer l receive their input from neighboring patches; and further, all neurons in any one map share their weights, i.e. their incoming synaptic weights have the same numerical values, arranged in the same spatial pattern within their input fields.

cross-entropy loss. a loss function $L = -\boldsymbol{y}^{*\mathsf{T}} \log(\boldsymbol{y})$ that is useful for networks with *softmax* outputs.

DDPG. The deep deterministic policy gradient algorithm.

decision boundary. Many neural networks receive vectors $\boldsymbol{x}$, belonging to some high-dimensional space X, as inputs (e.g. $\boldsymbol{x}$ might be a set of greyscale values of pixels in an image, and X the space of all such images of that size and shape), and they learn to classify the vectors (e.g. identifying $\boldsymbol{x}$ as an image of a handwritten

"7"). A decision boundary is a hypersurface in X separating one class from another, e.g. separating the $\boldsymbol{x}$-vectors representing "3"s from $\boldsymbol{x}$'s representing "8"s.

deep deterministic policy gradient (DDPG). an algorithm for learning policies in tasks where the state and action spaces are continuous (i.e. the sets of all possible states and actions are continua, not finite sets). The algorithm uses layered networks for the policy and to estimate the action-value function.

deep learning. backprop-based learning in networks of 3 or more layers (or 4 or more according to some authors).

dense network. a network where each neuron in each layer projects to all neurons in the next layer. Also called a fully-connected network.

discounting. In reinforcement learning, we try to maximize the expected return, where the return is the time-integral of discounted future rewards up to some time T, i.e. $R = r_t + \gamma^{\Delta t} r_{t+\Delta t} + \gamma^{2\Delta t} r_{t+2\Delta t} + \ldots$, and γ is a scalar discounting factor between 0 and 1, typically about 0.99. Discounting implies that we care less and less about rewards further and further in the future. One motivation is that we are less and less certain about those future rewards. Another is that discounting helps us avoid infinite returns.

dropout. a method of reducing overfitting by randomly switching off neurons in a network during learning.

early stopping. a method of reducing overfitting by halting learning when performance begins to decline on a validation set.

feedforward network. a neural network with no looping connections, i.e. where no neuron projects to itself directly or via other neurons.

fully-connected network. a dense network.

generative model. a network that learns to output fake but realistic data. Given a set of examples of, say, handwritten numerals, the model learns to produce new images, different from those in the training examples, that look like handwritten numerals, with variants appearing with realistic probabilities, e.g. if 5% of the “7”s in the training examples have crossbars then 5% of the model’s “7”s should also have crossbars.

GPU. the graphical processing unit, a part of a computer, separate from the CPU (central processing unit), designed to handle computations for fast graphical output, e.g. in computer games. GPUs are fast because they perform thousands of operations in parallel, whereas most current CPUs manage only about 8 operations in parallel. Though originally designed for graphics, GPUs are now also used for non-graphical machine-learning operations, such as matrix multiplications, which benefit from parallelization.

gradient noise. In most learning methods, we would like to know the gradient of the expected loss or return, but in fact we have only an estimate of that gradient based on one or a few examples (a minibatch). These estimates will vary around the true gradient depending on the specific examples. This variance of the estimate is called gradient noise.

hard-zero saturation. the fact that some types of artificial neurons have activation functions that are flat over some range of inputs, e.g. relu neurons output 0 whenever their inputs are below zero. As a result, the derivatives that drive learning in the neuron are also **0** when its example inputs are in the flat range, and so the cell cannot learn from those examples.

horizon. In some reinforcement learning tasks, we integrate rewards over only a finite stretch of time, i.e. to a finite final time T. These are called *finite-horizon* (or *episodic*) tasks. Other tasks have an *infinite horizon*, though these often use *discounting*, and then authors may still write loosely of a "horizon", e.g. saying that a small discounting factor γ results in a close horizon. Still other tasks maximize an integral that runs always some finite time into the future, e.g. always 100 ms forward from the current time; these are *receding-horizon* tasks.

KL divergence. Kullback-Leibler divergence, a measure of the similarity of two probability distributions, sometimes used with *softmax* networks, whose outputs can be regarded as probabilities.

L2 loss. the squared error loss function, $L = (\boldsymbol{y} - \boldsymbol{y}^*)^\top(\boldsymbol{y} - \boldsymbol{y}^*)$.

LSTM. long short-term memory, a kind of neuron used in some recurrent nets to control how certain signals persist or fade away as time passes.

maxout unit. a neuron whose activation function consists of multiple affine segments.

mel scale. a series of pitches which humans judge to be equally spaced, used in studies of phoneme recognition.

MNIST. a data set used to train networks to read hand-written numerals, with 60,000 training examples, each a 28-by-28-pixel image of one of the digits “0” through “9”, and additional test examples.

objective function. the thing we want our learning algorithm to optimize, e.g. expected loss or return.

observation. In this course, we assume for simplicity that the agent knows the state, $\boldsymbol{s}$, of the environment at each time point. In reality, an agent may not know $\boldsymbol{s}$ in its entirety, but may instead make a more limited observation $\boldsymbol{o}$ (i.e. gather some sensory or other data) at each time point, where $\boldsymbol{o}$ may not contain enough information to specify $\boldsymbol{s}$.

off-policy training. In reinforcement learning, we often learn the action-value function Q^{μ} of a policy $\boldsymbol{\mu}$ by gradient descent, using the Bellman error $e^Q = \langle Q\rangle(\boldsymbol{s}_t, \boldsymbol{a}_t) - r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \langle Q\rangle(\boldsymbol{s}_{t+\Delta t}, \boldsymbol{\mu}(\boldsymbol{s}_{t+\Delta t}))$. This choice of error makes sense because if the estimate $\langle Q\rangle$ equals the true Q^{μ} then $\mathrm{E}[e^Q] = 0$ on average. Further, that equality holds whether or not $\boldsymbol{a}_t$ equals $\boldsymbol{\mu}(\boldsymbol{s}_t)$, which means we can, if we wish, learn Q^{μ} using $\boldsymbol{a}_t$’s that do not fit the policy $\boldsymbol{\mu}$. If we do use $\boldsymbol{a}_\mathrm{t} \neq \boldsymbol{\mu}(\boldsymbol{s}_\mathrm{t})$, we are training off-policy. Our motivation might be to explore a wider range of actions than those chosen by $\boldsymbol{\mu}$, or to reuse training data $r(\boldsymbol{s}_t, \boldsymbol{a}_t)$ that were sampled in the past, when the agent was using a different policy.

one-hot. a kind of desired output vector for a network, where there is always one element of the vector equal to 1 and the others are all 0s.

overfitting. a situation where a learner improves its accuracy on its training examples, but gets worse on test data, i.e. on data it has not seen during its training.

pooling. Convolution nets often contain pooling layers, which combine data from multiple pixels, discarding some information, e.g. a max-pooling layer might act on a 24-by-24 pixel greyscale image to create a 12-by-12 image where the grey level of each pixel in the smaller image is the maximum over the 4 grey levels of 4 adjacent pixels in the larger image.

prior. prior probability, i.e. the probability of an event judged before some new data arrive, e.g. I might judge it unlikely that it rained last night, but then look out the window and see that the roads are wet; the prior probability that it rained was low, but the posterior probability, after I have looked at the roads, is high. More generally, priors can be any information or assumptions which I build into a network *before* it learns, for regularization e.g. if I have prior knowledge that some function I want to learn is quadratic, then I can design a network whose hypothesis space consists only of quadratic functions.

***Q*-function.** the action-value function.

Q-learning. a method of reinforcement learning where the agent learns the optimal Q-function directly, rather than through a series of policy-specific Q’s, using $e = \langle Q^* \rangle(\boldsymbol{s}, \boldsymbol{a}) - r(\boldsymbol{s}, \boldsymbol{a}) - \max_{\boldsymbol{a}'} \langle Q^* \rangle(\boldsymbol{s}', \boldsymbol{a}')$ or $e = \langle Q^* \rangle(\boldsymbol{s}, \boldsymbol{a}) - r(\boldsymbol{s}, \boldsymbol{a}) - \max_{\boldsymbol{a}'} Q^-(\boldsymbol{s}', \boldsymbol{a}')$, where Q^- is a target function. The agent’s policy is $\boldsymbol{\mu}(\boldsymbol{s}) = \operatorname{argmax}_{\boldsymbol{a}} \langle Q \rangle(\boldsymbol{s}, \boldsymbol{a})$. Hence Q-learning works only in problems where it is possible to compute that argmax efficiently. It is likely not useful for sensorimotor systems in the brain, where the argmax would be hard to compute because $\langle Q \rangle$ is a deep network.

recurrent network. a neural network with looping connections, i.e. where one or more neurons project to themselves, either directly or via other neurons.

regularization. any method used to reduce overfitting.

reinforcement learning. learning based on rewards, as opposed to supervised learning.

relu. rectified linear unit, i.e. a neuron whose activation function is linear (and in fact equals the identity function) except for a cut-off at zero — $y\{l\}_i = \max(0, v\{l\}_i)$.

replay buffer. In reinforcement learning there is often a network that learns a function (such as the action-value) based on training examples showing how various actions in various situations yielded specific rewards. In some methods, the control system uses these training data — states, actions, rewards — as it collects them, e.g. it might carry out a reaching movement, collect the current state, action, and reward at each moment as the reach proceeds, immediately use those data to adjust its network parameters, and then discard the data to make room for the next state, action, and reward. A problem with this approach is that, during a smooth movement like a reach, the successive states are very close together, and so the learner gets a series of very similar training examples. These examples may push the learner's parameters toward values that are optimal for that small region of state space but not very good overall. We can avoid that problem by storing many training examples in a replay buffer, and adjusting parameters at each time step based not on the *current* example but on a minibatch of training data from many different times and reaches, drawn randomly from the buffer.

return. time-integral of rewards.

reward. Any reinforcement learning task begins with a reward function $r(\boldsymbol{s}, \boldsymbol{a})$ which defines what the agent is trying to achieve. Specifically, the agent's aim is to maximize the *return*, which is a time-integral of the reward.

SARSA. a method of reinforcement learning, identical to Q-learning except that the agent learns using $e = \langle Q\rangle(\boldsymbol{s}, \boldsymbol{a}) - r(\boldsymbol{s}, \boldsymbol{a}) - \langle Q\rangle(\boldsymbol{s}', \boldsymbol{a}')$ or $e = \langle Q\rangle(\boldsymbol{s}, \boldsymbol{a}) - r(\boldsymbol{s}, \boldsymbol{a}) - Q^{-}(\boldsymbol{s}', \boldsymbol{a}')$, where the next-step $\boldsymbol{s}'$ and $\boldsymbol{a}'$ are both sampled, and Q^{-} is a target function. So SARSA, unlike Q-learning, doesn't compute $\text{argmax}_{\boldsymbol{a}'}\langle Q\rangle$ *for its learning*, but it does still use the policy $\boldsymbol{\mu}(\boldsymbol{s}) = \text{argmax}_{\boldsymbol{a}}\langle Q\rangle(\boldsymbol{s}, \boldsymbol{a})$. It is called SARSA because it computes its e based on a state, action, reward, and the subsequent state and action.

self play. training agents by having them compete against other agents, usually in a simulated environment.

SIFT. the scale-invariant feature transform, a computer-vision algorithm that detects certain image features that are useful for object recognition. These features were designed by computer scientists, whereas convolution nets instead *learn* useful features by backprop.

softmax. When training a one-hot network, it is often helpful to let the final-layer neurons be affine (even if the upstream neurons are, say, relu or tanh), and then define the net's output $\boldsymbol{y}$ to be a softmax function of the final-layer signal $\boldsymbol{y}\{n_l\}$, i.e. define the i-th element of $\boldsymbol{y}$ to be $y^i = \exp(y\{n_l\}^i)/\Sigma_j \exp(y\{n_l\}^j)$. This formula ensures that $\boldsymbol{y}$'s elements are all between 0 and 1, and sum to 1, like the elements of a one-hot $\boldsymbol{y}^*$ (and also like probabilities, so we can regard each element y^i of the softmax output as the network's estimate of the probability that the input belongs to class i).

state. roughly, a vector which contains maximal information about the future. 1) In a deterministic environment, the state is a vector $\boldsymbol{s}$ which contains enough information about the world that, given $\boldsymbol{s}$ at just one time t, and the actions at time t and all subsequent time points, it is possible in principle to predict the whole future evolution of $\boldsymbol{s}$. 2) In a stochastic environment, the state contains enough information that, given $\boldsymbol{s}$ at time t, and the actions at time t and subsequently, we can compute the probabilities of all future $\boldsymbol{s}$, and those probabilities would not be altered if we had any additional information from time t or earlier.

stochastic gradient descent. online gradient-descent learning where parameters are adjusted after each single example or minibatch, using a gradient estimate based only on that example or minibatch.

supervised learning. a method where a learner receives input vectors $\boldsymbol{x}$, computes outputs $\boldsymbol{y}$, and gets error feedback $\boldsymbol{e}$ that specifies the difference between its output and some *desired* output $\boldsymbol{y}^*$, e.g. $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{y}^*$. The learner adjusts its parameters to reduce some non-negative scalar function of $\boldsymbol{e}$, such as the squared error $\boldsymbol{e}^\mathsf{T}\boldsymbol{e}$.

tanh. the hyperbolic tangent function.

target function. 1) the function $\boldsymbol{\tau}$ a network is trying to learn. 2) a function such as Q^- in *DDPG*, used to simplify gradient descent.

***u*.** in control theory, the control signal, analogous to the action, $\boldsymbol{a}$, in reinforcement learning.

unsupervised learning. any method where a learner adjusts at least some of its parameters based on a criterion that does not involve its output error or reward, e.g. it might look for a way to compress its input data into a lower-dimensional representation before worrying about reducing its output errors or maximizing its returns.

validation set. To combat overfitting, we may withhold a subset of the training data called the validation set, so the learner cannot use those examples to adjust its parameters. We use the validation set to monitor for overfitting, periodically testing the learner on those withheld data to see whether its performance is getting worse. If it does begin to worsen on the validation set then we may halt its learning, even though it is still improving on the non-withheld training data, to avoid overfitting — a method called early stopping.

value function. The value $V^{\boldsymbol{\mu}}(\boldsymbol{s})$ of a state $\boldsymbol{s}$, given a policy $\boldsymbol{\mu}$ and a reward function r, is the expected time-integral of $r(\boldsymbol{s}, \boldsymbol{a})$, possibly discounted, from now to some final time T, if we start in state $\boldsymbol{s}$ and use policy $\boldsymbol{\mu}$. One way to state the aim of reinforcement learning is that it tries to find policies that maximize the value $V^{\boldsymbol{\mu}}(\boldsymbol{s})$ at every state $\boldsymbol{s}$.

whitening. applying a linear transformation $\boldsymbol{T}$ to a random vector $\boldsymbol{v}$ so that the covariance matrix of $\boldsymbol{w} = \boldsymbol{Tv}$ is the identity matrix, i.e. the elements are $\boldsymbol{w}$ are uncorrelated with each other, and each has variance 1. As a result, even if the vectors $\boldsymbol{v}$ form an elongated, rotated cloud, the $\boldsymbol{w}$'s will form a spherical ball. If $\boldsymbol{v}$ has dimensionality > 1 then there are many different linear transformations that will achieve whitening, because if $\boldsymbol{T}$ works then so does $\boldsymbol{OT}$, where $\boldsymbol{O}$ is any orthogonal matrix (because rotating a ball of $\boldsymbol{w}$ vectors leaves it in the form of a ball). Whitening input vectors, or vector signals between layers in a deep net, can speed up learning by improving *conditioning*.

ZCA whitening. zero-phase component analysis whitening, a form of whitening where the linear transformation $\boldsymbol{T}$ is the square root of the covariance of $\boldsymbol{v}$. It is the version of whitening that yields basis vectors as close as possible to the basis vectors originally used to represent $\boldsymbol{v}$.