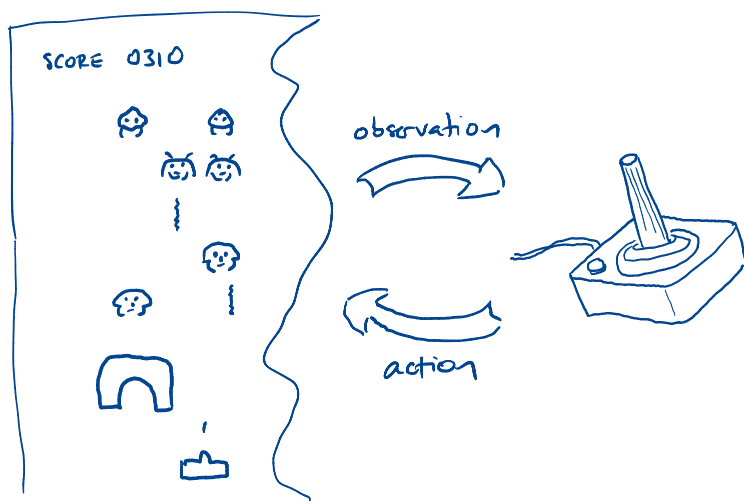
Reinforcement learning

Sequential decisions

A lot of machine learning problems are *one-shot*: we get some information, we make a prediction, and then we're done. Maybe we repeat the process later, but it's OK to forget everything about the circumstances surrounding one prediction before we make the next.

In *reinforcement learning*, we are interested in *sequential decisions*: each choice we make has consequences for the future. In a sequential decision problem, our world looks like this:



There's an external environment that provides us with observations — for example, images from a video game that we're playing. After each observation we get to choose an action — for example, we might move a joystick or press a button. The result is an alternating sequence of observations and actions, called a *trajectory*.

The environment typically has *internal state*: past events affect future possibilities. For example, if we use the joystick to move our spaceship, we might be able to hide under a bunker and stay safe for now —

but we won't be able to shoot aliens until we venture out from shelter. The *initial state* is part of the problem definition: the first state is selected from a fixed distribution.

We might or might not know the environment's internal state. In Space Invaders, everything we need to know is always onscreen: the internal state is *fully observable*. In other games (such as Starcraft), a lot of interesting stuff that happens isn't recorded anywhere, and we need to remember it: the state is *partially observable*. We'll say more about partially observable environments later.

Eventually the game might end, resulting in a finite-length trajectory. The maximum length of a trajectory is called the *horizon*. In some environments, a trajectory can go on forever: these are *infinite-horizon* problems.

Objectives

Some trajectories are good, and others are not so good. In Space Invaders, we might get a higher or lower score; in chess or go, we might win or lose the game; with a physical robot, we might bring someone a cup of coffee or spill it on them.

We measure how good a trajectory is using a *cost* or *reward* function. (The only difference between the two is whether we want to minimize or maximize.) Sometimes we might have a different form of objective, like *imitation* (do something that's similar to a given example), but that's outside the scope of these notes.

Write a trajectory as

$$\tau = (o_1, a_1, o_2, a_2, \ldots, o_H, a_H)$$

where o_t is an observation, a_t is an action, and H is the

(finite) horizon. For definiteness, let's say that we are minimizing a cost function. We'll assume that our cost takes the form of a sum over time steps: at each step we have an *instantaneous*, *immediate*, or *one-step* cost $c(o_h, a_h)$, and our total cost is

$$c(\tau) = \sum_{h=1}^{H} c(o_h, a_h)$$

This assumption is without loss of generality: we can always include extra information in the environment's state to handle other forms of cost functions.

Policy

Now that we have a cost function, we want to decide how to act in order to minimize it. We can represent our plan of action as a *policy*: a function that looks at our history of actions and observations, and tells us a distribution over our next action.

Each policy determines a distribution over trajectories: our policy determines action choices, and the environment determines initial state, observations, and state changes. We write P_π for this distribution, and $\mathbb{E}_\pi$ for the expectation under this distribution. So, our objective is to minimize

$$J(\pi) = \mathbb{E}_\pi(c(\tau)) = \sum_{\tau} P_\pi(\tau) c(\tau)$$

where the sum runs over all possible trajectories τ.

There are a few different kinds of policies. Maybe the simplest one is an *open-loop policy*, which just tells us some predetermined actions to take, with no dependence on the observations we receive. This is in contrast to a *closed-loop policy*, which does respond to observations. (These phrases refer to the feedback loop between observations and actions, which can be

either open (interrupted) or closed (allowing feedback).)

A policy might or might not have its own internal storage. If it doesn't, it's called *stateless* or *memoryless*. A memoryless policy uses only the most recent observation to determine the next action. If this most recent observation isn't very informative, then we need a policy that does use internal storage, called *stateful*.

We can distinguish between *deterministic* and *randomized* policies. We can also distinguish between *time-varying* and *time-invariant* policies, also called *nonstationary* or *stationary*.

Often we'll consider a *family* or *class* of policies, indexed by some parameter vector θ. We can write

$$P(a_h \mid \tau_h, \theta) = \pi_\theta(a_h \mid \tau_h)$$

where $\tau_h = (o_1, a_1, \ldots, o_h)$ is the part of the trajectory up to the observation at time step h. In this case we can use either θ or π_θ to refer to the policy: e.g., $J(\theta)$ or $\mathbb{E}_\theta(\ldots)$.

RL and generalization

In supervised learning, we have a training set and a test set sampled i.i.d. from the same distribution. Generalization means that we get good test set performance, and overfitting means that our test set performance is worse than we would predict by looking at our training set performance.

In RL the situation is a bit more complicated: there isn't just a single distribution involved. Instead, every policy leads to its own distribution; so, we might look at trajectories sampled from many different distributions over the course of training.

Generalization then means that our final policy π performs well: the observation-action pairs that we visit under P_π have low cost,

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{h=1}^{H} c(o_h, a_h) \right]$$

We might never have sampled from exactly the distribution P_π during training! Overfitting is therefore a much larger concern in RL than it is in supervised learning: especially if we have limited data, P_π might cover (o_h, a_h) pairs that we visited only rarely during training, and so the performance of π might be hard to predict accurately.

Model-free, model-based, and in between

Depending on our environment, we might know a lot or a little about how the world works. In some environments we know perfectly how to predict the future given past actions and observations: for example, in chess we know where the pieces start and how they move. This kind of predictor is called a *model* or an *environment model* or a *world model*, and if we've got one, we're solving a *model-based* problem.

Models can be stochastic: for example, in backgammon there are dice, but we still know how the pieces move given the die rolls. So, we still say that we have a perfect model, since we can perfectly describe the *distribution* of future states.

At the other end of the spectrum, we might only be able to learn about the world by doing physical experiments. For example, if we're trying to learn to grade a hill with a backhoe, it's really hard to write an accurate model; our best bet might be to experiment with real physical dirt. We say that environments like this are *model-free*.

There's a whole spectrum in between model-based and model-free environments. One common situation is to have a *simulation-only* model: we can't calculate the distribution over future events, but we can sample from it. Video games often fall in this category: we can run the video game code and conduct cheap experiments before we say "OK, this counts" and try for a high score. A wrinkle within simulations is whether simulation states are *reuseable*: can we save an environment state we've visited and restart the simulation from there?

Another common situation is that we have *both* a real environment *and* a simulator of it. For example, we could have a physical self-driving car, as well as a simulator that is supposed to behave similarly. We can try to use the simulator to keep from having to do as many experiments in the real world. But, we have to worry about *model errors* in the simulator: if we depend too heavily on the simulator, our optimized policy might not work well in the real world. (This is often called the *sim2real transfer* problem.)

If we don't start off with an environment model, we can try to learn one. Or, if we start with a simulator, we can try to learn a full model that lets us predict probabilities rather than just sample. Learning a model can be tricky, especially in partially observable environments: our learner has to come up with a representation of states as well as functions that update this representation and make predictions. If we do manage to learn, we again have to contend with model error.

Model-free and model-based algorithms

We'll meet a variety of learning algorithms below, each of which has different requirements and assumptions. Algorithms that work without a model are called *model-free*. Algorithms that need a model are called

model-based. We can still potentially use model-based algorithms in a model-free environment: we just have to learn an accurate-enough model.

Model-free algorithms seem more generally applicable, so why would we use a model-based algorithm? The problem with model-free algorithms is that they can take *lots* of training data: billions of time-steps is not uncommon. We typically can't afford that much experience in a real physical system: it would be way too time-consuming, but even more importantly, it would in many cases be very unsafe. (We don't want to crash thousands or millions of self-driving cars...) A model-based algorithm has the potential to do thought-experiments, potentially avoiding many real-world experiments.

Some researchers are reluctant to say that fully model-based algorithms are learning at all: after all, we already know everything relevant about the world before we start. In this view, fully-model based algorithms are more akin to planning methods than to learning methods — we say they are doing *decision-theoretic planning*.

On the other hand, if we just know the environment model and the goal, that knowledge is not in an *effective* form: it doesn't let us efficiently decide how to act. So it also makes sense to view model-based RL algorithms as doing *speed-up learning*: we use learning techniques to convert knowledge into a more effective and applicable form. I personally prefer this view, since the techniques needed are similar to other learning problems: e.g., drawing training examples, fitting a function to data, minimizing a loss with SGD.

Policy gradient

Let's start with the simplest possible reinforcement learning algorithm, the *policy gradient method*. Policy gradient is a model-free method, and it can work either from a simulator or directly from a real environment — although it may be prohibitively expensive in the latter case.

Given a policy class with parameters θ, $\pi_\theta(a_h \mid \tau_h)$, we would like to minimize the total expected cost of an entire trajectory

$$J(\theta) = \mathbb{E}_\theta(c(\tau))$$

To minimize this cost, we can run stochastic gradient descent:

$$\theta_{t+1} = \theta_t - \eta g_t$$

where t indexes over iterations of SGD, η is a learning rate, and g_t is a stochastic estimate of the gradient, satisfying

$$\mathbb{E}_{\theta_t}(g_t) = \frac{d}{d\theta} J(\theta)\bigg|_{\theta_t}$$

The key insight for policy gradient is that we can compute a suitable stochastic gradient estimate g_t just from a single trajectory τ. The only assumption we need is that τ was collected by following $\pi_t = \pi_{\theta_t}$: surprisingly, we don't need an environment model, even though the total cost $J(\theta)$ clearly depends on the environment as well as the policy.

Policy gradient theorem

To see why, we can take advantage of a really useful identity: for any function f,

$$\frac{d}{dx} f(x) = f(x) \frac{d}{dx} \ln f(x)$$

(To derive this identity, use the chain rule on $\frac{d}{dx} \ln f(x)$.)

This identity is helpful if the definition of f contains a product, since it replaces the derivative of a product (which is annoying) with the derivative of a sum (which is much nicer).

To use this identity, we can expand out the expectation over trajectories:

$$\frac{d}{d\theta}\mathbb{E}_\theta(c(\tau)) = \frac{d}{d\theta}\sum_\tau P_\theta(\tau)c(\tau) = \sum_\tau c(\tau)P_\theta(\tau)\frac{d}{d\theta}\ln P_\theta(\tau)$$

This helps us because $P_\theta(\tau)$ is a product over time steps: the probability of $\tau = (o_1, a_1, \ldots, o_H, a_H)$ is

$$P_\theta(\tau) = \prod_{h=1}^{H} P_\theta(a_h \mid \tau_h)\, P(o_{h+1} \mid \tau_h, a_h)$$

So,

$$\frac{d}{d\theta}\ln P_\theta(\tau) = \sum_{h=1}^{H}\frac{d}{d\theta}\ln P_\theta(a_h \mid \tau_h)$$

Note what happened here: the effect of the environment vanished, since $P(o_{h+1} \mid \tau_h, a_h)$ doesn't depend on our policy!

The remaining gradient term depends only on our policy class:

$$P_\theta(a_h \mid \tau_h) = \pi_\theta(a_h \mid \tau_h)$$

Typically, we would choose a policy class that lets us compute the gradient of $\ln \pi_\theta(a_h \mid \tau_h)$ analytically. For example, we could use a neural network whose last layer has one node per action, with a softmax activation. (The input to the network could be any representation of τ_h.) In this case we can use backprop to compute $\frac{d}{d\theta}\ln \pi_\theta(a_h \mid \tau_h)$.

To simplify notation, define the *action score*

$$g_\theta(a, \tau_h) = \frac{d}{d\theta}\ln \pi_\theta(a \mid \tau_h)$$

for all actions a. Now, putting everything together,

$$\frac{d}{d\theta}J(\theta) = \sum_{\tau} c(\tau)P_\theta(\tau)\sum_{h=1}^{H} g_\theta(a_h, \tau_h)$$

This has the form of an expectation over trajectories:

$$\frac{d}{d\theta}J(\theta) = E_\theta\left[c(\tau)\sum_{h=1}^{H} g_\theta(a_h, \tau_h)\right]$$

The above expression is one form of the *policy gradient theorem*: it enables a whole family of gradient-based RL methods, including the one that we are in the process of deriving.

In particular, given a trajectory τ sampled according to π_θ, we can compute a gradient sample as

$$g = c(\tau)\sum_{h=1}^{H} g_\theta(a_h, \tau_h)$$

which will have expectation

$$\mathbb{E}_\theta(g) = \frac{d}{d\theta}J(\theta)$$

A simplification

One last simplification comes from a useful property of the action scores: given any partial trajectory τ_h, the expected score of the next action is zero,

$$\mathbb{E}_\theta(g_\theta(a, \tau_h) \mid \tau_h) = 0$$

To see why, differentiate both sides of the identity $\sum_a \pi_\theta(a \mid \tau_h) = 1$ to get

$$\sum_a \frac{d}{d\theta}\pi_\theta(a \mid \tau_h) = 0$$

and use $\frac{d}{dx}f(x) = f(x)\frac{d}{dx}\ln f(x)$ to get

$$\sum_a \pi_\theta(a \mid \tau_h)\, g_\theta(a, \tau_h) = 0$$

which is equivalent to the above identity.

We can use this identity to simplify each of the terms in the sum inside the expression for $\frac{dJ}{d\theta}$: consider a single term,

$$\mathbb{E}_\theta\left[c(\tau)\, g_\theta(a_h, \tau_h)\right]$$

By the law of iterated expectations, this is equivalent to

$$\mathbb{E}_\theta[\,\mathbb{E}_\theta[c(\tau)\, g_\theta(a_h, \tau_h) \mid \tau_h]]$$

Now we can split up $c(\tau)$ into two parts, corresponding to time steps $1 \ldots h-1$ and $h \ldots H$:

$$c(\tau) = \sum_{j=1}^{H} c(o_j, a_j) = \sum_{j=1}^{h-1} c(o_j, a_j) + \sum_{j=h}^{H} c(o_j, a_j)$$

The first sum is constant given τ_h. So, we can factor it out of the inner expectation, and since $\mathbb{E}_\theta(g_\theta(a_h, \tau_h) \mid \tau_h) = 0$, that means that we can drop this part of the expression. That leaves

$$\mathbb{E}_\theta\left[\mathbb{E}_\theta\left[\sum_{j=h}^{H} c(o_j, a_j) g_\theta(a_h, \tau_h) \,\middle|\, \tau_h\right]\right]$$

We can use the law of iterated expectations now in the opposite direction to remove the inner expectation:

$$\mathbb{E}_\theta\left[\sum_{j=h}^{H} c(o_j, a_j) g_\theta(a_h, \tau_h)\right]$$

Substituting back in, we now have our final expression for the policy gradient:

$$\frac{d}{d\theta} J(\theta) = E_\theta\left[\sum_{h=1}^{H}\left[\sum_{j=h}^{H} c(o_j, a_j)\right] g_\theta(a_h, \tau_h)\right]$$

The above expression is our second form of the policy gradient theorem. It is somewhat sharper than the first form: by removing a term that has zero expectation but

nonzero variance, we have reduced the variance of our gradient estimates.

In particular, given a trajectory τ sampled according to π_θ, we can compute a gradient sample as

$$g = \sum_{h=1}^{H} \left[\sum_{j=h}^{H} c(o_j, a_j) \right] g_\theta(a_h, \tau_h)$$

which will have the correct expectation

$$\mathbb{E}_\theta(g) = \frac{d}{d\theta} J(\theta)$$

but lower variance than if we had used the first form of the policy gradient theorem.

REINFORCE

We are finally ready to summarize the overall policy gradient algorithm: it is

- Initialize policy with parameters θ_1.
- Repeat for $t = 1, 2, \ldots$:
    - Sample a trajectory τ by following policy π_t with parameters θ_t.
    - Use the above expression to calculate a gradient sample g_t by summing over time steps h.
    - Update $\theta_{t+1} = \theta_t - \eta g_t$.

This basic policy gradient algorithm is also called REINFORCE; it was introduced in the 1980s, and its descendants are still popular today.

Example

Let's work out policy gradient on an incredibly simple environment. This environment has fully observable, discrete states numbered $1 \ldots 5$. It has two actions, R

and L, which increment or decrement the state, saturating at the ends of the line. We start at state 4, in the middle of the line. The immediate cost is 1 per step, *except* for going L from state 1, which costs 0. Let's say our planning horizon is $H = 5$.

Sketch:

We can use a simple memoryless policy class:

$$P(R) = \frac{1}{1 + e^{-\theta}}$$

This makes our action scores be:

$$\frac{d}{d\theta} \ln P(R) =$$

$$\frac{d}{d\theta} \ln P(L) =$$

If our policy is $\theta = 0$, we random-walk until our time horizon expires. For example:

$$\tau = (3, L, 2, R, 3, R, 4, R, 5)$$

The contribution to the policy gradient from a single step is:

So the total is:

We update by moving along the negative gradient, say to

A good exercise would be to compute the policy gradient (either for $\theta = 0$ or for the new value of θ) if we now see another trajectory:

$$\tau = (3, L, 2, L, 1, L, 1, R, 2)$$

Value functions

Let's look a little more closely at one of the terms in $\frac{d}{d\theta} J(\theta)$. Using the law of iterated expectations as before, we can express it as

$$E_\theta \left[\left[\sum_{j=h}^{H} c(o_j, a_j) \right] g_\theta(a_h, \tau_h) \,\middle|\, \tau_h \right]$$

We can use the law of total expectation to split the expectation up into one term for each action a that we could execute at time step h:

$$\sum_a \pi_\theta(a \mid \tau_h)\, E_\theta \left[\left[\sum_{j=h}^{H} c(o_j, a_j) \right] g_\theta(a, \tau_h) \,\middle|\, \tau_h, a_h = a \right]$$

Given τ_h, $g_\theta(a, \tau_h)$ is a constant, so we can pull it out of the expectation:

$$\sum_a \pi_\theta(a \mid \tau_h) g_\theta(a, \tau_h)\, E_\theta\left[\sum_{j=h}^{H} c(o_j, a_j) \middle| \tau_h, a_h = a\right]$$

What remains inside the expectation is interesting, so let's give it a name:

$$Q_\theta(\tau_h, a) = E_\theta\left[\sum_{j=h}^{H} c(o_j, a_j) \middle| \tau_h, a_h = a\right]$$

In words, $Q_\theta(\tau_h, a)$ is the expected total future cost, starting at step h, given that we have observed a partial trajectory τ_h so far, given that we execute action a at step h, and given that we execute policy π_θ in the future, from step $h+1$ to H.

Q_θ is called the *action-value function* for policy π_θ, and it is a fundamental quantity in reinforcement learning. With it, we can

- Compare two (τ_h, a) pairs to see which one we are happier with (which one leads to lower expected cost in the future)
- Do lookahead search: Q_θ can play the same role as a heuristic in A^*, since it lets us compare the different places (different (τ_h, a) pairs) where we might end up
- Reduce the variance of our policy gradient estimate — more on this later
- Write down a third form of the policy-gradient theorem: plugging our definition of Q_θ to the previous expression for the policy gradient, we get

$$\frac{d}{d\theta} J(\theta) = E_\theta\left[\sum_{h=1}^{H} \sum_a \pi_\theta(a \mid \tau_h) g_\theta(a, \tau_h)\, Q_\theta(a, \tau_h)\right]$$

- Interpret the policy gradient update: from the

above equation, if there are two actions a and a' that we could take given the current partial trajectory τ_h, and if $Q_\theta(a, \tau_h) \gg Q_\theta(a', \tau_h)$, then we try to increment the log-probability of a given τ_h. Since we're working with log-probabilities, this means that we multiply the probability of a by a factor bigger than 1, and renormalize.

Unfortunately, the new form of the policy-gradient theorem is not applicable directly to REINFORCE, since we typically don't know the action-value function. But we will see below some algorithms that use estimates of Q_θ to try to reduce variance and speed up convergence.

We can summarize the action-value function into a *state-value function*, also sometimes just called a *value function*:

$$V_\theta(\tau_h) = \mathbb{E}_\theta[Q_\theta(a, \tau_h)]$$

That is, V_θ is the expectation of Q_θ, using actions selected from the policy π_θ. The state-value function has many of the same uses as the action-value function, and so many RL algorithms can be expressed in terms of either one.

Infinite horizon and discounting

There are a few simple variations of REINFORCE that are worth knowing about. The first is for infinite-horizon problems; the math in this case is almost the same.

In an infinite-horizon problem, it no longer works to represent our cost as a sum over time steps: the infinite sum might not converge. Instead we will minimize total *discounted* cost:

$$\sum_{t=1}^{T} \gamma^{t-1} c(o_t, a_t)$$

where $\gamma \in [0, 1)$ is called a *discount factor*. The geometric sum formula implies that the total discounted cost is always bounded.

Intuitively, discounting means that we care more about present costs than future costs. In fact, one common interpretation of a discount factor is as an interest rate: if costs are in dollars, and if our bank account pays us interest by multiplying our balance by a factor $\frac{1}{\gamma}$ on each time step, then a present cost of one dollar is in fact exactly equivalent to a cost of γ^t dollars in t time steps.

I personally think the interest rate interpretation isn't very helpful: we are almost never in the position of earning interest on our costs or rewards. Instead, I think it's more intuitive to think of γ as representing uncertainty. Our environment models are typically flawed, and we should expect something completely unexpected to happen every so often. If we have a probability of $1 - \gamma$ of something unexpected on each time step, then the total discounted cost is exactly the cost we expect to incur before something unexpected happens. We don't want to plan about costs after that, since our models will be wrong, making our plans irrelevant.

The policy gradient in an infinite-horizon problem satisfies almost exactly the same expression as in a finite-horizon problem:

$$\frac{d}{d\theta} J(\theta) = E_\theta \left[\sum_a \pi_\theta(a \mid \tau_h) g_\theta(a, \tau_h)\, Q_\theta(a, \tau_h) \right]$$

We just need to redefine Q_θ for the discounted case:

$$Q_\theta(\tau_h, a) = E_\theta\left[\sum_{j=h}^{\infty} \gamma^{j-1} c(o_j, a_j) \,\middle|\, \tau_h, a_h = a\right]$$

Note that the above expression means that the tail end of a long trajectory barely contributes to the policy gradient. It's a common heuristic to scale up this contribution by redefining

$$Q_\theta(\tau_h, a) = E_\theta\left[\sum_{j=h}^{\infty} \gamma^{j-h} c(o_j, a_j) \,\middle|\, \tau_h, a_h = a\right]$$

i.e., by discounting only from the current step rather than from the beginning of time. In principle this heuristic can cause the policy gradient method to fail to converge. But in practice it seems to be beneficial, since it lets us squeeze more information out of the later parts of a long trajectory.

Partial observability

A second variation is whether the environment has full or partial observability. Our example had full observability, which meant that our policy only needed to depend on the current state. But, REINFORCE works equally well in a partially observable environment, so long as we use an appropriate policy class.

In a partially observable environment, the optimal action distribution can depend on all past observations, not just the most recent one. So, our policy has to have memory. The simplest sort of memory is to remember the entire history, and pass it to our policy using a variable-sized network such as an LSTM.

Or, we can compress the history down to a summary statistic. These two approaches aren't necessarily distinct: a model like an LSTM will effectively learn a compression to fit all relevant history information into a

fixed-dimensional recurrent activation vector.

The ideal summary of history is

- rich enough to allow us to act (approximately) optimally, and
- able to be updated locally: given the statistic at time h, an action a_h, and an observation o_h, we can compute the statistic at the next time step $h+1$.

A statistic with these properties is called an (approximate) *state* or *state representation*. The ability for local updates is called the *Markov property*.

If we can find a good state representation, we can update most of our RL definitions to depend on the state rather than the entire history. For example, if s is our state, we can write

$$\pi_\theta(a \mid s) \qquad Q_\theta(s, a) \qquad V_\theta(s)$$

for the policy, action-value function, and state-value function.

Our remembered state isn't necessarily the the same as the environment's internal state. For example, we might get noisy readouts of a robot's position. In this case we can distinguish

- the most recent readout: this is one of our observations
- our best estimate of the robot's position, based on all past readouts: this is our remembered state
- the robot's true underlying position: we never know this, so it's mostly of philosophical interest, but we can think of it as the internal state of the environment.

Computing or discovering a good state representation is an important learning problem. One class of

methods tries to discover a state implicitly — e.g., as the latent activations of a recurrent neural network, as described above. These methods can work in either a model-free or a model-based setup.

Another class of methods requires an environment model, which can be either given or learned. We explicitly maintain a distribution over the possible internal states of the environment, called a *belief state*. We start off with a prior distribution. Each observation gives us a likelihood, which we multiply into our distribution and renormalize to get a posterior. A belief state is guaranteed to be a state — in particular, it satisfies the Markov property.

Once we have a state representation — either as a sequence or as a fixed-dimensional vector — we can pass it as input to our policy class. REINFORCE then works exactly as it does for fully-observable environments.

In fact, the distinction between fully and partially observable is quite blurry: if our policy class ignores any information about the state, then we might as well not have passed in this information in the first place. And, in any realistic environment, we have to use a compactly represented policy class, which necessarily can't represent every function from states to action probabilities.

Track the belief state:

Modify our 5-state model from above to include partial observability: each action has a 50% chance of failing (doing nothing), and instead of observing the exact state index, we just see ℓ or r, with probabilities

obs \ state	1	2	3	4	5
ℓ	1	0.75	0.5	0.25	0

r	0	0.25	0.5	0.75	1

We start in state 3:

We go left:

We observe r:

We go left:

We observe ℓ:

Different kinds of models

REINFORCE is also very flexible about the type of environment model that we use. In our example above, we had a discrete, fully-observable state. This type of environment is called a *Markov decision process* or *MDP*. It is specified by

- a set of possible states
- a set of possible actions
- an initial state or distribution over states
- a transition function, telling us the probability of each next state given the current state and action

In our MDP, to specify a task, we need to give

- a cost function, telling us the immediate cost of each action in each state
- either a horizon H or a discount factor γ.

We can represent the transition function as a set of *transition matrices* T_a, one per action: $[T_a]_{ij}$ is the probability of arriving in state j if we start in state i and choose action a. And, we can represent our objective as a set of *cost vectors* c_a, one per action: $[c_a]_i$ is the immediate cost of taking action a in state i.

If our MDP is small enough that we can write down and work with the transition matrices and cost vectors directly, we call it a *tabular* environment — i.e., it is given by a set of tables. Tabular environments are the most straightforward to work with; they can sometimes have a bad reputation as toy problems, but they are nonetheless useful in modeling real systems.

If our state is partially observable, we have instead a *partially obesrvable MDP* or *POMDP*. A POMDP is specified by the same information as an MDP, plus

- a set of observations
- an observation function, telling us the probability of each observation given the current state and action.

In a POMDP, we would often track the belief state, resulting in a *belief state MDP*. This MDP has infinitely many states, represented as vectors in the simplex of

probability distributions.

A variation on an MDP or POMDP is to use a *factored state*: the state is represented by several random variables, and the transition and observation functions are written compactly in terms of these variables. For example, we could write a multi-agent system using factored state, with each agent owning a few of the state variables such as its position.

Another variation is a *discrete event system*, which is often used to simulate industrial processes. In such a system, the time between state transitions can be variable: we wait for the arrival of a job at a processing station, or the arrival of raw materials at a loading dock. An MDP with a factored state space and variable-time transitions is called a *semi-Markov* decision process.

A final type of model that we will often see is a *differential equation*: we have a continuous state and action $x \in \mathbb{R}^n$, $a \in \mathbb{R}^m$, and the state evolves in continuous time as

$$\dot{x} = f(x, a)$$

for some model function f. (Here $\dot{x}$ is shorthand for the derivative of x with respect to time.) Differential equations can include randomness (*stochastic* differential equations, where the state follows a Brownian motion or similar process). They can also include partial observability: we designate some components of x as observable, and maintain a belief over the others using a tool such as an *(extended) Kalman filter*.

Exploration and safety

One of the biggest differences between RL and supervised learning is that, in RL, we don't know ahead

of time what parts of the environment are interesting. Instead we have to discover low-cost or high-reward states as we optimize our policy. This is called the *exploration problem*.

In the worst case, the exploration problem is impossible: at the back of every wardrobe there might be an entrance to Narnia. We can't afford to spend all our time looking for wormholes like this. This tension — using our existing knowledge to minimize cost vs. looking for new unknown states and actions that might let us do even better — is called the *explore-exploit tradeoff*.

More precisely, in order to fully explore an environment, we might need to visit every distinct state enough times to collect data there. To explore in a reasonable amount of time in practice, we have to make assumptions about what states really count as distinct, and use these assumptions to organize our search. If these assumptions are wrong, we might miss out on arbitrarily good opportunities.

A useful class of exploration methods is *optimism in the face of uncertainty*. In these methods, we try to learn an approximate action-value (Q) function, and we use the learner to help decide what states are effectively distinct: if our function class tells us that we have enough data to reliably determine the value of a particular (s, a) pair, then we don't need to explore there.

In more detail, we use our function approximator to keep track of how confident we are in our predictions of values. In areas where we have a lot of data, we'll be highly certain of the values; in areas where we don't have much data, we'll be much more uncertain. At every opportunity to explore, we look for an area that has a *high upside*: e.g., if we build a confidence interval

for $Q(s, a)$ for a state-action pair in that area, the bottom (low-cost) end of that confidence interval is very low.

A full treatment of exploration methods is beyond our scope. But, some good places to start looking are:

- RMAX: this algorithm uses optimism under uncertainty to explore small (tabular) environments in polynomial time.
- MCTS (Monte-Carlo tree search): this algorithm extends the same intuition to larger environments, but only for relatively shallow look-ahead. It's included in famous RL systems like AlphaGo.
- Thompson sampling, in particular bootstrap Thompson: these methods try to efficiently search for high-upside areas by drawing samples from a posterior distribution over value functions.

The problem with trying to explore is *safety*: if we intentionally seek out areas where we don't know what's going to happen, we might risk bad consequences. For example, we might want to be very careful about exploring with a nuclear reactor controller, or a bank regulator, or a drug trial.

As with exploration, safety is an impossible problem in the worst case: without further information, Freddy could be hiding around every corner. Often, we need to depend on external safety knowledge, such as an expert-designed system that detects anomalies and invokes an emergency policy to return our system to a safe state.

A useful primitive, though, is *pessimism in the face of uncertainty*: just as above, we use a function approximator to tell us about the range of possible outcomes from visiting an area of state space. But now, instead of seeking out uncertainty, we avoid it: we

only plan to go to places where we are confident of success.

In general, it's an open problem to develop a practical and well-motivated system that trades off optimism and pessimism to achieve both exploration and safety. It's not even well-understood how to express and use different varieties of expert knowledge to help with these problems.