Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2024

Contents

CHAPTER 11

Reinforcement Learning

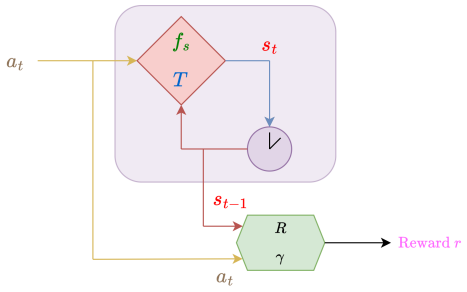
11.0.1 MDP Review

Last chapter, we explored MDPs, a tool for simulating a "game". We, the "player", choose which **actions** we take.

- Different **actions** can
  - Change the **state** of the world: what our system looks like.
  - Provide us with **rewards**, based on our actions.

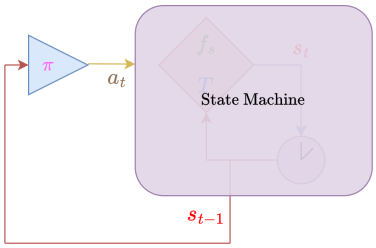
However, we our system isn't perfectly consistent:

- The **transitions** between states are **probabilistic**: we don't know our exact next state, but we know the **odds** of each possible next state.



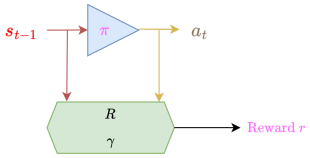
Here's our MDP: our model of a system that we can change over time.

Given complete knowledge of our system, we wanted to come up with the best possible strategy (**policy**) for getting the most reward, over time.



Our policy chooses different actions, based on what state our state machine gives us.

We evaluate our policies based on the **average expected reward**, for each state.



Combining these three parts (state machine, reward function, policy), we would find the best policy, using **value functions**, and **Q-value functions**.

11.0.2 What if we don't know as much?

There's a major limitation of this approach:

- It assumes we have know everything about our system.

Concept 1

Value functions can only be computed if you have **complete knowledge** of your MDP:

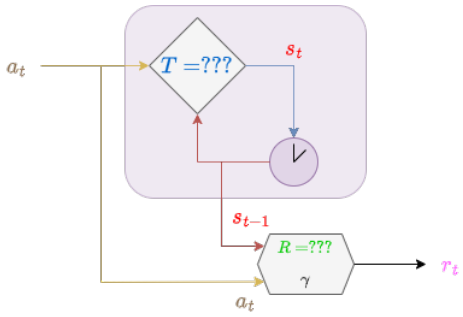
- What are the odds of your **state transitions**?
- What **rewards** will you get in different situations?

Without this information, it's not possible to compute the "value" of a policy, using our previous techniques.

$$V_\pi^H(s) = R\Big(s, \pi(s)\Big) + \sum_{s'} T\Big(s, \pi(s), s'\Big) \cdot V_\pi^{H-1}(s') \tag{11.1}$$

- This equation is impossible to compute without those crucial variables T and R.

- But in plenty of real situations, you won't know exactly what effects your actions might have.

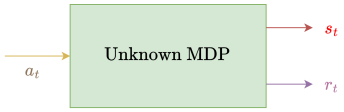


Often, we don't know T and R.

11.0.3 Learning about our MDP

If we don't know our transitions, or our rewards, our model is reduced to a simple box: based on an action, you see the next state s_t, and your reward r_t.

This simplified object is called the "environment" for our player.



Since we don't know what's inside, we reduce our MDP to a simple input-output machine.

The only way to learn our MDP is to **exploring** and gathering data.

The only way we can interact with our MDP is by taking **actions**. So, we do that:

- We are given the initial state s_0.
- We experiment, and take an action a_1.
- We learn some information:
  - We get reward r_1.
  - We transition to new state s_1.

Repeat.

We continue until we're satisfied, choosing actions and getting feedback.

Concept 2

In **reinforcement learning** (RL), we want to learn more about our MDP, so we **experiment**, by taking different **actions**.

- We take an **action**, and see what it **does** (state transition, reward).
- We do it again. And again.

By **experimenting**, and continuously getting feedback, we slowly learn about our MDP. This gathers up our data:

$$\begin{bmatrix} s_0 & s_1 & s_2 & \cdots s_n \end{bmatrix} \qquad \begin{bmatrix} r_1 & r_2 & \cdots r_n \end{bmatrix}$$

Example: You have a panel of buttons. You ask yourself, "what does this one do?", and press one of them.

- Then, you might ask: what if I press them in a different order? In different situations?
- As you learn more, you gradually figure out a "better" way to play.

This is very similar to how you might play a video game when you first pick it up.

11.0.4 Reinforcement Learning

Now that we know what to do, we need to **formalize** it.

Represent things with math, give each part a name, etc. Things that will make it easier to talk about.

We can divide up this process into two:

Definition 3

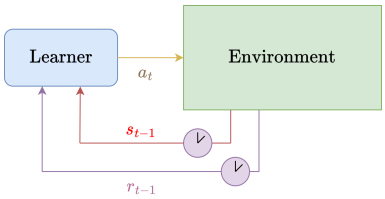
Our **reinforcement learning** (RL) problem can be divide into two main parts:

- The **learner**: this is the "player" of the game.
  - The learner chooses which **actions** to take: they decide the policy.
  - Based on what they observe from the environment, they learn to make **different choices**.
  - Eventually, the learner's goal is to make better choices, to get the most **rewards**.
- The **environment**: this is the "game" that the player is interacting with.
  - The environment reacts to the learner's actions, responding with a **reward** and **state change**.

The learner is trying to **learn** about the environment, and discover the best **policy**.

"Learning a better policy" is what we wanted in the MDP chapter: this time, it just takes more work.

The learner chooses the action, and the environment teaches the learner. So, they work in a feedback cycle:



Why does our diagram use s_{t-1} and r_{t-1}? Because **past** data is used to make **future** decisions, like a_t.

Our learner makes decisions, while the environment gives feedback on those decisions. This feedback is used in the future to make better decisions.

Notation 4

In this chapter, we use capital R to represent the reward **function**, and lower-case r_t to represent a **single** reward at time t.

$$R(s_{t-1}, a_t) = r_t$$

If we expect our environment to behave like an MDP, that's what we'd put inside the "environment" block. But, RL isn't necessarily limited to that framework:

Clarification 5

So far, we've used MDPs as a concrete example for RL, but RL can be used for some other related systems, as well.

Example: One alternative environment is the "partially observable (PO) MDP".

This requires more inference than we'll cover in this class.

11.0.5 Supervised vs. Unsupervised vs. RL

RL is a bit different from our previous training frameworks: "supervised" and "unsupervised".

Let's review:

- **Supervised learning**: you're explicitly given an input $x^{(i)}$, and a desired output $y^{(i)}$
  - **Example:** This is similar to being given a test, with the answer key.
- **Unsupervised learning**: you're given inputs, but you're not given an output: you have to look for patterns or structure without outside help.
  - **Example:** You're given a set of photos, and asked to sort them, based on what object is in the image. You aren't given labels.
- **Semi-supervised learning**: you're given *some* answers, but not most of them.

Reinforcement learning is a bit different:

Concept 6

Reinforcement learning (RL) provides data to the model differently from supervised / unsupervised frameworks:

- The model has some **choice** in which data it sees: it chooses **action** a_t, which affects the feedback s_t and r_t.
- This means the model doesn't just learn by observing the data: it **interacts** with it.

Over the course of training, our model can make **different choices** about what to learn, based on what it's already seen.

This approach forces our model to not only learn the structure of the data, but how to ask questions.

Just like how a student learns when to ask questions, and what they need to practice.

11.1 Reinforcement Learning Algorithms Overview

Our "learner" is more complex than our previous system for choosing actions:

- It gradually **learns** the rewards and state transitions.
- It has to not only choose the best rewards, but also choose what parts of the environment to **explore**.

But even so, it only really makes one decision: choosing actions. It's still a type of **policy**.

Concept 7

A **reinforcement-learning (RL) algorithm** is a type of **policy**.

It chooses our **next action** based on all of its past data:

- States, actions, rewards

Similar to $\pi(s)$, the goal is to **maximize rewards**.

- However, our RL algorithm first has to **explore** different parts of the environment, to know what the rewards and transitions are.
- This is why it needs to use all of our past data: to **learn**.

When we're finished training, it may be possible to just keep the policy $\pi : \mathcal{S} \to \mathcal{A}$, and discard all of our past data.

This is similar to how, when we finish training our NN, we just give people the model, not the training data.

Our trained model contains the information we need to make decisions: we don't need all the training data.

11.1.1 Evaluating RL algorithms

How do we evaluate our RL algorithm? There are a couple different ways:

Concept 8

We can **evaluate** our learning algorithm based on **how long** it takes for it to learn a mostly-**optimal policy**.

- In other words: "how long does it take to train?"

In this scenario, we ignore the rewards we get while learning: our model is allowed to make **mistakes**.

In other situations, we want our model to do well while training:

Concept 9

We can also **evaluate** our learning algorithm based on **expected rewards** while training.

- In other words, "can it perform well, while still training?"

In this scenario we're focused on rewards **while learning**: we don't want our model to make as many mistakes.

Which do we usually use?

- We use the first one more **often**, because it's often easier to design and measure: we just train the model first, and keep track of the total time.
- The second one is more **challenging**: it's difficult to create a model that can perform reasonably well, while still learning.

But, sometimes the latter is necessary: you may need to train in real situations, where the rewards really matter.

Concept 10

If we have a **safe**, cheap environment to train in, it's easier to train first, and then figure out performance later.

But if you're training in a **costly** environment, you need to make sure your model performs well, even while still training.

Example: Suppose that you want to train a car in real traffic environments: simulations aren't good enough.

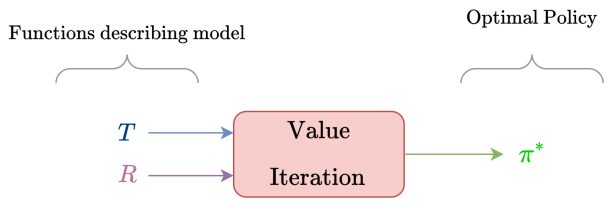
- You really don't want your car to make major mistakes in real traffic: even if you're "training", the accidents are very real.

11.1.2 Different types of RL models

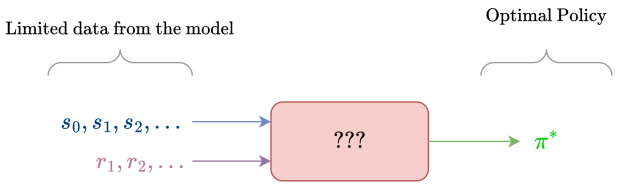
In a reinforcement learning situation, what we're missing is **information** about our model.

- We don't know our transitions T, or our reward function R.
- In our value-iteration setting, we used these to compute what's **optimal**: Q and π.

Value iteration can use **full information**:



Reinforcement learning is more restricted. We have **limited data**: some data points s_t and r_t.

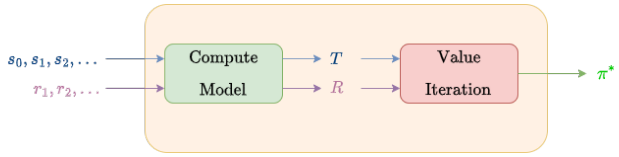


11.1.3 Types of Reinforcement Learning

There are multiple ways we can solve this problem. We'll focus on two types of approaches: **model-based** and **model-free** methods.

In a **model-based** RL algorithm, we use our data to try to **guess** the MDP model: we compute an approximation of T and R.

- Once we've computed T and R, we can use **value iteration**.



In this approach, we can re-use our previous logic.

Definition 11

A **model-based** RL algorithm uses our previous MDP techniques to the find the **optimal policy**. This approach requires **knowledge** of our model.

- First, we **approximate** our **model** (T and R) based on data (s_t and r_t).
- Then, we use that to do **value iteration**.

Our other approach is to use a **model-free** RL algorithm, where **skip** trying to compute T and R.

- Instead, we **directly** compute either Q or π^*.

We don't even bother learning our MDP.

We don't need T, R, or value iteration.

Definition 12

A **model-free** RL algorithm gives up our previous MDP techniques. Meaning, we **don't try** to directly compute our model (T and R).

Instead, we find other ways to use our data (s_t and r_t):

- In **Q-learning**, we approximate the **state-action value function** $Q(s, a)$, using **Q-learning**.
  - We find the best policy by maximizing $Q(s, a)$.
- In **policy search**, we represent our policy π with a **computable function** $f(\theta)$, and try to **optimize** that function.
  - We might use gradient descent, for example.

In this chapter, we will go in the following order:

- Model-free methods
  - Q-learning
  - Policy Search
- Model-based methods
- Bandit problems

11.2 Model-free methods

As we've already discussed, model-free methods are those where we don't learn T and R (our model). Instead, we skip over that, more directly learning our solution.

We generally boil these down into two kinds of approaches:

Definition 13

We can sort **model-free methods** into two basic types:

- **Value-based** methods: we compute the value function V or Q.
- **Policy-based** methods: we compare policies π directly.

These two approaches aren't necessarily completely separate from one another:

Clarification 14

Often, in more detailed models, the line between **value-based** and **policy-based** methods is blurry.

- Some techniques are somewhere in between.

We can even **combine** these into a single, more detailed algorithm.

- Some complex algorithms incorporate all of these elements: value functions, policies, transition/reward models.

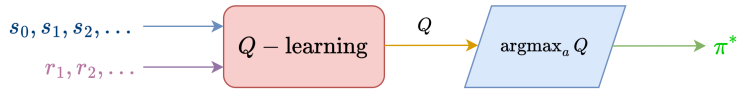
11.2.1 Q-learning: Computing Q from new data

We'll start with a popular **value-based** approach: Q-learning. Our goal is to compute Q directly.

- Then, we can find the optimal policy by maximizing Q.

$$\pi^*(s) = \arg\max_a \Big(Q(s,a)\Big)$$

This is our process:



Rather than use T and R to compute Q, we compute Q directly.

Note the major difference:

Clarification 15

Value iteration and **Q-learning** can seem similar, because they both use our **model** to compute Q.

The main difference is that:

- Value iteration is used when you **fully understand your model** (T and R).
- Q-learning is used when we have **data points** (s_t and r_t).

In Q-learning, we determine Q based on our **experiences**.

How do we do Q-learning? Well first, let's remind ourselves of how we traditionally compute Q.

$$Q(s,a) = R\Big(s,a\Big) + \gamma \sum_{s'} T\Big(s,a,s'\Big) \cdot \max_{a'} \Big(Q(s',a')\Big) \tag{11.2}$$

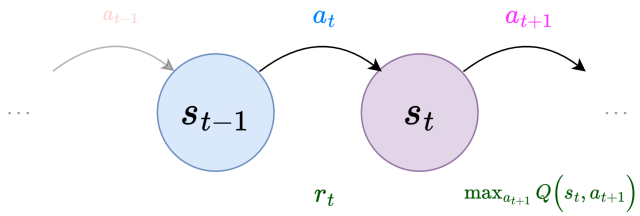
We don't have T or R. But, we can still use the basic idea:

- We take **one action** a, and end up going from **state** s to s'.
- Then, we use the **optimal policy** starting from state s': that's why we take the max of Q.

Let's try to apply this to one timestep of our "exploration data":

- We started in state s_{t-1}, and took action a_t.
- We moved to state s_t, and got reward r_t.

Our future rewards come from picking the **best** next action a_{t+1}.



We'll apply this to our Q equation, to get an **approximation** of Q.

$$Q_{data}(s_{t-1}, a_t) = r_t + \gamma \cdot \max_{a_{t+1}} \left(Q_{old}(s_t, a_{t+1}) \right) \tag{11.3}$$

Notice that we don't average over possible states ($\sum_{s'}$, TQ):

- This "expected value" was used because we **didn't know** what our next state, s', looked like.
- But in this case, we know which state we moved to: s_t.

Key Equation 16

When deriving our Q-value, we broke our reward into two parts:

$$Q(s, a) = \left(\text{immediate reward}\right) + \left(\text{future reward}\right)$$

If we apply this to one timestep of our simulation, we get:

$$Q_{data}(s_{t-1}, a_t) = r_t + \gamma \cdot \max_{a_{t+1}} \left(Q_{old}(s_t, a_{t+1}) \right)$$

11.2.2 Q-learning: Making an update rule

Now, we have an approximation. But this approximation is only based on one data point. How do we incorporate **multiple**?

- We could **update** our Q-value every time we get new data for that state/action pair.

- That way, we can update it repeatedly, to incorporate multiple data points.

Our current equation doesn't allow us to "update" our Q-value, though: it **replaces** it. We'll modify the above equation to get our **update rule**:

- We want to **average** our new Q-value with our old one.

How much do we emphasize our new Q-value, versus our old one? We'll represent this with a **learning rate** α.

Definition 17

When we **update** our Q-value, our **learning rate** α ($\alpha \in (0, 1]$) tells us how much we emphasize our new Q-value, based on **one data point**.

- $(1 - \alpha)$ tells us how much we emphasize our old Q-value, based on all past data points.

$$Q_{new}(s_{t-1}, a_t) = \alpha \cdot Q_{data}(s_{t-1}, a_t) \quad + \quad (1 - \alpha) \cdot Q_{old}(s_{t-1}, a_t)$$

We can also describe α a little more conceptually:

Concept 18

If α is **small** ($\alpha \approx 0$), we care **very little** about new data.

If α is **large** ($\alpha \approx 1$), we are almost entirely **focused on** new data.

- We call this a "**learning rate**", because it tells us how much we **learn** from new data.
- But we could also think of it as a "**forgetting rate**": in order to learn from new data, we **pay less attention** to older data.

Our equation for Q is going to be messy, so let's change notation:

Notation 19

We update all of our variable names:

- $s = s_{t-1}$, $s' = s_t$, $a = a_t$, $a' = a_{t+1}$, $r_t = r$

As well as our value function: $Q = Q_{old}$

If we plug in our previously calculated Q_{data}, we have our Q-learning equation:

Key Equation 20

In Q-learning, all of our Q-values start as 0 (similar to value iteration):

$$Q_{new}(s,a) = 0$$

With each new data point, we **update** our Q value:

$$Q_{new}(s,a) = (1-\alpha)\cdot Q(s,a) + \alpha \cdot Q_{data}(s,a)$$

$$Q_{new}(s,a) = (1-\alpha)\cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'}\left(Q(s',a')\right)\right)$$

If we're being specific, we sometimes call this approach **tabular Q-learning**.

11.2.3 Selecting our action: ϵ-greedy

Now, we have a way to **update** our Q-value, based on a data.

- But we need a way to actually **get** our data: we need to start **exploring** the space.
- Which means our model **decides** what **data** it wants to **see**.

We could try always exploring whichever action seems most optimal. But this is a bad strategy when you're starting out:

Concept 21

It's not usually a good idea to **always** use the most "**apparently optimal**" strategy during training.

- There may be plenty of strategies that don't seem good *at first*, but will look more rewarding after some **exploration**.
- Your Q values are often very inaccurate, early in training.

Example: Suppose that there's some treasure at the end of a path.

- You might take 3 steps, and give up: walking around takes work, and you're not immediately rewarded.
- You'll miss out on that treasure, because you don't know it's there yet.

But exploring blindly isn't entirely helpful: it's *too slow*.

- It's often more useful to search near **high-reward** areas: these are more likely to be searched by (and be useful to) a **good policy**.

This is the **exploration vs. exploitation problem**.

Definition 22

When we're trying to find the **best policy** for exploring a space, we run into a problem called **Exploration versus Exploitation**.

- **Exploration**: you're trying to **learn** more about the space, and you're not as focused on maximizing reward. You *explore* your options.
- **Exploitation**: based on what you've learned, you want to get the **maximum reward** from it. You *exploit* your knowledge.

If you explore more, you might learn how to get better rewards. But if you explore for too long, you'll waste time you could've spent taking advantage of that knowledge.

How much of each should we use? It depends on the context:

- If you only have **10 seconds** left in a game, it might not be worth it to explore anymore: you might as well cash in what you know how to do.
- But if you have **5 hours**, you're more likely to find something useful before the game ends: maybe you should explore more.

For Q-learning, our simplest option is to **randomly** alternate between the two modes: *explore* with probability ϵ, and *exploit* with probability $(1-\epsilon)$.

Definition 23

The **ε-greedy strategy** for Q-learning chooses our **actions** for interacting with the environment, **randomly**:

- With probability ϵ, we choose an action $a \in \mathcal{A}$ **uniformly, at random**.
  - We are equally likely to choose any action: we're **exploring**.
- With probability $(1-\epsilon)$, we choose the action that gives us the **most reward**, based on what we know:

$$\arg\max_{a \in \mathcal{A}} Q(s,a)$$

  - We're getting the most reward we can: we're **exploiting**.

How long do we want to run our Q-learning algorithm? It depends on the situation:

Concept 24

We can choose our **termination condition** for Q-learning based on our needs.

We could, for example:

- Terminate after a **fixed** number of timesteps T
- Terminate when our Q-values **aren't changing** much on successive iterations
- Terminate if we get **stuck** in the same state, or a loop

11.2.4 Q-learning

Based on this, we now have a completed Q-learning algorithm:

Definition 25

Q-learning is a strategy for learning the Q-values of our MDP, so we can find the **optimal policy** for our model.

We use the following steps:

- We set all Q-values to 0. Start from some initial state, s_0.

$$Q(s, a) = 0$$

- We repeat the following, until we reach our **termination condition**:
  - Select an **action** based on Q and current state (possibly with ϵ-greedy)

$$a_t = \text{select\_action}(Q, s)$$

  - Record the **result**

$$\text{MDP}(s_{t-1}, a_t) = s_t, r_t$$

  - Update our **Q-values** accordingly.

$$Q(s, a) \Longleftarrow (1-\alpha) \cdot Q(s, a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} \left(Q(s', a')\right)\right)$$

- Compute our **policy** based on Q.

Q-learning is guaranteed to **converge** under surprisingly simple conditions:

`select_action` isn't a specific function: in our case, it could just be ϵ-greedy. But we could choose other options.

Theorem 26

Q-learning **converges** if

- Over an **infinitely-long run**, we visit every state an **infinite number of times**.

With this requirement, we ensure that our model doesn't decide on a sub-optimal strategy, without checking out other possibilities.

Guaranteed convergence does require our learning rate α to **decay**, or gradually shrink over time.

Okay, so an infinite amount of time isn't exactly promised to us... but it's better than a lot of other stricter convergence requirements!

- But typically, we set α to a constant, for convenience.

We can set it to decay, but this also slows down the learning process.

Now that we have our completed Q-learning strategy, let's go through some details that we skipped over.

11.2.5 Initialization

When we're starting our Q-learning process, we have to choose some initial state, s_0.

- For some problems (like a chess game), there's a **natural choice** of initial state.
- For other problems (like a robot moving across terrain), there may be **multiple** possible "initial states".

In the latter case, we often **randomly** select our initialization.

> **Concept 27**
> When we're uncertain what **initial state** s_0 to use, we often **randomly sample** from our state space.

This choice of initialization often **biases** what we learn about the state space: which sections we **visit**, what we **learn**, etc.

So, it's often helpful to run Q-learning through **several initializations**: we have one "run" of our MDP for each initialization.

- So we don't lose all of our progress, we usually modify it so that our Q-table (computed Q values) is **carried over** between different "runs".

> **Concept 28**
> To explore our **state-action space** (possible options) more thoroughly, we may take **several different paths** through our MDP.
>
> - Each path starting with a **different initialization**, s_0.
>
> We **share** our Q-values between these "runs" of our MDP, so that we can build up a more complete representation of the environment.

11.2.6 Action and state space

Our previous approach to Q-learning assumes that our **action space** and **state space** are both **discrete**.

- But this might not always be a realistic assumption. We might need a **continuous** space.

Concept 29

Our above approach to Q-learning is called **tabular Q-learning**.

It assumes that we have a **discrete** (typically finite) **state space** and **action space**.

- Other versions of Q-learning, on the other hand, allow this space it be **continuous**.

We call it "tabular Q-learning" because our values could be stored in a **table**.

Example: A discrete state space might be $\{1, 2, 3, 4, 5, 6\}$. A continuous one might be $[1, 6]$.

- $1 + \sqrt{2}$ is allowed in the latter, but not the former.

This causes us problems, though: if we have a continuous state/action space, we have an **infinite** number of possible states/actions.

- It's impossible to get the Q-values for all of these state-action pairs.

Many Q-learning variations enable continuous action/state spaces. Later, we'll focus on one example: **Deep Q-learning**.

11.2.7 An alternate view of Q-learning (Optional)

Consider our basic, conceptual Q-learning equation:

We're averaging our immediate data, with our past experience with this Q-value.

$$Q_{new}(s, a) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot Q_{data}(s, a) \tag{11.4}$$

We get something interesting if we rearrange it:

$$Q_{new}(s, a) = Q(s, a) + \alpha\Big(Q_{data}(s, a) - Q(s, a)\Big) \tag{11.5}$$

The right term could be seen as the **disagreement** between our new data, and past experience.

- And thus, α tells us how much we **care** about that disagreement, and want to account for it.

$$Q_{new}(s, a) = Q(s, a) + \alpha \overbrace{\Big(Q_{data}(s, a) - Q(s, a)\Big)}^{\text{"Error" of our old answer}} \tag{11.6}$$

This is an **update rule**: the difference between our new and old answer decides how we want to update.

Concept 30

We can view **Q-learning** as a direct **update rule**:

- We "update" our current Q value based on the **difference** from what the **newest data point** predicts.

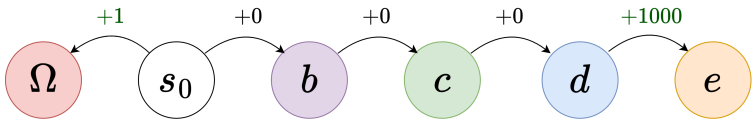
$$Q_{new}(s,a) = Q(s,a) + \alpha \overbrace{\left(Q_{data}(s,a) - Q(s,a)\right)}^{\text{"Error" of our old answer}}$$

11.2.8 Problems with Q-learning: Slow Convergence

Because Q-learning only updates one state-action pair at a time, it often **converges slowly**.

Let's see an example:

- **Example:** We'll re-use our example of going down a long hallway, with treasure at the end.
- Each "state" is one tile of the hallway. We'll arrange them **left-to-right**: we can move left or right down the hallway. We start on s_0.



Above the arrows, we can see the reward we get for going left/right in each state.

If we go left, we get a small reward. If we go right, we'll *eventually* get a huge reward.

Assume that every state transition we don't show (left/right) is +0.

Being able to see from above, it's obvious to us that going **right** is better. But what does the robot see?

So long as γ isn't really small: if our model is really likely to fail after 1 or 2 steps, then the right reward isn't worth it.

- Go right once. **No reward.**
- Go left once. **Reward!**
- We should go left!

Concept 31

At first, our Q-learning algorithm will prioritize **short-term** rewards over long-term rewards.

- It hasn't had time to **find** rewards further from s_0.

Well, as the robot explores, it'll learn to get the reward, right? Let's see what happens as we move right, to our Q values.

$$Q(s,a) \quad \Longleftarrow \quad (1-\alpha)\cdot Q(s,a) \;+\; \alpha\cdot\left(r+0.9\cdot\max_{a'}\left(Q(s',a')\right)\right) \tag{11.7}$$

For simplicity, we'll use $\alpha = 1, \gamma = 0.9$.

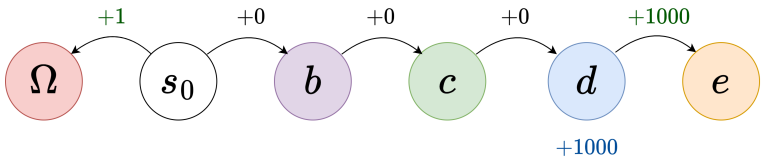
$$Q(s,a) \quad \Longleftarrow \quad r+0.9\cdot\max_{a'}\left(Q(s',a')\right) \tag{11.8}$$

Based on this model, our reward for going left (←) is simple:

$$Q(s_0, \leftarrow) = +1 \tag{11.9}$$

Let's go **right** ($\rightarrow$) instead.

- Move from s_0 to b: no reward. $Q(s_0, \rightarrow) = 0$
- Move from b to c: no reward. $Q(b, \rightarrow) = 0$
- Move from c to d: no reward. $Q(c, \rightarrow) = 0$
- Move from d to e: reward! $Q(d, \rightarrow) = +1000$



We learned that d is able to produce a +1000 reward.

We did it! We learned something, at the very end. Will our robot go the way we want?

- We start over from s_0. Let's **compare** the left and right rewards.

$$Q(s_0, \leftarrow) = +1 \qquad Q(s_0, \rightarrow) = 0 \tag{11.10}$$

- Let's go left again!

No luck – it still prefers the **short-term** reward.

Concept 32

Even once we find a reward, Q-learning will only update that **single state-action pair**.

- That means that nearby states, **don't know** about that reward!

We have to run Q-learning through a nearby state *again* to find the reward.

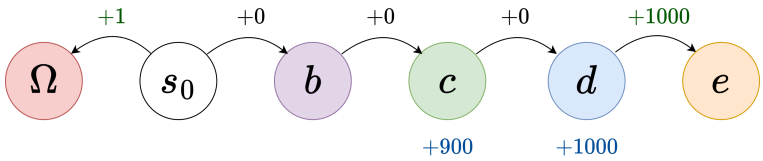
So, let's take another journey:

- Move from s_0 to b: no reward. $Q(s_0, \rightarrow) = 0$
- Move from b to c: no reward. $Q(b, \rightarrow) = 0$
- Move from c to d: no reward. **However**, Q **remembers** that d can provide a reward!

$$Q(c, \rightarrow) \quad \Longleftarrow \quad r + 0.9 \cdot \max_{a'} \left(Q(d, a')\right)$$

$$Q(c, \rightarrow) \quad \Longleftarrow \quad 0 + 0.9 \cdot \overbrace{Q(d, \rightarrow)}^{+1000} = +900$$

We know that d is valuable. Thus, we've learned that c is valuable, because it's attached to d.



It's worth visiting c, because it allows you to visit d.

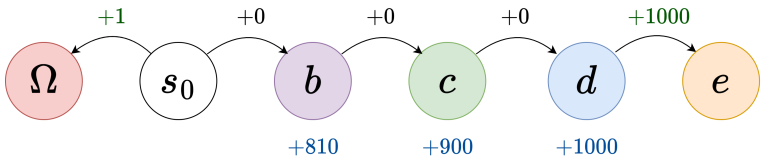
Concept 33

Each time that we run though a path to a reward, **one more state** learns about the reward.

- If state d has an **action** with a **reward**, then d **is valuable**.
- If state c can move to d, c **is valuable**, because it gives a **path to reach** d.
- If state b can move to c, b **is valuable**, because it gives a **path to reach** c.
- This repeats until we reach s_0.

Each Q-learning run will update one more state.

If we continue, we get the result we're looking for:



We can now see that b has a lot of value. (The bottom number is the "expected value" we can get after reaching state s, if we make the best choice.)

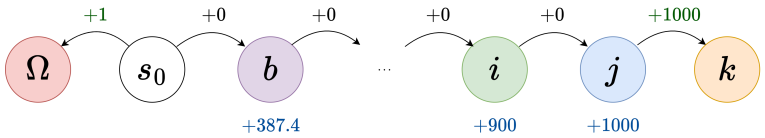
Let's compute $Q(s_0, \rightarrow)$, now that we know b is valuable:

$$Q(s_0, \leftarrow) = +1 \qquad Q(s_0, \rightarrow) = +729 \tag{11.11}$$

Finally, we go right!

- But it took 4 trips right before we knew that.

This is already annoying, but it can get even worse: suppose we only reached the reward after moving right **10 times**.



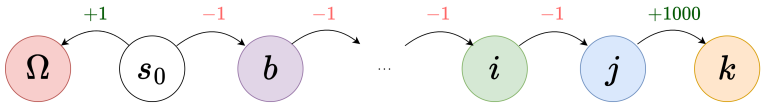
It's still worth it to go right, but it takes a painfully long time to figure that out.

Instead of making 4 trips right, we'll have to make 10 trips right.

Each trip is, thankfully, shorter than the last. But that's still really slow.

- And imagine if the "reward" for going right was -1 instead of +0.
- Our model would *always* prefer to go left, until the very end. Which means, it only has an $\epsilon/2$ chance of moving right.
- Going right n times in a row has a chance of $(\epsilon/2)^n$.

ϵ chance to move in a random direction, and 1/2 chance to randomly move right.



If moving left from $(b, c, d...)$ is still +0, our model will try to avoid going right. It's even harder to make progress, now.

Concept 34

The **longer** it takes to reach a distant reward, the more **difficult** it is to **propagate** that information back to s_0.

- This shows how *inefficient* Q-learning can be: only updating one **state-action** pair at a time, means that information travels **slowly** between states.

11.2.9 Deep Q-learning

Earlier, we mentioned that our state space $\mathcal{S}$ and action space $\mathcal{A}$ are **discrete** and relatively **small**.

- But what do we do if we need them to be **continuous**?

Or, just very large? A large finite space is still a pain.

One solution is to treat Q like any other continuous variable we want to **predict**.

- What do we do with complicated, continuous variables we want to predict? We use **neural networks**.

Definition 35

In **Deep Q-Learning**, we use **deep neural networks** to predict Q-values: a **regression** problem.

- This approach allows us to handle **continuous** state and action spaces.

To teach this network, we train it the way that we train any neural network, using data we receive while exploring:

- Input: **states** and/or **actions**
- Output: expected **reward**, Q_{NN}.

Our goal is to make the most accurate predictions of the Q-value. We determine Q based on each data point,

$$Q_{data}(s_{t-1}, a_t) = r_t + \gamma \cdot \max_{a_{t+1}} \Big(Q(s_t, a_{t+1}) \Big) \tag{11.12}$$

So, we want our predicted Q value (Q_{NN}) to be **as close** to Q_{data} as possible.

Definition 36

Our **deep Q-learning** neural network will use **squared error**:

$$\Big(Q_{NN}(s,a) - Q_{data}(s,a)\Big)^2$$

In other words, our goal is for our NN (Q_{NN}) to match the Q-values of our data points (Q_{data}), as close as possible.

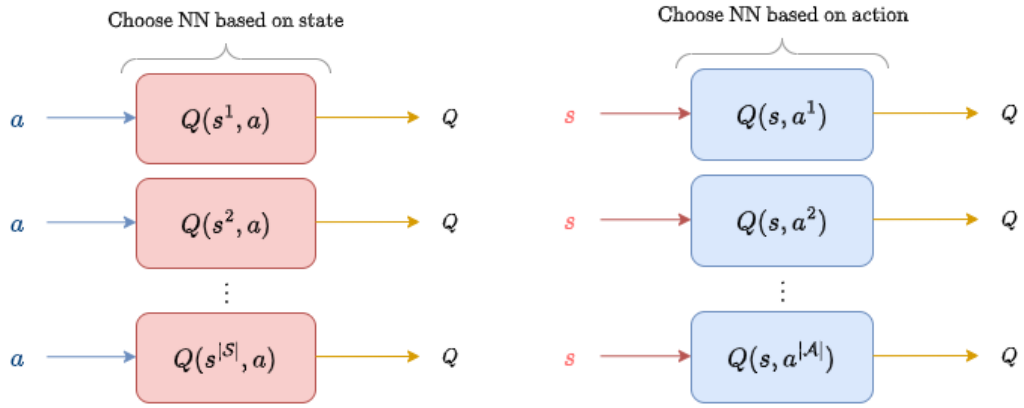
$$\left(Q_{NN}(s,a) - \Big(r + \gamma \cdot \max_{a_{t+1}} Q_{NN}(s',a')\Big)\right)^2$$

Note that, in our definition, we said states **and/or** actions: we might not have both as the input to our neural network. How?

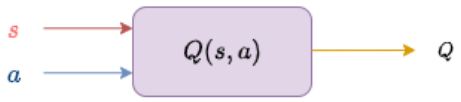
Concept 37

There are three main ways we can design our **neural network**: in all cases, $Q(s, a)$ is the **output**.

- Each **action** a has a separate neural network. **State** s is the input.
  - This only works with a **small, discrete action space**.
- Each **state** s has a separate neural network. **Action** a is the input.
  - This only works with a **small, discrete state space**.
- We have one neural network shared by **all inputs**. **State** s and **action** a are *concatenated* into the input.
  - This one is the most flexible, but it's **very hard** to find $\arg\max_a Q(s, a)$.



In one system, each **state** s^i has its own neural net. In the other system, each **action** a^j has its own neural net.



This version works for continuous state/action spaces, but comes with its own difficulties.

Unfortunately, deep Q-learning is often pretty unstable.

Improving, and then getting worse, for example.

- But it's still useful enough to try, in a lot of contexts.

11.2.10 Catastrophic Forgetting

Here, we'll address one of these forms of **instability**.

- When training a typical neural network, all of our data is **IID**: independent, and coming from the same distribution.
- But this is **not** the case for Q-learning.

Concept 38

In Q-learning, our data are **correlated in time** ("temporally correlated"). Meaning, **timing** affects our data.

- Why? Because two **states** which are "**near**" each other, typically behave **similarly**.
- If the **time** between two data points is **short**, they're probably **nearby** in state space. So, they're more likely to be similar.

Example: Consider a robot moving across the earth.

- **Example 1:** If, at time t, our robot is on a **mountain**, it's more likely to be on a mountain at $(t-1)$ and $(t+1)$.
- **Example 2:** The 12 hours of daytime may seem very different from the 12 hours of nighttime.

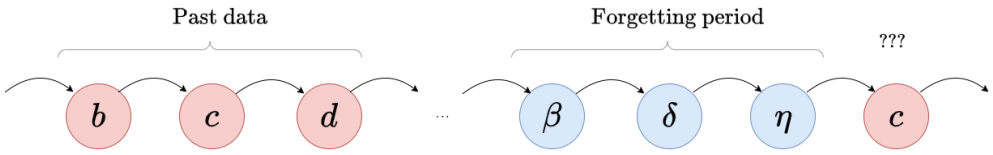
Why is this a problem? Because our neural network adjusts Q based on **new data**.

That means that our NN is capable of **forgetting**: if there's been a long time since we've used some information, it will be replaced by information from a **different context**.

Definition 39

Catastrophic forgetting occurs when our neural network hasn't seen a certain type of data in a long time, and **forgets** how to do a task.

- In deep Q-learning, it can occur when **recent** data doesn't reflect **past** data.
- So, our model *forgets* about the portion of the state space it visited in the past.



When we return to the red region, we've forgotten what we learned the first time!

This is still a problem, even if we don't return to the red region during this MDP run:

- It could be important for future runs.

11.2.11 Experience Replay

What do we do if a person is worried about forgetting something? You **remind** them of the past information.

- Perhaps they talk about that memory, or you periodically mention it to them.

This is our solution: we **keep track** of these past experiences, and re-use them later: we essentially "refresh" our NN, so that it doesn't forget.

This is called **experience replay**.

Definition 40

Experience replay is a technique for addressing **catastrophic forgetting**.

- In experience replay, we store our **past experiences** (s, a, s', r) in a **replay buffer**.
- This buffer is used to "**remind**" our NN of past events.

After every timestep, we do two things:

- We store our newest experience in our replay buffer.
- We **randomly** pull n memories from our replay buffer, and "**re-experience**" them: we re-train our model, based on these past events.

This storage can get painfully large, though. This can be problematic:

- If the memories are too far back, they may just **not be relevant** anymore: they're in an undesirable part of the state space.
- It becomes **expensive** to store all of those memories.

So, we tend to only keep some of them.

Definition 41

Rather than storing *every* event in our **replay buffer**, we only keep the k **most recent memories**, in a **sliding window**.

- This prevents our memory from getting **too full**, or focusing on memories that are **too old**.

The best **size** for our sliding buffer **depends** on the problem, and what our state space is like.

Another reason that we like **experience replay** is for improving on a weakness we mentioned before:

Concept 42

Randomly reviewing **old memories** has a second benefit: it allows us to **propagate rewards** between states faster.

- Previously, we only updated **one state-action value**, for each experience.
- This means that, when we get our **reward**, we **only** update the state s_r we got the reward in.

With experience replay, we're more likely to **revisit** a state s_n "near" our reward:

- If s_n is near s_r, then s_n is more valuable, for being a **path** to the reward.

Example: This might, for example, help speed up the hallway problem.

By "nearby", we mean that there's an short series of actions a_t that moves us from s_n to s_r.

The one we used earlier in the chapter.

11.2.12 Fitted Q-learning

Here, we'll try a *different* approach for deep Q-learning, that avoids the "catastrophic forgetting" problem.

Our "forgetting" problem is caused by the fact that our data comes in a **particular order**: older data is learned earlier, and risks being forgotten, all together.

- Is there a way to "**shuffle**" the data we receive, before using it to train Q?

The problem is, we use Q to **choose** our data.

Concept 43

We want to gather data **before** training Q (so we can shuffle it).

- But we use Q to **decide** how to gather data.

We would need to have Q, in order to train Q – that seems paradoxical.

The solution? We have **two Q functions**: one we use to gather new data (but not train), another we train afterwards.

- Q_{old}: trained on all **previous data**. We use this function to **decide** our actions, and gather more data.
  - We **do not** re-train Q_{old} as we receive new data: we want to *avoid* training our data **in order**.
- Q_{new}: once we've gathered enough data, we use **all of our data** (old and new) to train a new Q function **from scratch**.
  - We **shuffle** our data, so that Q_{new} *also* avoids training our data in order.

If we have no data yet, Q_{old} is just the "default" Q-value function: $Q_{old}(s, a) = 0$.

Meaning, we start with $Q_{new}(s, a) = 0$, and then train.

Q_{new} can, then, be used to decide our future actions: it replaces Q_{old}.

$$Q_{old} \Leftarrow Q_{new} \tag{11.13}$$

Concept 44

In typical Q-learning, we take an **action**, get data, and immediately **update** Q with that new data.

- This means our Q-value function is trained in the **order** we receive the data.

In **fitted Q-learning**, we separate the **data-gathering process** (Q_{old}) from the **training process** (Q_{new}).

- We use the same Q-value function, Q_{old}, to gather data for a while: **we don't re-train Q_{old} with our new data.**
- Then, using **all** of our data shuffled (new and old), we train a **new** Q-value function, Q_{new}.

We can compare the two processes. First, typical Q-learning:

- Use Q, get **one data point**.
- Update Q with **newest data point**.
- **Repeat**.

And now, fitted Q-learning:

- Use Q_{old}, get **many new data points**.
- Train Q_{new} with **all data**.
- Replace Q_{old} with Q_{new}.
- **Repeat**.

Using pseudocode:

Definition 45

Fitted Q-learning uses the following procedure:

FITTED-Q-LEARNING($\mathcal{A}, s_0, \gamma, \alpha, \epsilon, m$)

```
s = s_0                          # Initial state
D = { }                          # No data yet

Q(s, a) = 0                      # Initial Q-values

while True:

    D_new = gather_data(Q, m)    # Gather m points of data using Q_old
    D = D ∪ D_new                # Add new data to database

    D_train = convert_data(D)    # Convert (s, a, s', r) to (x_i, y_i)

    Q = NN_train(D_train)        # Train Q_new, ignore Q_old
```

"convert_data" turns each experience (s, a, s', r) into a data point (x_i, y_i):

- Input $x^{(i)}$: **state** and **action**

$$x^{(i)} = (s, a) \tag{11.14}$$

- Output $y^{(i)}$: **expected reward** (based on reward r, and new state s')

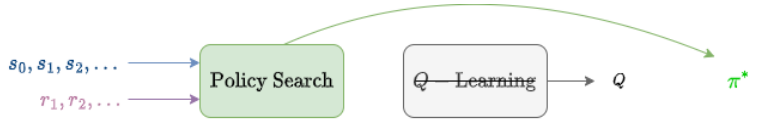
In other words, the Q-value, based on this data point.

$$y^{(i)} = r + \gamma \cdot \max_{a'} Q(s', a') \tag{11.15}$$

11.2.13 Policy Search

So far, we've been focused on methods for directly computing Q.

- But we could even go one step further: directly computing π.



Not only do we ignore T and R, but even Q.

Our strategy is to represent our policy as a **function** with **parameters** we can optimize.

Definition 46

In **policy search**, we represent our policy π as a *differentiable* **function** f, with **parameters** θ.

- This approach treats our policy like a **hypothesis**.

$$\pi(s) = f(s; \theta) = a$$

Using this approach, we can **optimize** our parameters θ, to get the greatest average reward.

- We need our function to be differentiable, for the same reasons as we needed for **gradient descent**.

One possible problem: often, we have a **discrete** action space. Our output will be a category: **not a continuous variable**.

- Discrete outputs aren't differentiable!

Our solution is the same as it was in classification: we use **probabilities**.

Concept 47

Rather than outputting the chosen action, a, we output the **probability** of that action.

- Because we chose our **action based on** our **state**, it's a **conditional probability**:

$$f(s, a; \theta) = \mathbf{P}\{a \mid s\} = \text{Prob of choosing action } a, \text{ given state } s$$

This allows us to output a **continuous** variable.

Once we have our continuous function, we can use **gradient descent**.

Key Equation 48

If θ is **low-dimensional**, we can use **numerical gradient descent**/ascent to train our policy:

- Slightly **adjust** θ_i by ε, see whether the **total reward** R is higher or lower: we approximate the derivative.

$$\frac{\partial R}{\partial \theta_i} \approx \frac{\Delta R}{\Delta \theta_i} = \frac{R(\theta_i + \varepsilon) - R(\theta_i)}{\varepsilon}$$

- Repeat for every θ_i term, to get a **numerical gradient**.

$$\nabla_\theta R = \begin{bmatrix} \partial R/\partial \theta_1 \\ \partial R/\partial \theta_2 \\ \vdots \\ \partial R/\partial \theta_n \end{bmatrix}$$

- Apply **gradient ascent** (we want to maximize R, rather than minimize $\mathcal{L}$)

$$\theta \Leftarrow \theta + \eta \cdot \nabla_\theta R$$

We could use **gradient descent** by choosing $\mathcal{L} = -R$.

For problems with higher-dimensional θ, this is often too slow/inefficient.

- Instead, we use other, more complex algorithms, like REINFORCE.

But these algorithms are often tricky.

Policy search works best in those lower-dimensional cases.

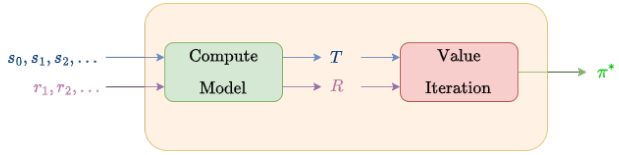
Concept 49

Policy search works best when

- The policy's **functional** form is known and **simple**.
- Estimating the MDP would be **difficult**.

11.3 Model-based RL

Rather than try to directly compute π or Q, we could also *re-use* our previous techniques: we just need to compute T and R.



Once we compute T and R, we can do value iteration, like we did before.

Notation 50

We want to **approximate** T and R.

We'll represent these approximations as $\widehat{T}$ and $\widehat{R}$, respectively.

11.3.1 Computing $\widehat{T}$

To compute $\widehat{T}$, let's remind ourselves: what does T represent?

Definition 51

Review from MDP Chapter, pt. 1:

The **transition function** T gives the probability of

- Entering state s',
- Given that we chose action a in state s

$$T(s, a, s') = \mathbf{P}\left\{S_t = s' \;\middle|\; S_{t-1} = s, \quad A_t = a\right\}$$

After a transition, we will be in **exactly one** new state s'.

So, we want to compute this probability.

Key Equation 52

We can **approximate** the probability of event E happening, by **counting** the number of times it does/doesn't occur:

$$\mathbf{P}\Big\{\mathsf{E}\Big\} = \frac{\text{Number of times } \mathsf{E} \text{ happens}}{\text{Total number of chances for } \mathsf{E} \text{ to happen}}$$

or,

$$\mathbf{P}\Big\{\mathsf{E}\Big\} = \frac{\#\mathsf{E}}{\#\text{Total events}}$$

Let's apply this to our situation: first, our "total events".

- We're computing the probability, **if** we chose action a, in state s.

If we were in a **different state**, or chose a **different action**, then that doesn't affect the probability.

$$\#\text{Total events} = \#(s, a) \tag{11.16}$$

- And we're looking for the chance that we **enter state** s'.

$$\#\mathsf{E} = \#(s, a, s') \tag{11.17}$$

So, we get:

$$\overbrace{\widehat{\mathsf{T}}(s, a, s') \approx \frac{\#(s, a, s')}{\#(s, a)}}^{\text{Not our final equation}} \tag{11.18}$$

But there's a **problem** with this equation.

11.3.2 The Laplace Correction

In particular, we have *two* problems with this equation.

- If we have **no data**, what's our estimated probability? $0/0$.
  - This is **nonsense**: we're dividing by zero.
- If we have **one data point**, and didn't get E, what is our probability? $0/1 = 0$.
  - If we have only one data point, why should we be so sure that E **never** happens?

Concept 53

The equation

$$\mathbf{P}\left\{E\right\} = \frac{\#E}{\#\text{Total events}}$$

Has two major flaws:

- It gives a **non-number** if we have no data.
- If we have **no events** E, it says that there's a **0% chance** that E will appear. Our model shouldn't be so confident.

Example: Imagine you flipped a coin 3 times, and happened to get heads 3 times. This will happen 1/8 of the time, on a fair coin.

- But our model has decided that there's a 0% chance of ever getting tails.

Let's solve each of these problems:

- We don't want to **divide by 0**. We need to add something to the **bottom**.
- We don't want to give a **0% chance of** E, when we don't have enough data. We'll add something to the **top**.

$$\widehat{T}(s, a, s') = \frac{\#(s, a, s') + b}{\#(s, a) + c} \tag{11.19}$$

How do we decide these constants? Well, let's return to the situation where we have **no data**.

$$\widehat{T}(s, a, s') = \frac{b}{c} \tag{11.20}$$

We want b/c to be our "**default**" assumption: what do we think are the odds of transitioning to state s', without any data?

- We have $|\mathcal{S}|$ different states, that we could **transition** to.
- Without any data, we have no reason to prefer one state over another. So, we assume all states to be **equally likely**.

If we split our probability evenly, we get:

$$\widehat{T}(s, a, s') = \frac{1}{|\mathcal{S}|} \tag{11.21}$$

We have our **correction terms**.

Definition 54

The **laplace correction** is an adjustment to our **probability equation**, that solves the problems of

- Dividing by 0
- Computing probability to be 0, with very low data

The solution is to set a **default** probability for an event: we split probability **evenly** between all of our N **possible outcomes**.

$$\mathbf{P}\left\{E \;\middle|\; \text{No data}\right\} = \frac{1}{N}$$

Applying this to our general equation, we get

$$\mathbf{P}\left\{E\right\} \approx \frac{1 + \#E}{N + \#\text{Total}}$$

As we gather more data, this correction term gradually **vanishes**.

Example: Let's say we have **5 possible outcomes**, and the odds of our event E are 40% (0.4).

And let's say our data exactly matches our probability, just to make things easier.

- We'll compare the prediction for 5,50, and 500 data points.

$$\frac{1+2}{5+5} = 0.3 \qquad \frac{1+20}{5+50} \approx 0.381 \qquad \frac{1+200}{5+500} \approx 0.398$$

- As we get more data, the laplace correction becomes less and less important.

Now, we can show our approximation for T.

Key Equation 55

Our **approximation** for the transition function T is given by the equation

$$\widehat{T}(s, a, s') = \frac{\#(s, a, s') + 1}{\#(s, a) + |\mathcal{S}|}$$

11.3.3 Computing $\widehat{R}$

Our reward function R, on the other hand, is much simpler to "approximate", because it's **deterministic**:

- The same state-action pair (s, a) will **always give the same reward**.
- So, we don't have to approximate our reward: if we get our reward once, we know **exactly** what it'll be.

Key Equation 56

Our "**approximation**" for the reward function R comes directly from our observations:

$$\widehat{R}(s, a) = r_t \qquad \text{if} \quad s_{t-1} = s, \quad a_t = a$$

This isn't really an approximation: it gives our **exact reward**.

$$\widehat{R}(s, a) \quad = \quad r_t \quad = \quad R(s, a)$$

In some situations, our reward might not be deterministic.

- In which case, we can compute the reward probability function, or the expected reward for our state-action pair.

11.3.4 Solving our MDP

Once we've computed our approximations $\widehat{T}$ and $\widehat{R}$, we can now construct the "approximated MDP":

$$\text{MDP}\left(\mathcal{S}, \mathcal{A}, \widehat{T}, \widehat{R}, \gamma\right) \tag{11.22}$$

And we can just solve it like any other MDP, using a technique like **value iteration**.

Definition 57

Our **model-based RL algorithm** has three basic parts:

- Computing our model $\left(\mathcal{S}, \mathcal{A}, \widehat{T}, \widehat{R}, \gamma\right)$.

$$\widehat{T}(s, a, s') = \frac{\#(s, a, s') + 1}{\#(s, a) + |\mathcal{S}|}$$

$$\widehat{R}(s, a) = r_t \qquad \text{if} \quad s_{t-1} = s, \quad a_t = a$$

- Using that model to do **value iteration**.

$$Q(s, a) = \widehat{R}\left(s, a\right) + \gamma \sum_{s'} \widehat{T}\left(s, a, s'\right) \cdot \max_{a'}\left(Q(s', a')\right)$$

- Using Q-values to find the optimal policy.

$$\pi^*(s) = \arg\max_a\left(Q(s, a)\right)$$

The approach requires us to approximate T and R for every possible combinations of input variables.

- So we can't use it if our state space is too large.

Concept 58

Model-based RL algorithms work best when we have a **small, discrete** state space $\mathcal{S}$.

- They're difficult to generalize to **large, or continuous** state spaces.

11.4 Bandit Problems

Here, we'll move away from our MDP framework.

- We'll consider a different kind of problem: one without **states**.

11.4.1 Slot machines

Here's our general idea: we have k different **choices** we can make. We want to **explore** each option, and figure out which one is the best.

No states here: just actions you can take.

- This sounds simple: we just try each option **once**, and pick the best one.
- But it's not so easy: each choice has a **randomized** outcome.
  - And each lever has different odds of giving you a particular reward.

We can think of this problem like a "**slot machine**" with k levers: each one has different odds of giving you a reward.

Slot machines have, in the past, been called "one-armed bandits", because they take your money. This is why we call these "bandit problems".

Concept 59

In a **bandit problem**, you have a set of k independent **actions** you can choose from.

- Each action will give you a **randomized** reward.

You goal is to maximize the **total rewards** you get (while training!)

Again, note that **states** have been completely removed from the problem.

11.4.2 Formalizing the Bandit Problem

We'll define each part of the bandit problem.

- First, we'll need our "options", or **actions** $\mathcal{A}$.
- What are the possible **rewards** we can get? That's our set $\mathcal{R}$.

$$a \in \mathcal{A} \qquad r \in \mathcal{R}$$

Finally, we need the **probability** of getting a reward, if we take an action. This is similar to the **transition function** T:

- Rather than returning our next state s', it instead gives us the **odds** of ending up in state s'.

If we take **action** a in **state** s.

In the same spirit, we'll have a function R_p, which gives us the **probability** of getting **reward** r, from **action** a.

$$R_p(a, r) = \mathbf{P}\Big\{\text{Getting } \textbf{reward } r \text{ from } \textbf{action } a\Big\} \tag{11.23}$$

Using conditional notation,

$$R_p(a, r) = \mathbf{P}\Big\{r \;\Big|\; a\Big\} \tag{11.24}$$

Based on our inputs and outputs, we can write this with **function notation**:

$$R_p : \Big(\mathcal{A} \times \mathcal{R}\Big) \to \Big[0, 1\Big] \tag{11.25}$$

$R_p : (\mathcal{A} \times \mathcal{R}) \to \mathbb{R}$, where $\mathbb{R}$ are real numbers, is also acceptable.

These are the three parts of our bandit problem.

Definition 60

A **bandit problem** has three parts:

- A set of actions $\mathcal{A}$
- A set of rewards $\mathcal{R}$
- A **reward-probability** function

$$R_p : \Big(\mathcal{A} \times \mathcal{R}\Big) \to \Big[0, 1\Big]$$

That tells us the **chance** of getting reward r, from action a.

Bandit problems are very, very important to reinforcement learning, and computer science.

- But we won't go through solutions/theorems here.

11.4.3 k-armed bandit problem

The simplest version of this problem is called the **k-armed bandit problem**:

- Every action ("arm") will either provide a reward ($r = 1$) or not ($r = 0$).

So, each action varies only by the **chance** that you get a reward.

In other words, we don't have "different types" of rewards.

Definition 61

The k-**armed bandit problem** is a simplified bandit problem, where

- You either get a simple **reward**, or you get nothing: $\mathcal{R} = \{0, 1\}$
- You have k **actions** to choose from: $|\mathcal{A}| = k$

11.4.4 Exploration vs. Exploitation

In bandit problems, you have to balance **exploration vs. exploitation**:

- **Exploration**: Do we want to improve our **estimate**, $\widehat{R}_p$? The better our estimate is, the more likely we are to make optimal choices, moving forward.
- **Exploitation**: Do you want to maximize your rewards, **based on** $\widehat{R}_p$? You'll get more benefits short-term than if you keep exploring.

If you want to maximize your rewards, while still **learning**, you can't just explore, and you can't "exploit" blindly without data.

There's lots of interesting details we'll skip here, but the basic idea is:

Concept 62

The longer your horizon h (or the larger γ is), the longer you should continue to **explore**.

- The same amount of exploring takes up **smaller fraction** of your time: so, it takes away **less** of the exploitation reward.

Example: Consider a simplified version: you have h **turns** to play.

You spend n turns "exploring", and then $h - n$ turns "exploiting". You get 0 reward for exploring.

This example uses a lot of huge simplifications. Let's say these as "average" benefits for exploring/exploiting.

- You get \$10 for exploiting if you explored a **little** ($n = 3$)
- You get \$15 for exploiting if you explored **a lot** ($n = 10$)

If you have only a little time ($h = 15$), it's not really worth it to explore more.

$$\overbrace{(15-3)\cdot 10 = 120}^{\text{More time to exploit}} \qquad \overbrace{(15-10)\cdot 15 = 75}^{\text{More reward for exploiting}} \tag{11.26}$$

If you have plenty of time ($h = 100$), it's definitely worth it.

$$(100-3)\cdot 10 = 970 \qquad (100-10)\cdot 15 = 1350 \tag{11.27}$$

Of course, this exploration/exploitation process is fairly sensitive to "luck": whether we get better or worse outcomes than the average.

Which can happen very easily, with small sample sizes.

The longer we can train, the more likely we are to be near the true average.

Concept 63

"**Bad luck**" (getting low rewards for a valuable lever) is often more harmful than "**good luck**" (getting high rewards for a bad lever):

- If you get bad luck, you're **less likely** to keep trying that lever: you **won't find out** it's a good lever.
- If you get good luck, you'll probably **keep trying** that (seemingly profitable) lever: you'll get **lots of data** to learn that it isn't as good as you thought.

11.4.5 Contextual Bandit Problems

Our typical bandit problems lack the concept of a "**state**". However, if we *do* need states, we can use a *contextual bandit problem*:

Definition 64

In a **contextual bandit problem**, we re-introduce **states** $\mathcal{S}$.

- Each state s has its own bandit problem.

11.5 Terms

- MDP (Review)
- Value function (Review)
- Q-value function (Review)
- Reinforcement Learning
- Learner
- Environment
- Supervised Learning (Review)
- Unsupervised Learning (Review)
- Model-based RL
- Model-free RL
- Q-learning
- Value iteration (Review)
- Learning rate α
- ϵ-greedy strategy
- Exploration vs. Exploitation
- Tabular Q-learning
- Deep Q-learning
- Temporally Correlated
- Catastrophic forgetting
- Experience Replay
- Replay Buffer
- Sliding Window
- Fitted Q-learning
- Policy search
- Conditional Probability (Review)
- Numerical Gradient Descent
- $\widehat{\mathsf{T}}$

- $\widehat{R}$
- Laplace Correction
- Bandit Problem
- Reward-probability function R_p
- k-armed bandit problem
- Contextual Bandit Problem