



In this chapter

- You understand the inherent challenges of training reinforcement learning agents with non-linear function approximators.
- You create a deep reinforcement learning agent that when trained from scratch with minimal adjustments to hyperparameters can solve different kinds of problems.
- You identify the advantages and disadvantages of using value-based methods when solving reinforcement learning problems.

“ Human behavior flows from three main sources:
desire, emotion, and knowledge. ”

— Plato

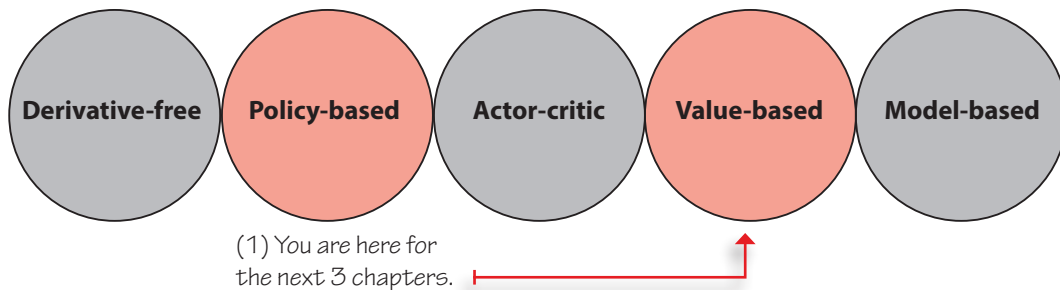
A philosopher in Classical Greece
and Founder of the Academy in Athens

We have made a great deal of progress so far, and you are ready to grok deep reinforcement learning truly. In chapter 2, you learned to represent problems in a way reinforcement learning agents can solve using Markov Decision Processes (MDP.) In chapter 3, you developed algorithms that solve these MDPs. That is agents that find optimal behavior in sequential decision-making problems. In chapter 4, you learned about algorithms that solve one-step MDPs without having access to these MDPs. These problems are uncertain because the agents do not have access to the MDP. Agents learn to find optimal behavior through trial-and-error learning. In chapter 5, we mixed these two types of problems: sequential and uncertain, so we explore agents that learn to evaluate policies. Agents didn't find optimal policies but were able to evaluate policies, were able to estimate value functions accurately. In chapter 6, we studied agents that find optimal policies on sequential decision-making problems under uncertainty. These agents go from random to optimal by merely interacting with their environment and deliberately gathering experiences for learning. In chapter 7, we learned about agents that are even better at finding optimal policies by getting the most out of their experiences.

Chapter 2 is a foundation for all chapters in this book use. Chapter 3 is about planning algorithms that deal with sequential feedback. Chapter 4 is about bandit algorithms that deal with evaluative feedback. Chapters 5, 6, and 7 are about RL algorithms, algorithms that deal with feedback that is simultaneously sequential and evaluative. This type of problem is what people refer to as 'tabular' reinforcement learning. Starting from this chapter, we dig into the details of deep reinforcement learning.

More specifically, in this chapter, we begin our incursion on the use of deep neural networks for solving reinforcement learning problems. In deep reinforcement learning, there are different ways of leveraging the power of highly non-linear function approximators, such as deep neural networks. They are value-based, policy-based, actor-critic, model-based, and gradient-free methods. This chapter goes in-depth on value-based deep reinforcement learning methods.

Types of algorithmic approaches you learn about in this book



The kind of feedback deep reinforcement learning agents use

In deep reinforcement learning, we build agents that are capable of learning from feedback that is simultaneously evaluative, sequential, and sampled. I've been restating this throughout the book because you need to understand what that means.

In the first chapter, I mentioned that deep reinforcement learning is about complex sequential decision-making problems under uncertainty. You probably thought, "what a bunch of words." But as I promised, all these words mean something. "Sequential decision-making problems" is what you learned about in chapter 3. "Problems under uncertainty" is what you learned about in chapter 4. In chapters 5, 6, and 7, you learned about "sequential decision-making problems under uncertainty." In this chapter, we add the "complex" part back to that whole sentence. Let's use this introductory section to review one last time the three types of feedback a deep reinforcement learning agent uses for learning.



BOIL IT DOWN

Kinds of feedback in deep reinforcement learning

	Sequential (as opposed to one-shot)	Evaluative (as opposed to supervised)	Sampled (as opposed to exhaustive)
Supervised Learning	×	×	✓
Planning (Chapter 3)	✓	×	×
Bandits (Chapter 4)	×	✓	×
'Tabular' reinforcement learning (Chapters 5, 6, 7)	✓	✓	×
Deep reinforcement learning (Chapters 8, 9, 10, 11, 12)	✓	✓	✓

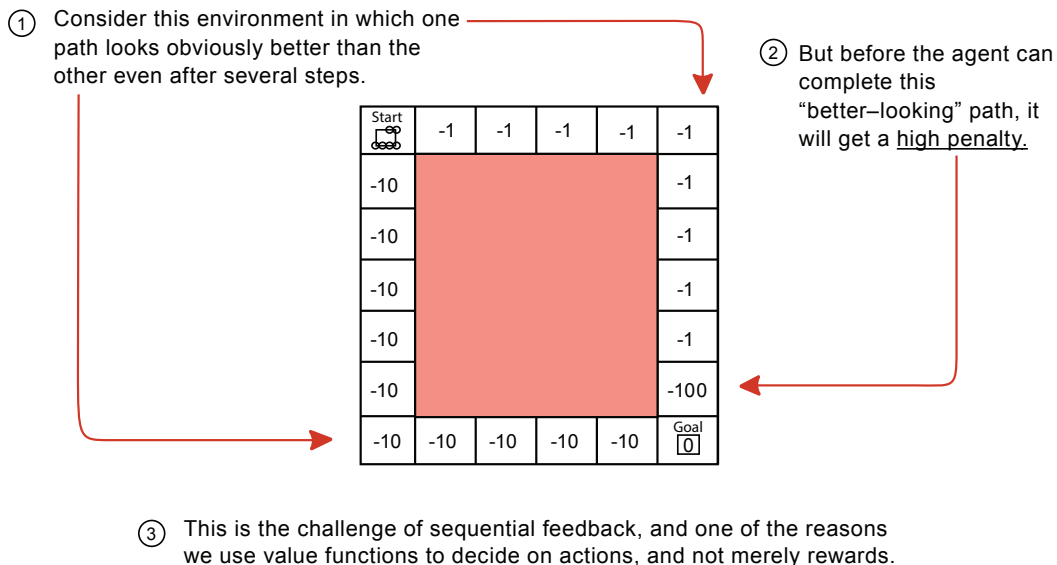
Deep reinforcement learning agents deal with sequential feedback

Deep reinforcement learning agents have to deal with sequential feedback. One of the main challenges of sequential feedback is that your agents can receive delayed information.

You can imagine a chess game in which you make a few wrong moves early on, but the consequences those wrong moves only manifest at the end of the game when and if you materialize a loss.

Delayed feedback makes it tricky to interpret the source of the feedback. Sequential feedback gives rise to the temporal credit assignment problem, which is the challenge of determining which state, action, or state-action pair is responsible for a reward. When there is a temporal component to a problem and actions have delayed consequences, it becomes challenging to assign credit for rewards.

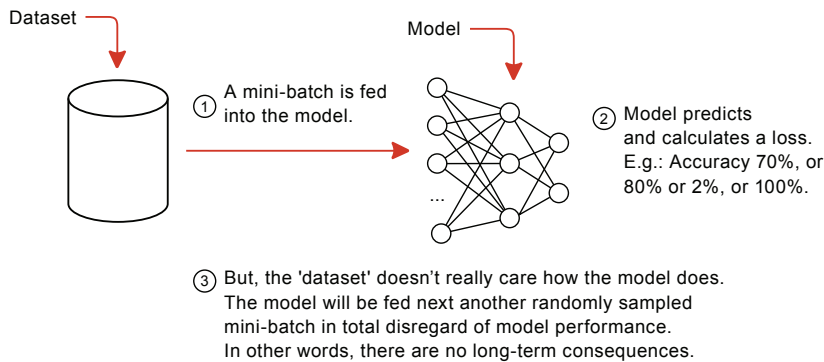
Sequential feedback



But, if it is not sequential, what is it?

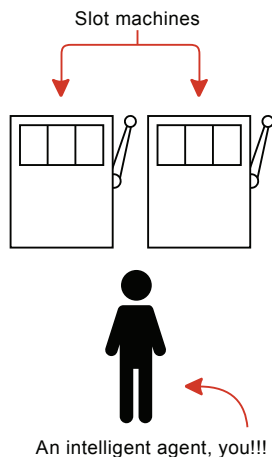
The opposite of delayed feedback is immediate feedback. In other words, the opposite of sequential feedback is one-shot feedback. In problems that deal with one-shot feedback, such as supervised learning or multi-armed bandits, decisions do not have long-term consequences. For example, in a classification problem, classifying an image, whether correctly or not, has no bearing on future performance; for instance, the images presented to the model next are not any different whether the model classified correctly or not the previous batch. In DRL, this sequential dependency exists.

Classification problem



Moreover, in Bandit problems, there is also no long-term consequence, though perhaps a bit harder to see why. Bandits are one-state one-step MDPs in which episodes terminate immediately after a single action selection. Therefore, actions do not have long-term consequences in the performance of the agent during that episode.

2-armed bandit



Note: We assume slot machines have a stationary probability of pay off, meaning the probability of payoff will not change with a pull, which is likely incorrect for real slot machines.

- ① When you go to a casino and play the slots machines, your goal is to find the machine that "pays" the most, and then stick to that arm.
- ② In bandit problems, we assume the probability of pay off stays the same after every pull. This makes it a one-shot-kind of problem.

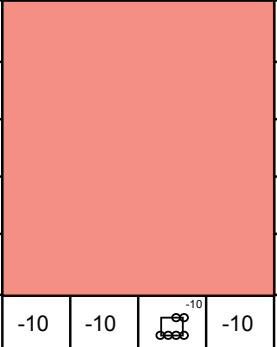
Deep reinforcement learning agents deal with evaluative feedback

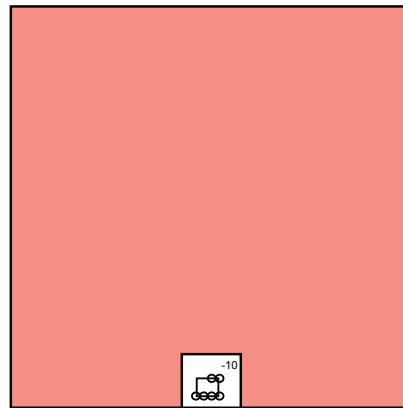
The second property we learned about is that of evaluative feedback. Deep reinforcement learning, 'tabular' reinforcement learning, and bandits, all deal with evaluative feedback. The crux of evaluative feedback is that the goodness of the feedback is only relative because the environment is uncertain. We do not know the actual dynamics of the environment; we do not have access to the transition function and reward signal.

As a result, we must explore the environment around us to find out what's out there. The problem is, by exploring, we miss capitalizing on our current knowledge and, therefore, likely accumulate regret. Out of all this, the exploration-exploitation tradeoff arises. It's a constant by-product of uncertainty. While not having access to the model of the environment, we must explore to gather new information or improve on our current information.

Evaluative feedback

- ① To understand the challenge of evaluative feedback you must be aware that agents don't see entire maps such as this one

Start 0	-1	-1	-1	-1	-1
-10					-1
-10					-1
-10					-1
-10					-1
-10					-100
-10	-10	-10	-10	-10	Goal 0



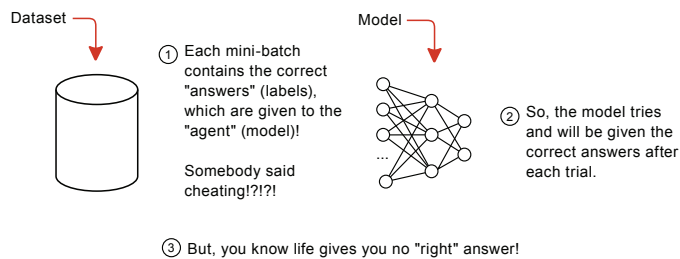
- ② Instead, they only see the current state and reward such as this one.

- ③ So, is that -10 bad? Is it good?

But, if it is not evaluative, what is it?

The opposite of evaluative feedback is supervised feedback. In a classification problem, your model receives supervision; that is, during learning, your model is given the correct labels for each of the samples provided. There is no guessing. If your model makes a mistake, the correct answer is provided immediately after. What a good life!

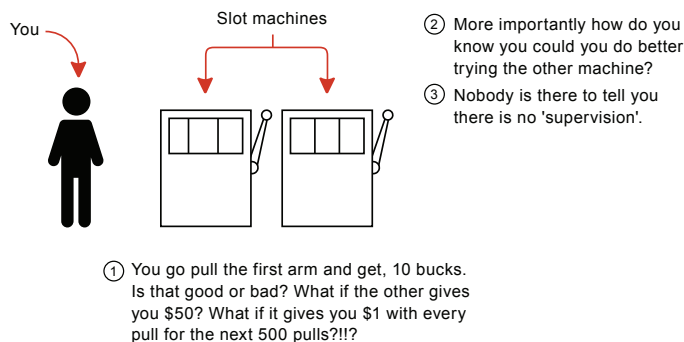
Classification is "supervised"



The fact that correct answers are given to the learning algorithm makes supervised feedback much easier to deal with than evaluative feedback. That is a clear distinction between supervised learning problems and evaluative-feedback problems, such as multi-armed bandits, 'tabular' reinforcement learning, and deep reinforcement learning.

Bandit problems may not have to deal with sequential feedback, but they do learn from evaluative feedback. That's the core issue bandit problems solve. When under evaluative feedback, agents must balance exploration vs. exploitation requirements. Now, if the feedback is evaluative and sequential at the same time, the challenge is even more significant. Algorithms must simultaneously balance immediate- and long-term goals and the gathering and utilization of information. Both, 'tabular' reinforcement learning, and deep reinforcement learning learn from feedback that is simultaneously sequential and evaluative.

Bandits deal with evaluative feedback



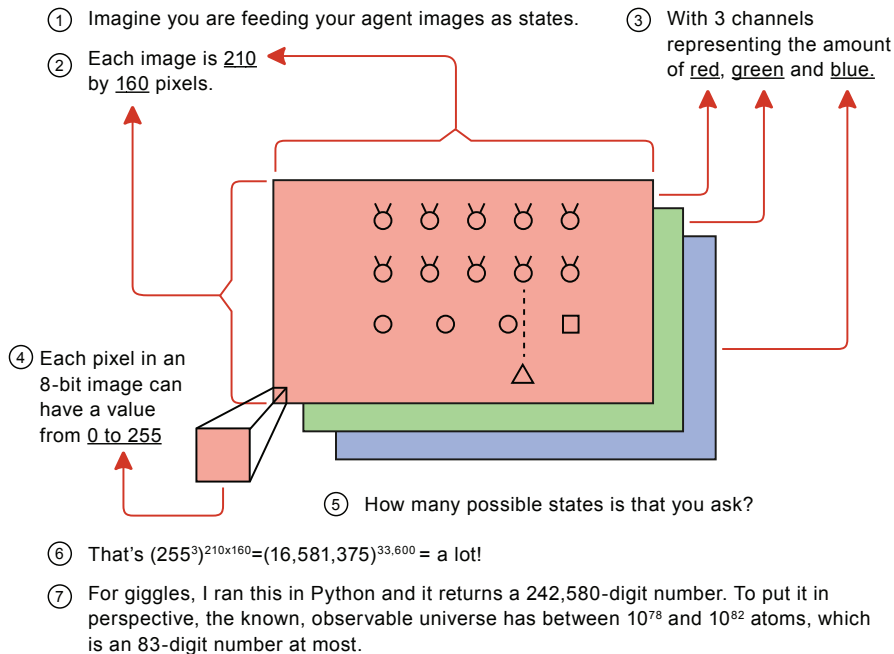
Deep reinforcement learning agents deal with sampled feedback

What differentiates deep reinforcement learning from 'tabular' reinforcement learning is the complexity of the problems. In deep reinforcement learning, agents are unlikely to be able to sample all possible feedback exhaustively. That means that agents need to generalize using the gathered feedback and come up with intelligent decisions based on that generalization.

Think about it. You can't expect exhaustive feedback from life. You can't be a doctor and a lawyer and an engineer all at once. At least not if you want to be good at any of these. So, you must use the experience you gather early on to make more intelligent decisions for your future. It's basic. Were you good at math in high-school? Great, then, pursue a math-related degree. Were you better at the arts? Then, go to pursue that path. Generalizing helps you narrow your path going forward by helping you find patterns, make assumptions, and connect the dots, that help you reach your optimal self.

By the way, supervised learning deals with sampled feedback. Indeed, the core challenge in supervised learning is to learn from sampled feedback: to be able to generalize to unseen samples, which is something neither multi-armed bandit nor 'tabular' reinforcement learning problems do.

Sampled feedback




But, if it is not sampled, what is it?

The opposite of sampled feedback is exhaustive feedback. To exhaustively sample environments means agents have access to all possible samples. 'Tabular' reinforcement learning, and bandits agents, for instance, only need to sample for long enough to gather all necessary information for optimal performance. To be able to gather exhaustive feedback is also why there are optimal convergence guarantees in 'tabular' reinforcement learning. Common assumptions, such as "infinite data" or "sampling every state-action pair infinitely often," are reasonable assumptions in small grid worlds with finite state and action spaces.

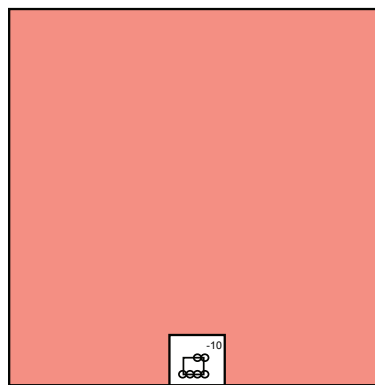
Sequential, evaluative and exhaustive feedback

- ① Again, this is what sequential feedback looks like



Start 0	-1	-1	-1	-1	-1
-10					-1
-10					-1
-10					-1
-10					-1
-10					-100
-10	-10	-10	-10	-10	Goal 0

- ② And this is what evaluative feedback looks like



- ③ But, given you have a discrete number of states and actions, you can assume exhaustively sampling the environment. In a small state and action spaces, things are easy in practice, and theory is doable. As the number of states and actions spaces increase, the need for function approximation becomes evident.

This dimension we haven't dealt with until now. In this book so far, we surveyed the 'tabular' reinforcement learning problem. 'Tabular' reinforcement learning learns from evaluative, sequential, and exhaustive feedback. But, what happens when we have more complex problems in which we cannot assume our agents will ever exhaustively sample environments? What if the state space is high-dimensional, such as a Go board with 10^{170} states? How about ATARI games with $(255^3)^{210 \times 160}$ at 60 Hz? What if the environment state space has continuous variables, such as a robotic arm indicating joint angles? How about problems with both high-dimensional and continuous states or even high-dimensional and continuous actions? These complex problems are the reason for the existence of the field of deep reinforcement learning.

Introduction to function approximation for reinforcement learning

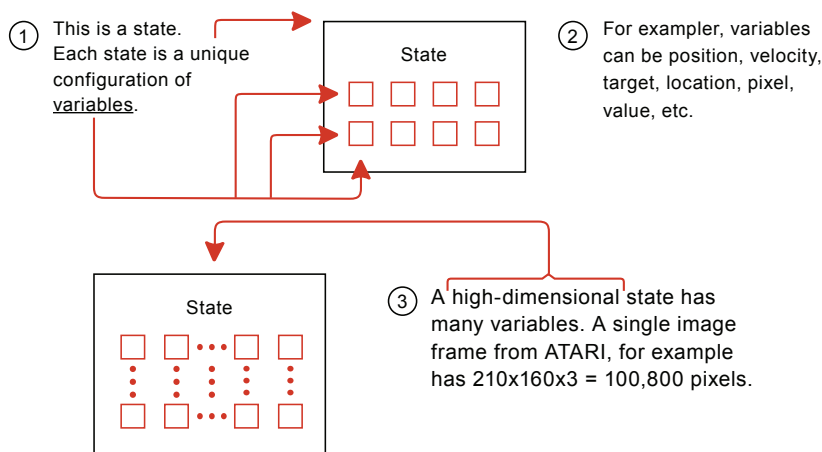
It's essential to understand why we use function approximation for reinforcement learning in the first place. It is common to get lost in words and pick solutions due to the hype. You know, if you hear "deep learning," you get more excited than if you hear non-linear function approximation, yet they are the same. That's human nature. It happens to me; it happens to many, I'm sure. But our goal is to remove the cruft and simplify our thinking.

In this section, I motivate the use of function approximation to solve reinforcement learning problems in general. Perhaps a bit more specific to value functions, than RL overall, but the underlying motivation applies to all forms of DRL.

Reinforcement learning problems can have high-dimensional state and action spaces

The main drawback of 'tabular' reinforcement learning is that the use of a table to represent value functions is no longer practical in complex problems. Environments can have high-dimensional state spaces, meaning that the number of variables that comprise a single state is vast. For example, ATARI games described above are high dimensional because of the 210 by 160 pixels, and the 3 color channels. Regardless of the values that these pixels can take when we talk about 'dimensionality,' we are referring to the number of variables that make up a single state.

High-dimensional state spaces



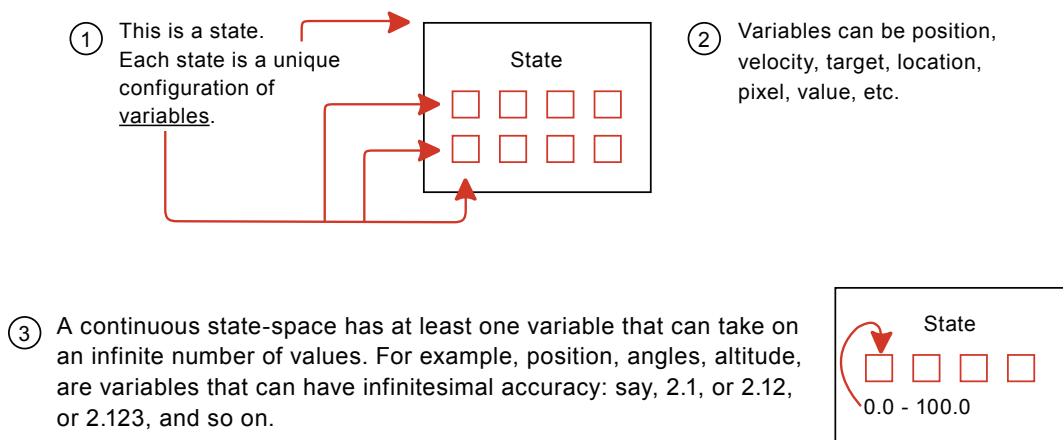
Reinforcement learning problems can have continuous state and action spaces

Moreover, environments can additionally have continuous variables, meaning that a variable can take on an infinite number of values. Now, to clarify, state and action spaces can be high-dimensional with discrete variables, they can be low-dimensional with continuous variables, and so on.

Now, even if the variables are not continuous and, therefore, not infinitely large, they can still take on a large number of values to make it impractical for learning without function approximation. This is the case of ATARI, for instance, where each image-pixel can take on 256 values (0-255 integer values.) There you have a finite state-space, yet large enough to require function approximation for any learning to occur.

But, sometimes, even low-dimension state spaces can be infinitely large state spaces. For instance, imagine a problem in which only the x, y, z coordinates of a robot compose the state-space. Sure, a three-variable state-space is a pretty low-dimensional state-space environment, but what if any of the variables is provided in continuous form, that is, that variable can be of infinitesimal precision. Say, it could be a 1.56, or 1.5683, or 1.5683256, and so on. Then, how do you make a table that takes all these values into account? Yes, you could discretize the state space, but let me save you some time and get right to it: you need function approximation.

Continuous state spaces



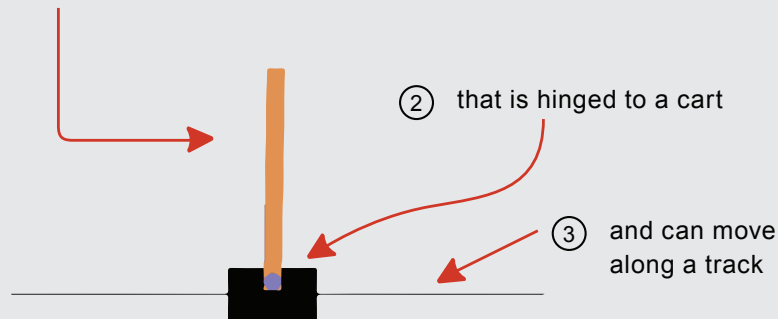


CONCRETE EXAMPLE

The Cart-Pole environment

The cart-pole environment is a classic in reinforcement learning. The state space is low-dimensional but continuous, making it an excellent environment for developing algorithms; training is fast, yet still somewhat challenging, and function approximation can help.

① The cart-pole environment consists on balancing a pole



Its state space is comprised of four variables:

- The cart position on the track (x axis) with a range from -2.4 to 2.4
- The cart velocity along the track (x axis) with a range from -inf to inf
- The pole angle with a range of ~ 40 degrees to ~ 40 degrees
- The pole velocity at the tip with a range of -inf to inf

There are two available actions in every state:

- Action 0 applies a -1 force to the cart (push it left)
- Action 1 applies a +1 force to the cart (push it right)

You reach a terminal state if:

- The pole angle is more than 12 degrees away from the vertical position
- The cart center is more than 2.4 units from the center of the track
- The episode count reaches 500 time steps (more on this later)

The reward function is:

- +1 for every time step

There are advantages when using function approximation

I'm sure you get the point that in environments with high-dimensional or continuous state spaces, there are no practical reasons for not using function approximation. In earlier chapters, we discussed planning and reinforcement learning algorithms. All of those methods represent value functions using tables.

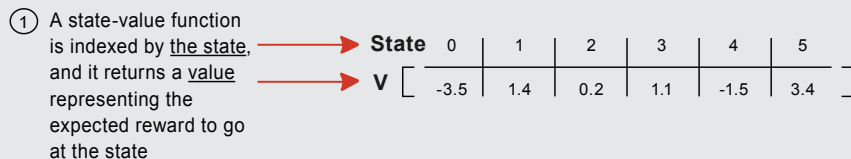


REFRESH MY MEMORY

Algorithms such as Value Iteration and Q-learning use tables for value functions

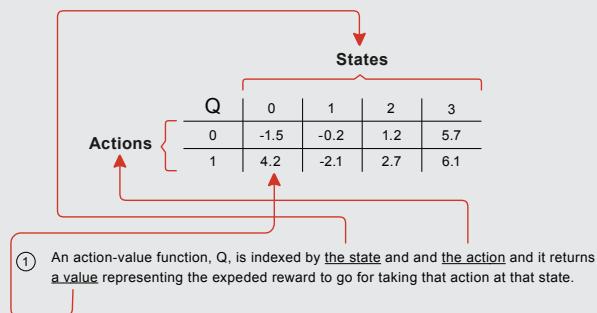
Value iteration is a method that takes in an MDP and derives an optimal policy for such MDP by calculating the optimal state-value function, v^* . To do this, value iteration keeps track of the changing state-value function, v , over multiple iterations. In value iteration, the state-value function estimates are represented as a vector of values indexed by the states. This vector is stored with a lookup table for querying and updating estimates.

A state-value function



The Q-learning algorithm does not need an MDP and does not use a *state-value function*. Instead, in Q-learning, we estimate the values of the optimal *action-value function*, q^* . Action-value functions are not vectors, but, instead, are represented by matrices. These matrices are 2-d tables indexed by states and actions.

An action-value function





BOIL IT DOWN

Function approximation can make our algorithms more efficient

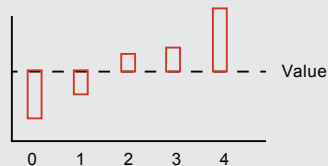
In the cart-pole environment, we want to use generalization because it is a more efficient use of experiences. With function approximation, agents learn and exploit patterns with less data (and perhaps faster).

A state-value function with and without function approximation

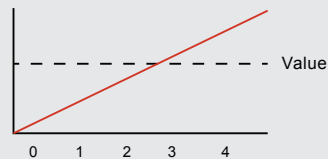
- ① Imagine this state-value function.

$V = [-2.5, -1.1, 0.7, 3.2, 5.6]$

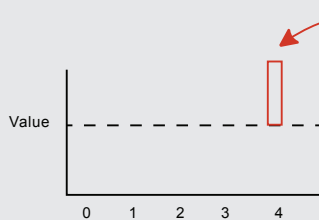
- ② Without function approximation, each value is independent.



- ③ With function approximation the underlying relationship of the states can be learned and exploited.

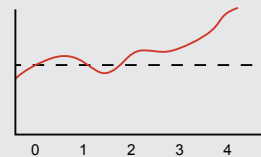


- ④ The benefit of using function approximation is particularly obvious if you imagine these plots after just a single update.



- ⑤ Without function approximation, the update only changes one state.

- ⑥ With function approximation, the updates changes multiple states.



- ⑦ **Note:**

Of course, this is a simplified example, but it helps illustrate what's happening. What would be different in 'real' examples?

First, if we approximate an action-value function, Q , we would have to add another dimension.

Also, with non-linear function approximator, such as a neural network, more complex relationship can be discovered.

While the inability of Value Iteration and Q-learning to solve problems with sampled feedback make them impractical, the lack of generalization makes them inefficient. What I mean by this is that we could find ways to use tables in environment with continuous-variable states, but we would be paying a price by doing so. Discretizing values could indeed make tables possible, for instance. But, even if we could engineer a way to use tables and store value functions, by doing so, we'd be missing out on the advantages of generalization.

For example, in the cart-pole environment, function approximation would help our agents learn a relationship in the x distance. Agents would likely learn that being 2.35 units away from the center is a bit more dangerous than being 2.2 away. We know that 2.4 is the x boundary. This additional reason for using generalization is not to be understated. Value functions often have underlying relationships that agents can learn and exploit. Function approximators, such as neural networks, can discover these underlying relationships.



BOIL IT DOWN

Reasons for using function approximation

Our motivation to using function approximation is not only to solve problems that are not solvable otherwise, but also to solve problems more *efficiently*.

NFQ: The first attempt to value-based deep reinforcement learning

The following algorithm is called **Neural Fitted Q Iteration** (NFQ), and it is probably one of the first algorithms to successfully use neural networks as a function approximation to solve reinforcement learning problems.

For the rest of this chapter, I discuss several components most value-based deep reinforcement learning algorithms have. I want you to see it as an opportunity to decide on different parts that we could've used. For instance, when I introduce using a loss function with NFQ, I discuss a few alternatives. My choices are not necessarily the choices that were made when the algorithm was originally introduced. Likewise, when I choose an optimization method, whether RMSprop or Adam, I give some reason why I use what I use, but more importantly, I give you context so you can pick and choose as you see fit.

What I hope you notice is that my goal is not only to teach you this specific algorithm but, more importantly, to show you the different places where you could try different things. Many RL algorithms feel this "plug-and-play" way, so pay attention.

First decision point: Selecting a value function to approximate

Using neural networks to approximate value functions can be done in many different ways. To begin with, there are many different value functions we could approximate.



REFRESH MY MEMORY

Value functions

You've learned about the following value functions:

- The state-value function $v(s)$
- The action-value function $q(s,a)$
- The action-advantage function $a(s,a)$

You probably remember that the *state-value function* $v(s)$, though useful for many purposes, is not sufficient on its own to solve the control problem. Finding $v(s)$ helps you know how much expected total discounted reward you can obtain from state s and using policy π thereafter. But, in order to determine which action to take with a V-function, you also need the MDP of the environment so that you can do a one-step lookahead and take into account all possible next states after selecting each action.

You likely also remember that the *action-value function* $q(s,a)$ allows us to solve the control problem, so it is more like what we need in order to solve the cart-pole environment: in the cart-pole environment we are looking to learn the values of *actions* for all states in order to balance the pole by controlling the cart. If we had the *values of state-action pairs*, we could differentiate the actions that would lead us to, either gain information, in the case of an exploratory action, or maximize the expected return, in the case of a greedy action.

I want you to notice too, that what we want to estimate is the optimal action-value function and not just simply an action-value function. However, as we learned in the generalized policy iteration pattern, we can do on-policy learning using an epsilon-greedy policy and estimating its values directly, or we can do off-policy learning and always estimate the policy greedy with respect to the current estimates, which then becomes an optimal policy.

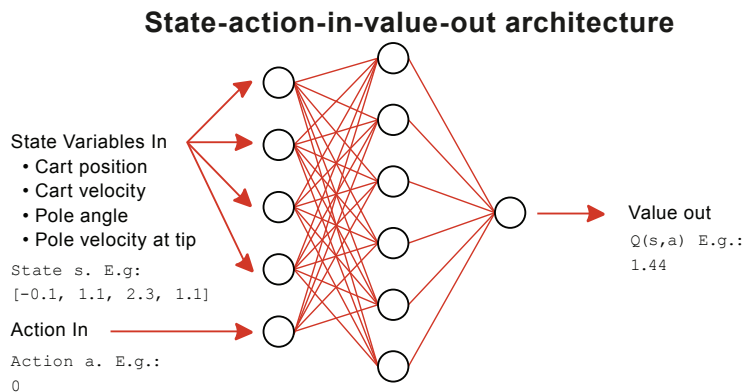
Lastly, we also learned about the *action-advantage function* $a(s,a)$, which can help us differentiate between values of different actions, and it also lets us easily see how much better than average an action is.

We'll study how to use the $v(s)$ and $a(s)$ functions in a few chapters. For now, let's settle on estimating the *action-value function* $q(s,a)$, just like in Q-learning. We refer to the *approximate action-value function* estimate as $Q(s,a; \theta)$; that means the Q estimates are parameterized by θ , the weights of a neural network, a state s and an action a .

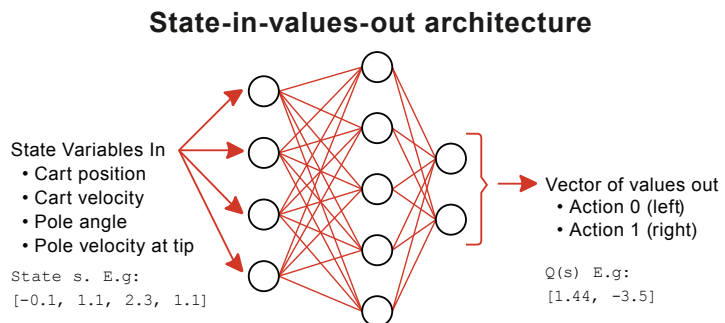
Second decision point: Selecting a neural network architecture

We settled on learning the *approximate the action-value function* $Q(s,a; \theta)$. But although I suggested the function should be parameterized by θ , s , and a , that doesn't have to be the case. The next component we discuss is the neural network architecture.

When we implemented the Q-learning agent, you noticed how the matrix holding the action-value function was indexed by state and action pairs. A straightforward neural network architecture is to input the state (the 4 state variables in the cart-pole environment), and the action to evaluate. The output would then be one node representing the Q-value for that state-action pair.



This architecture would work just fine for the cart-pole environment. But, a more efficient architecture consists of only inputting the state (4 for the cart-pole environment) to the neural network and outputting the Q-values for all the actions in that state (2 for the cart-pole environment). This is clearly advantageous when using exploration strategies such as epsilon-greedy or SoftMax, because having to do only one pass forward to get the values of all actions for any given state yields a high-performance implementation, more so in environments with a large number of actions.



For our NFQ implementation, we use the *state-in-values-out architecture*: that is 4 input nodes and 2 output nodes for the cart-pole environment.



I SPEAK PYTHON

Fully-Connected Q-function (state-in-values-out)

```
class FCQ(nn.Module):
    def __init__(self,
                  input_dim,
                  output_dim,
                  hidden_dims=(32, 32),
                  activation_fc=F.relu):
        super(FCQ, self).__init__()
        self.activation_fc = activation_fc

        self.input_layer = nn.Linear(input_dim,
                                     hidden_dims[0])

        self.hidden_layers = nn.ModuleList()
        for i in range(len(hidden_dims)-1):
            hidden_layer = nn.Linear(
                hidden_dims[i], hidden_dims[i+1])
            self.hidden_layers.append(hidden_layer)

        self.output_layer = nn.Linear(
            hidden_dims[-1], output_dim)

    def forward(self, state):
        x = state
        if not isinstance(x, torch.Tensor):
            x = torch.tensor(x,
                             device=self.device,
                             dtype=torch.float32)
            x = x.unsqueeze(0)

        x = self.activation_fc(self.input_layer(x))
        for hidden_layer in self.hidden_layers:
            x = self.activation_fc(hidden_layer(x))
        x = self.output_layer(x)
        return x
```

(1) Here you are just defining the input layer. See how we take in 'input_dim' and output the first element of the 'hidden_dims' vector.

(2) We then create the hidden layers. Notice how flexible this class is that allows you to change the number of layers and units per layer. Just pass a different tuple, say (64,32,16), to the 'hidden_dims' variable, and it will create a network with 3 hidden layers of 64, 32 and 16 units respectively.

(3) We then connect the last hidden layer to the output layer.

(4) In the forward function, we first take in the raw state and convert it into a tensor.

(5) We pass it through the input layer and then through the activation function.

(6) Then we do the same for all hidden layers.

(7) And finally for the output layer. Notice that we do not apply the activation function to the output but return it directly instead.

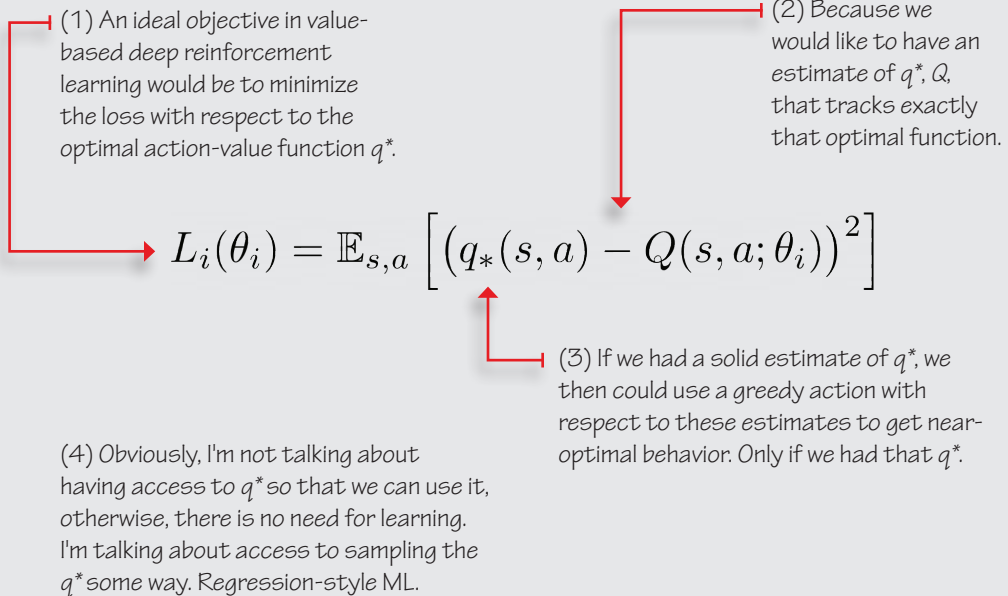
Third decision point: Selecting what to optimize

Let's pretend for a second that the cart-pole environment is a supervised learning problem. Say you have a dataset with states as inputs and a value function as labels. Which value function would you wish to have for labels?



SHOW ME THE MATH

Ideal objective



Of course, the dream labels for learning the optimal action-value function are the corresponding optimal q-values for the state-action input pair. That is exactly what the *optimal* action-value function $q^*(s,a)$ represents, as you know.

If we had access to the optimal action-value function, we would just use that, but if we had access to sampling the optimal action-value function, we could then minimize the loss between the approximate and optimal action-value functions, and that'd be it.

The optimal action-value function is what we are after.



REFRESH MY MEMORY

Optimal action-value function

(1) As a reminder, here is the definition of the optimal action-value function.

(2) This is just telling us that the optimal action-value function...

(3) ... is the policy that gives...

$$q_*(s, a) = \max_{\pi} \mathbb{E}_{\pi} [G_t | S_t = s, A_t = a], \forall s \in S, \forall a \in A(s)$$

(4) ... the maximum expected return...

(5) ... from each and every action in each and every state.

But why is this an impossible dream? Well, the visible part is we don't have the optimal action-value function $q^*(s, a)$, but to top that off, we cannot even sample these optimal q -values because we do not have the optimal policy either.

Fortunately, we can use the same principles learned in generalized policy iteration in which we alternate between policy-evaluation and policy-improvement processes to find good policies. But so that you know, because we are using non-linear function approximation, convergence guarantees no longer exist. It's the wild west in the "deep" world.

For our NFQ implementation, we do just that. We start with a randomly initialized action-value function (and implicit policy.) Then, evaluate the policy by sampling actions from it, as we learned in chapter 5. Then, improve it with an exploration strategy such as epsilon-greedy, as we learned in chapter 4. Finally, keep iterating until we reach the desired performance, as we learned in chapters 6 and 7.



BOIL IT DOWN

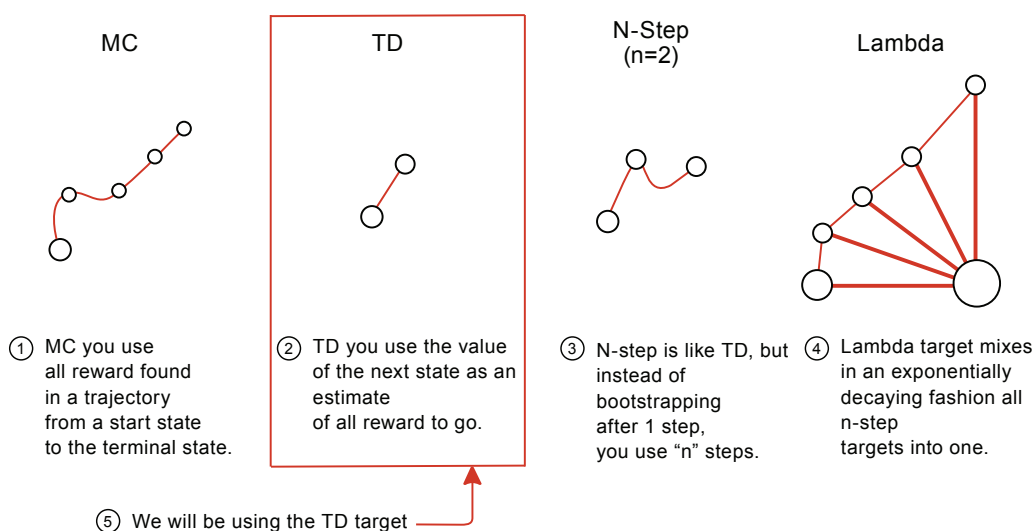
We can't use the ideal objective

We can't use the ideal objective because we don't have access to the optimal action-value function and don't even have an optimal policy to sample from. Instead, we must alternate between evaluating a policy (by sampling actions from it), and improving it (using an exploration strategy, such as epsilon-greedy). Just like you learned in chapter 6, in the generalized policy iteration pattern.

Fourth decision point: Selecting the targets for policy evaluation

There are multiple ways we can evaluate a policy. More specifically, there are different *targets* we can use for estimating the action-value function of a policy π . The core targets you learned about are the Monte-Carlo (MC) target, the Temporal-Difference (TD) target, the N-step target, and Lambda target.

MC, TD, N-step and Lambda targets



We could use any of these targets and get solid results, but this time our NFQ implementation, we keep it simple and use the TD target for our experiments.

Hopefully you remember that the TD targets can be either on-policy or off-policy depending on the way you bootstrap the target. The two main ways for bootstrapping the TD target are to either use the action-value function of the action the agent will take at the landing state, or alternatively, to use the value of the best action at the next state.

Often in the literature, these are called Sarsa target for the on-policy bootstrapping, and Q-learning target for the off-policy bootstrapping.



SHOW ME THE MATH

On-policy and off-policy TD targets

(1) Notice that both on-policy and off-policy targets estimate an action-value function.

(2) However, if we were to use the on-policy target, the target would be approximating the behavioral policy. In other words, the policy generating behavior and the policy being learned would be the same.

$$y_i^{Sarsa} = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}; \theta_i)$$

$$y_i^{Q-learning} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_i)$$

(3) This is not true for the off-policy target in which we always approximate the greedy policy, even if the policy generating behavior is not totally greedy.

In our NFQ implementation, we use the same off-policy TD target we used in the Q-learning algorithm. At this point, to get an objective function, we need to substitute the optimal action-value function $q^*(s,a)$, that we had as the ideal objective equation, by the Q-learning target.



SHOW ME THE MATH

The Q-learning target, an off-policy TD target

(1) In practice, an online Q-learning target would look something like this.

(2) Bottom line is we use the experienced reward, and the next state to form the target.

$$y_i^{Q-learning} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_i)$$

(4) But it is basically the same. We are using the expectation of experience tuples.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i) \right)^2 \right]$$

(3) We can plug in a more general form of this Q-learning target here.

(5) To minimize the loss.

(6) Now, when differentiating through this equation, it is important you notice the gradient doesn't involve the target.

(7) The gradient must only go through the predicted value. This is one common source of error.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$



I SPEAK PYTHON

Q-learning target

```
q_sp = self.online_model(next_states).detach()
```

(2) The 'detach' here is important. We should not be propagating values through this. We are only calculating targets.

(1) First, we get the values of the Q-function at s prime (next state). The "s" in 'next_states' means that this is a batch of 'next_state'.

```
max_a_q_sp = q_sp.max(1)[0].unsqueeze(1)
```

(4) The 'unsqueeze' just adds a dimension to the vector so the operations that follow work on the correct elements.

(3) Then, we get the max value of the next state 'max_a'.

(5) One important step, often overlooked, is to ensure terminal states are grounded to zero.

(6) Also, notice the name "is_terminals" are batches of "is_terminal" flags, which are merely flags indicating whether the "next_state" is a terminal state or not.

```
max_a_q_sp = * (1 - is_terminals)
```

```
target_q_s = rewards + self.gamma * max_a_q_sp
```

(7) We now calculate the target.

```
q_sa = self.online_model(states).gather(1, actions)
```

(8) Finally, we get the current estimate of $Q(s,a)$. At this point, we are ready to create our loss function.

I want to bring to your attention two issues that I, unfortunately, see very often in DRL implementations of algorithms that use TD targets.

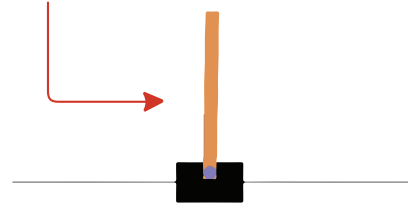
First, you need to make sure that you only back-propagate through the "predicted" values. Let me explain. You know that in supervised learning, you have predicted values, which come from the learning model, and true values, which are commonly constants provided in advance. In RL, often the "true values" depend on predicted values themselves, they come from the model.

For instance, when you form a TD target, you use a reward, which is a constant, and the discounted value of the next state, which comes from the model. Notice, this value is also not a true value, which is going to cause all sorts of problems that we address in the next chapter. But what I also want you to notice now, is that the predicted value comes from the neural network. You have to make this predicted value a constant. In PyTorch, you do this only by calling the 'detach' method. Please, look at the two previous boxes and understand these points. They are vital for the reliable implementation of DRL algorithms.

The second issue that I want to raise before we move on is the way terminal states are handled when using OpenAI Gym environments. The OpenAI Gym `'step'`, which is used to interact with the environment, returns after every step a handy flag indicating whether the agent just landed on a terminal state. This flag helps make the value of terminal states zero, which, as you remember from chapter 2, is a requirement to keep the value functions from diverging. You know the value of life after death is nil.

The tricky part is that some OpenAI Gym environments, such as the cart-pole, have a wrapper code that artificially terminates an episode after some time steps. In `CartPole-v0`, the time step limit is 200, and in `CartPole-v1` is 500. Now, this wrapper code helps to prevent agents from taking too long to complete an episode, which can be useful, but it can get you in trouble. Think about it, what do you think the value of having the pole straight up in time step 500 be? I mean, if the pole is straight up, and you get +1 for every step, then straight-up is infinite. Yet since your agent landed in a terminal time, and you got a done flag, will you bootstrap on zero, then? This is bad. I cannot stress this enough. There are a handful of ways you can handle this issue. You can either (1) use the `'unwrapped'` property of the `'env'` instance to get an environment that doesn't time out, you can (2) keep a time step count and bootstrap when you reach it, or you can (3) check the return value of the `'_past_limit'` function of the `'env'` instance and bootstrap on failure. (2) is the most common, but I'll use (3).

① Can you guess what the value of this state is?



② HINT: This state looks pretty good to me! The cart pole seems to be "under control" in a straight-up position. Perhaps the best action is to push right, but it doesn't seem like a critical state. Both actions are probably similarly valued.



I SPEAK PYTHON

Properly handling terminal states

```
new_state, reward, is_terminal, _ = env.step(action)
past_limit = hasattr(env, '_past_limit') and env._past_limit()
```

① We collect a experience tuple as usual

② Then check if the `'_past_limit'` function exists and it returns True.

```
is_failure = is_terminal and not past_limit
```

③ A failure is defined as follows.

```
experience = (state, action, reward, new_state, float(is_failure))
```

④ Finally, we add the "terminal" flag if the episode ended in failure. If it is not a failure we want to bootstrap on the value of the "new_state".

Fifth decision point: Selecting an exploration strategy

Another thing we need to decide is on which policy improvement step to use for our generalized policy iteration needs. You know this from chapters 6 and 7 in which we interleave a policy evaluation method, such as MC or TD, and a policy improvement method that accounts for exploration, such as decaying ϵ -greedy.

In chapter 4, we surveyed many different ways to balance the exploration-exploitation tradeoff, and almost any of those techniques would work just fine. But in an attempt to keep it simple, we are going to use an epsilon-greedy strategy on our NFQ implementation.

But, I want to highlight the implication of the fact that we are training an off-policy learning algorithm here. What that means is that there are two policies: a policy that generates behavior, which in this case is an ϵ -greedy policy, and a policy that we are learning about, which is the greedy (an ultimately optimal) policy.

One interesting fact of off-policy learning algorithms you studied in chapter 6 is that the policy generating behavior can be virtually anything. That is, it can be anything as long as it has broad support, which means it must ensure enough exploration of all state-action pairs. In our NFQ implementation, I use an epsilon-greedy strategy that selects an action randomly 50% of the time during training. However, when evaluating the agent, I use the action greedy with respect to the learned action-value function.



I SPEAK PYTHON

Epsilon-greedy exploration strategy

```
class EGreedyStrategy():
    <...>
    def select_action(self, model, state):
        with torch.no_grad():
            q_values = model(state).cpu().detach()
            q_values = q_values.data.numpy().squeeze()
            if np.random.rand() > self.epsilon:
                action = np.argmax(q_values)
            else:
                action = np.random.randint(len(q_values))
        <...>
        return action
```

(1) The 'select_action' function of the 'EGreedy Strategy' starts by pulling out the q-values for state s.

(2) I make the values "numpy friendly" and remove an extra dimension.

(3) Then, get a random number and if greater than epsilon act greedily.

(4) Otherwise, act randomly in the number of actions.

(5) NOTE: I always query the model in order to calculate stats. But, you should not do that if your goal is performance!

Sixth decision point: Selecting a loss function

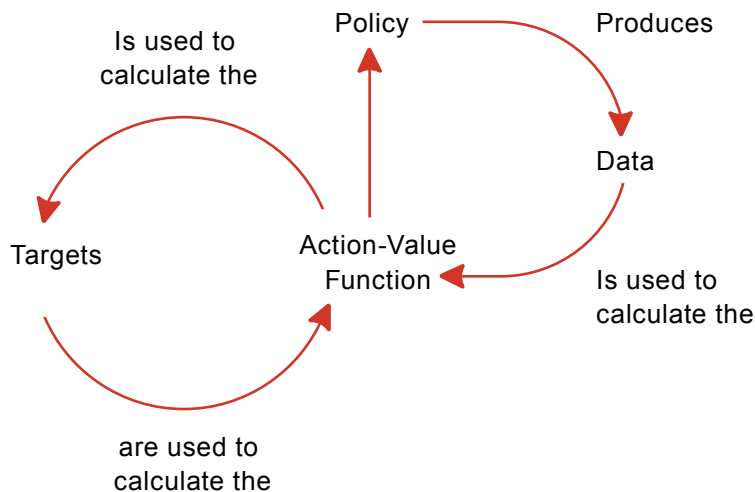
A loss function is a measure of how well our neural network predictions are. In supervised learning, it is more straightforward to interpret the loss function: given a batch of predictions and their corresponding true values, the loss function computes a distance score indicating how well the network has done in this batch.

There are many different ways for calculating this distance score, but I continue to keep it simple in this chapter and use one of the most common ones: MSE (mean squared error, or L2 loss).

Still, let me restate that one challenge in reinforcement learning, as compared to supervised learning, is that our "true values" use predictions that come from the network.

MSE (or L2 loss) is defined as the average squared difference between the predicted and true values; in our case, the "predicted values" are the predicted values of the action-value function that come straight from the neural network, all good. But the "true values" are, yes, the TD targets, which depend on a prediction also coming from the network, the value of the next state.

Circular dependency of the action-value function



As you may be thinking, this circular dependency is bad. It is not well-behaved as it doesn't respect some of the assumptions made in supervised learning problems. We'll cover what these assumptions are later in this chapter and the problems that arise when we violate them in the next chapter.

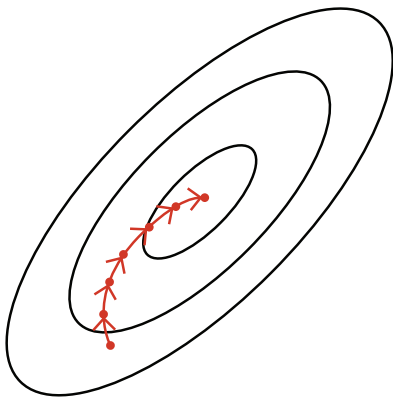
Seventh decision point: Selecting an optimization method

Gradient descent is a stable optimization method given a couple of assumptions: one, referred to as the IID assumption, which stands for Independent and Identically Distributed, and another that targets are stationary. In reinforcement learning, however, we cannot ensure any of these assumptions hold, so choosing a robust optimization method to minimize the loss function can often make the difference between convergence and divergence.

If you visualize a loss function as a landscape with valleys, peaks, and planes, an optimization method is the hiking strategy for finding areas of interest, usually the lowest or highest point in that landscape.

A classic optimization method in supervised learning is called batch gradient descent. The batch gradient descent algorithm takes the entire dataset at once, calculates the gradient of given the dataset, and steps towards this gradient a little bit at a time. Then, it repeats this cycle until convergence. In the landscape analogy, this gradient represents a signal telling us the direction we need to move. Batch gradient descent is not the first choice of researchers because it is not practical to process massive datasets at once. When you have a considerable dataset with millions of samples, batch gradient descent is too slow to be practical. Moreover, in reinforcement learning, we don't even have a dataset in advance, so batch gradient descent is not a practical method for our purpose either.

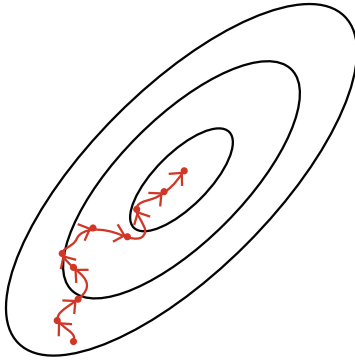
Batch gradient descent



- ① Batch gradient descent goes smoothly towards the target because it uses the entire dataset at once, so lower variance is expected.

An optimization method capable of handling smaller batches of data is called mini-batch gradient descent. In mini-batch gradient descent, we use only a fraction of the data at a time. We process a mini-batch of samples to find its loss, then back-propagate to compute the gradient of this loss, and then adjust the weights of the network to make the network better at predicting the values of that mini-batch. With mini-batch gradient descent, you can control size of the mini-batches, which allows the processing of large datasets.

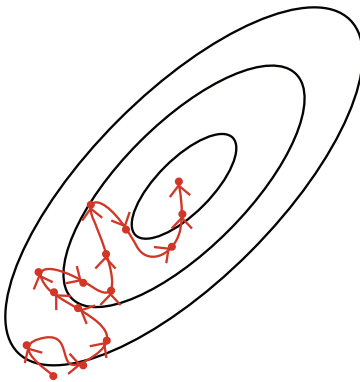
Mini-batch gradient descent



- ① In mini-batch gradient descent we use a uniformly sampled mini batch. This results in noisier updates, but also faster processing of the data.

As one extreme, you can set the size of your mini-batch to the size of your dataset, in which case, you are back at batch gradient descent. On the other hand, you can set the mini-batch size to a single sample per step; in this case, you are using an algorithm called stochastic gradient descent.

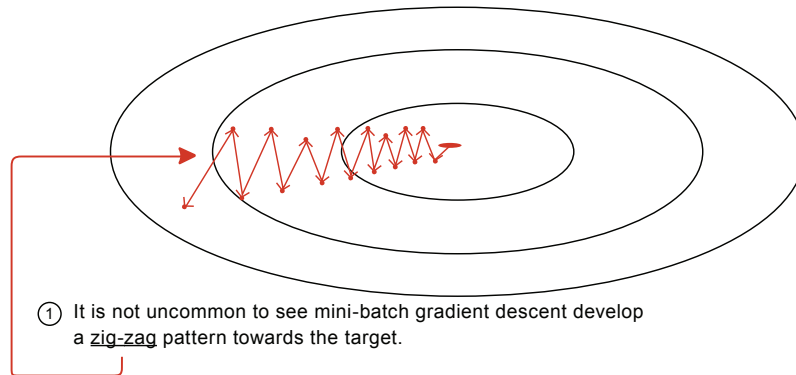
Stochastic gradient descent



- ① With stochastic gradient descent every iteration we step only through one sample. This makes it a very noisy algorithm. It wouldn't be surprising to see some steps taking us further away from the target, and later back towards the target.

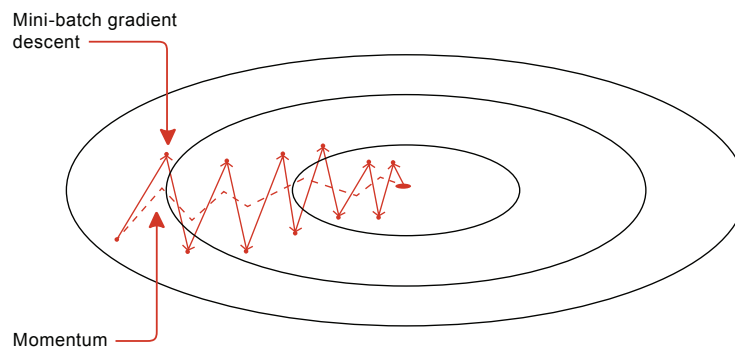
The larger the batch, the lower the variance the steps of the optimization method have. But a batch too large, and learning slows down considerably. Both extremes are too slow in practice. For these reasons, it is common to see mini-batch sizes ranging from 32 to 1024.

Zig-zag pattern of mini-batch gradient descent



An improved gradient descent algorithm is called gradient descent with momentum, or just momentum for short. This method is a mini-batch gradient descent algorithm that updates the network's weights in the direction of the moving average of the gradients, instead of the gradient itself.

Mini-batch gradient Descent vs Momentum



An alternative to using momentum is called root mean square propagation (RMSprop). Both RMSprop and momentum do the same thing of dampening the oscillations and moving more directly towards the goal, but they do so in different ways.

While momentum takes steps in the direction of the moving average of the gradients, RMSprop takes the safer bet of scaling the gradient in proportion to a moving average of the magnitude of gradients. It reduces oscillations by merely scaling the gradient in proportion to the square root of the moving average of the square of the gradients or, more simply put, in proportion to the average magnitude of recent gradients.



MIGUEL'S ANALOGY

Optimization methods in value-based deep reinforcement learning

To visualize RMSprop, think of the steepness change of the surface of your loss function. If gradients are high, such as when going downhill, and the surface changes to a flat valley, where gradients are small, the moving average magnitude of gradients is higher than the most recent gradient, therefore, the size of the step is reduced, preventing oscillations or overshooting.

If gradients are small, such as in a near-flat surface, and they change to a significant gradient, as when going downhill, the average magnitude of gradients is small, and the new gradient large, therefore increasing the step size and speeding up learning.

A final optimization method I'd like to introduce is called adaptive moment estimation (Adam). Adam is a combination of RMSprop and momentum. The Adam method steps in the direction of the velocity of the gradients, as in momentum. But, it scales updates in proportion to the moving average of the magnitude of the gradients, as in RMSprop. These properties make Adam an optimization method a bit more aggressive than RMSprop, yet not as aggressive as momentum.

In practice, both Adam and RMSprop are sensible choices for value-based deep reinforcement learning methods. I make extensive use of both in the chapters ahead. However, I do prefer RMSprop for value-based methods, as you'll soon notice. RMSprop is stable and less sensitive to hyperparameters, and this is particularly important in value-based deep reinforcement learning.

0001 A BIT OF HISTORY

Introduction of the NFQ Algorithm

NFQ was introduced in 2005 by Martin Reidmiller on a paper called "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method". After 13 years working as a Professor on a number of European Universities, Martin took a job as a Research Scientist at Google DeepMind.



IT'S IN THE DETAILS

The full Neural Fitted Q-Iteration (NFQ) algorithm

Currently, we have made the following selections, we:

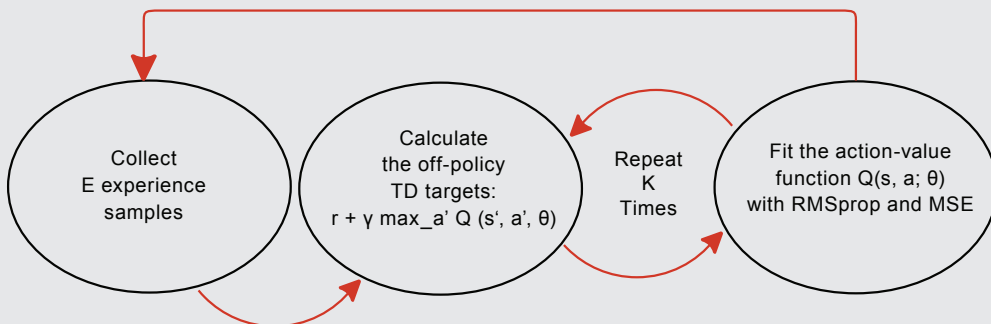
- Approximate the action-value function $Q(s, a; \theta)$.
- Use a state-in-values-out architecture (nodes: 4, 512, 128, 2).
- Optimize the action-value function to approximate the optimal action-value function $q^*(s, a)$.
- Use off-policy TD targets $(r + \gamma \max_{a'} Q(s', a'; \theta))$ to evaluate policies.
- Use an epsilon-greedy strategy (epsilon set to 0.5) to improve policies.
- Use mean squared error (MSE) for our loss function.
- Use RMSprop as our optimizer with a learning rate of 0.0005.

NFQ has three main steps:

1. Collect E experiences: (s, a, r, s', d) tuples. We use 1024 samples.
2. Calculate the off-policy TD targets: $r + \gamma \max_{a'} Q(s', a'; \theta)$.
3. Fit the action-value function $Q(s, a; \theta)$: Using MSE and RMSprop.

Now, this algorithm repeats steps 2 and 3 K number of times before going back to step 1. That's what makes it "fitted"; the nested loop. We'll use 40 fitting steps K .

NFQ

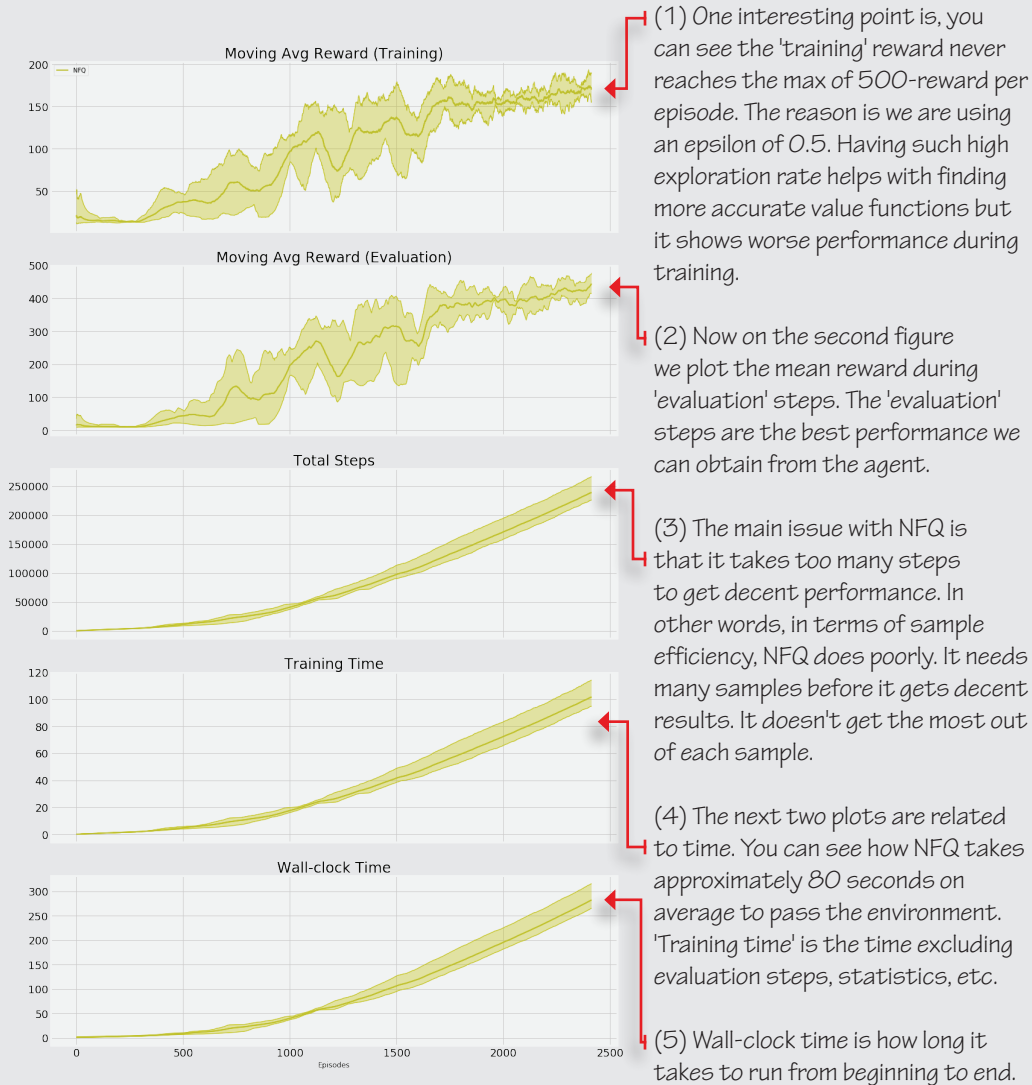




TALLY IT UP

NFQ passes the cart-pole environment

Although NFQ is far from a state-of-the-art value-based deep reinforcement learning method, in a somewhat simple environment, such as the cart pole, NFQ shows a decent performance.



Things that could (and do) go wrong

There are two issues with our algorithm. First, because we are using a powerful function approximator, we can generalize across state-action pairs, which is excellent, but that also means that the neural network adjusts the values of all similar states at once.

Now, think about this for a second, our target values depend on the values for the next state, which we can safely assume are similar to the states we are adjusting the values of in the first place.

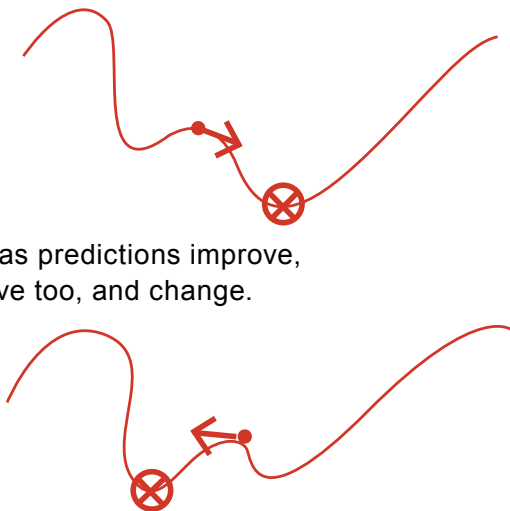
In other words, we are creating a non-stationary target for our learning updates. As we update the weights of the approximate Q-function, the targets also move and make our most recent update outdated. Thus, training becomes unstable very quickly.

Non-stationary target

- ① At first our optimization will behave as expected going after the target.

- ② The problem is that as predictions improve, our target will improve too, and change.

- ③ Now, our optimization method can get in trouble.

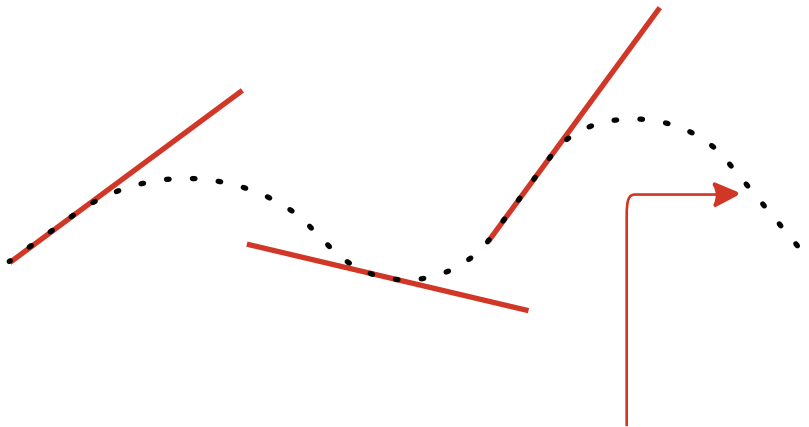


Second, in NFQ, we batched 1024 experience samples collected online, and update the network from that mini-batch. As you can imagine, these samples are correlated, given that most of these samples come from the same trajectory and policy. That means the network learns from mini-batches of samples that are very similar, and later using different mini-batches that are also internally correlated, but likely different from previous mini-batches, mainly if a different, older policy collected the samples.

All this means that we are not holding the IID assumption, and this is a problem because optimization methods assume the data samples they use for training are independent and identically distributed (IID). But we are training on almost the exact opposite: samples on our distribution are not independent because the outcome of a new state s' is dependent on our current state s .

And, also, our samples are not identically distributed because the underlying data generating process, which is our policy, is changing over time. That means we do not have a fixed data distribution. Instead, our policy, which is responsible for generating the data, is changing and hopefully improving periodically. So, every time our policy changes, we receive new and likely different experiences. Optimization methods allow us to relax the IID assumption to a certain degree, but reinforcement learning problems go all the way, so we need to do something about this, too.

Data correlated with time



- ① Imagine we generate these data points in a single trajectory. Say the y axis is the position of the cart along the track, and the x axis is the step of the trajectory. You can see how likely it is data points at adjacent time steps will be similar making our function approximator likely to overfit to that local region.

In the next chapter, we look at ways of mitigating these two issues. We start by improving NFQ with the algorithm that arguably started the deep reinforcement learning 'revolution,' DQN. We then follow by exploring many of the several improvements proposed to the original DQN algorithm over the years. We look at Double DQN also in the next chapter, and then in chapter 10, we look at Dueling DQN and PER.

Summary

In this chapter, you learned about value-based deep reinforcement learning methods. You had an in-depth overview of different components commonly used when building deep reinforcement learning agents. You learned you could approximate different kinds of value functions, from the state-value function $v(s)$ to the action-value $q(s, a)$. Also, you learned different neural network architectures to approximate action-value functions; from the state-action pair in, value out, to the more efficient state-in, values out.

You know there are many different targets you can use to train your network. You surveyed exploration strategies, loss functions, and optimization methods. You learned that deep reinforcement learning agents are susceptible to the loss and optimization methods we select. You learned about RMSprop and Adam as the stable options for optimization methods.

You learned to combine all of these components into an algorithm called Neural Fitted Q-iteration. You learned about the issues commonly occurring in value-based deep reinforcement learning methods. You learned about the IID assumption and the stationarity of the targets. You also learned that not being careful with these two issues can get us in trouble.

By now you:

- Can solve reinforcement learning problems with continuous state-spaces.
- Have an in-depth understanding of the components and issues in value-based deep reinforcement learning methods.