

# Deep Reinforcement Learning IN ACTION

Alex Zai  
Brandon Brown



MANNING





**MEAP Edition  
Manning Early Access Program  
Deep Reinforcement Learning in Action  
Version 5**

Copyright 2019 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](http://manning.com)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/deep-reinforcement-learning-in-action>  
**Licensed to Ivan Hom <ihom627@yahoo.com>**

# welcome

---

Firstly, we want to thank you for giving this book a chance. With so many options available in the machine learning space for tutorials and training, you may be wondering why we chose to join the fray. Well, there are, in fact, a lot of resources out there for learning the basics of machine learning and deep learning (to shamelessly plug another Manning book, we recommend Andrew Trask's *Grokking Deep Learning* to get up to speed there). But once people get a handle on the basics, where do they go from there?

We think a great next step for the newly-minted deep learning aficionado is to apply their new skills to the field of reinforcement learning. Reinforcement learning has seen tremendous success in the past few years and exciting results are happening every day, yet the landscape of truly beginner-level material in this area is comparatively sparse. Unlike deep learning, which has already infiltrated almost every major technology, reinforcement learning has just recently started to take form as a viable solution to practical problems. This means if you learn reinforcement learning now, you'll be walking in on the ground floor of something that is surely going to surge in value in the near future.

While reinforcement learning is a distinct field from deep learning, the marriage of the two into deep reinforcement learning is a natural and powerful combination and is the dominant form of machine learning being developed. This book aims to teach you to use core deep reinforcement learning skills to solve real-world problems in the most approachable and intuitive manner possible.

If you have any questions, comments, or suggestions, please share them in Manning's [Author Online forum](#) for our book. We hope you'll enjoy the ride!

— Alex Zai and Brandon Brown

# *brief contents*

---

## **PART 1: FOUNDATIONS**

- 1 What is reinforcement learning*
- 2 Modeling Reinforcement Learning Problems: Markov Decision Processes*
- 3 Predicting the Best States and Actions: Deep Q-Networks*
- 4 Learning to Pick the Best Policy: Policy Gradient Methods*
- 5 Tackling more complex environments with Actor-Critic methods*

## **PART 2: ABOVE AND BEYOND**

- 6 Alternative Optimization Methods: Evolutionary Strategies*
- 7 Understanding the environment and making plans: Model-Based Learning*
- 8 Curiosity-driven exploration*
- 9 Playing at your level: Self-Play*
- 10 Conclusion*

## **APPENDIXES**

- A Mathematics*
- B Deep Learning*
- C PyTorch*
- D Glossary*

# 1

## *What is Reinforcement Learning?*

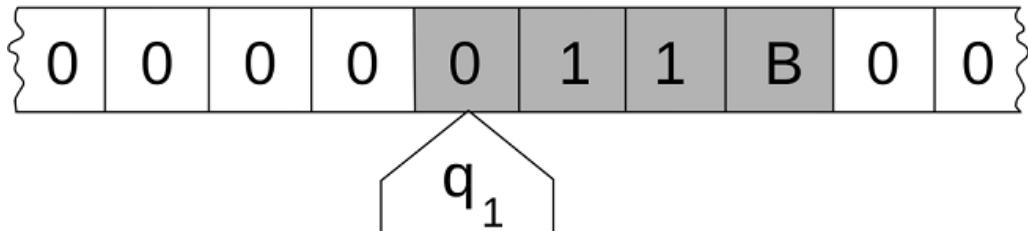
“Computer languages of the future will be more concerned with goals and less with procedures specified by the programmer.”

– Marvin Minsky, 1970

---

### 1.1 The Journey Here

In 1936 the English mathematician Alan Turing published a paper entitled “*On Computable Numbers, with an Application to the Entscheidungsproblem*,” in which he developed a mathematical description of an algorithm, which later became known as a Turing machine. His conception of the Turing machine became the basis of the development of modern electronic computers. Interestingly, he originally developed the Turing machine as a tool to solve a mathematical problem, not because he wanted to invent a new field of computer science. Nevertheless, he and many others recognized the power of this new idea of computation and it immediately begged the question of whether these new computing systems could become intelligent like humans. Alan Turing himself kicked off the field of artificial intelligence by subsequently publishing the paper “Computing Machinery and Intelligence” in 1950 where he began with the question, “Can machines think?” The board game Chess has always been considered an intellectual sport, and thus it was natural to use it as a testbed for whether machines could be made intelligent. Several attempts at writing Chess-playing algorithms were made over the subsequent years, starting with algorithms that were only described on paper and had to be run by a human following a sequence of rules, up to the landmark defeat of the Chess world champion Gary Kasparov in 1996 by IBM’s DeepBlue algorithm.



**Figure 1.1:** This depicts how a single-tape Turing machine works. The shape labeled “q1” is a read and write head that can move left or right across a tape that has printed values in evenly spaced cells. The read/write head follows a predefined set of rules and can read certain values on the tape, erase, and write new values to the tape. This simple model captures the essence of computation and started the field of computer science.

While these Chess playing machines were impressive at the time, they were (and continue to be) mostly based on exhaustive search and pre-programmed logic rather than performing any sort of human-like analysis or strategizing. Indeed, most of what was called artificial intelligence in the early days was just hard-coded logic and heuristics. The first descriptions of artificial neural networks, algorithms that attempted to model biological neural networks, appeared in the 1940s and would become one of many algorithms that fall under the umbrella of machine learning. One particularly influential theory of how neural networks learn was advanced by psychologist Donald Hebb in his 1949 book “Organization of Behavior.” Hebb proposed that learning happens at the level of neurons when two neurons repeatedly fire an electrochemical signal called an action potential in synchrony and in close proximity, then the causal relationship between these two neurons will become enhanced, or reinforced. His theory later became known as Hebbian learning and is still considered relevant with significant empirical evidence backing it.

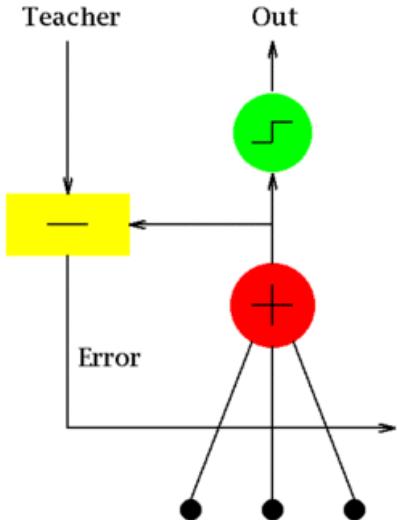


Figure 1.2: A diagram of one of the first neural networks called ADALINE that was used to filter out echoes in telephone lines. The network simply computed a weighted sum of its input variables and if the sum was greater than some threshold parameter, it would return 1, otherwise it returns 0 (called a step function). By defining some error function, you could follow simple rules to modify the threshold parameter in order to reduce the error and improve the performance of the algorithm.

With some theoretical neuroscience to take advantage of, the first artificial neural networks were implemented on some of the earliest computing machines in the 1950s. Remarkably, in 1959, researchers at Stanford created the first commercial application of a neural network, an algorithm called ADALINE, which was designed to filter out echoes in telephone line signals and is still in use today. Any neural network algorithm, and indeed most machine learning algorithms, involve parameters that control their behavior. In order for the neural network to perform whatever task is desired, the network's parameters must be "trained" to be set to the right values. ADALINE and other early neural networks were trained by simple heuristics and searching procedures, although some did employ what would later be rigorously worked out and defined as **backpropagation**. The history of the backpropagation technique is a little uncertain, but it was not widely adopted until the mid 1980s where it was able to train more sophisticated neural networks than ADALINE and remains the standard training procedure for neural networks.

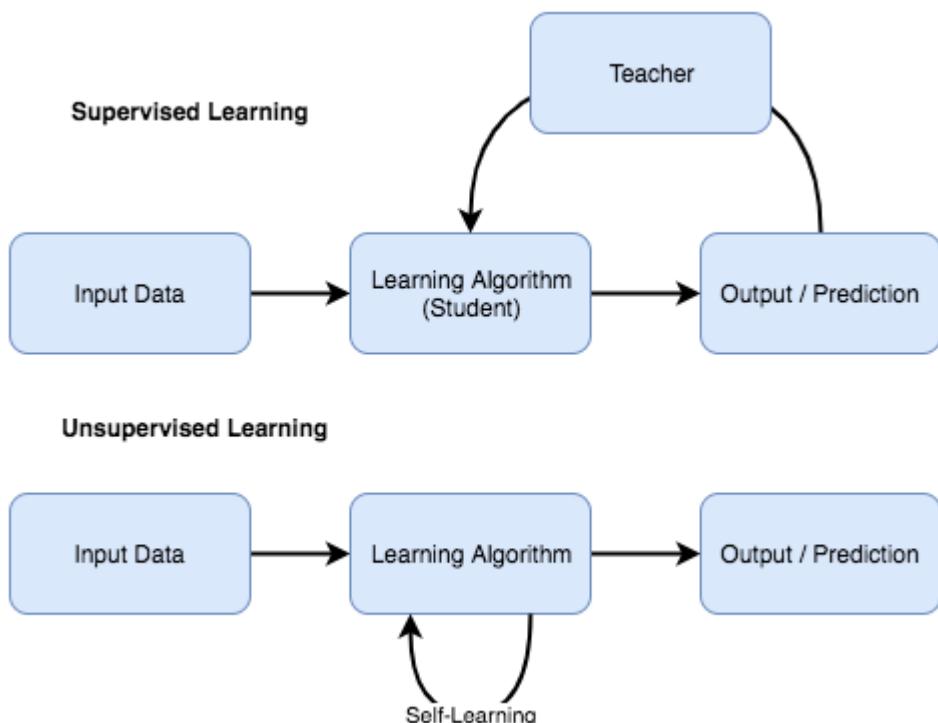
## 1.2 Supervised and Unsupervised Learning

ADALINE, like most of the neural networks and other types of machine learning algorithms that followed it, was a **supervised learning** algorithm. Supervised learning algorithms are trained in a teacher-student manner. For example, if you want to build an algorithm to classify

a dataset of images into certain categories, then a teacher would supply the algorithm with images to classify, and the algorithm would make a prediction, and then based on the accuracy of the prediction, the teacher would provide corrective feedback such that the algorithm would perform a little better next time. Doing this over many examples of images, the algorithm would eventually perform well enough. More accurately, a learning algorithm is given an input datum and produces an output. The output is then evaluated by some objective function that compares the algorithm's output to the known correct or labeled outputs, and produces a measure of how wrong the algorithm's output was (i.e. an error). Then the parameters of the algorithm are updated to minimize the error produced by the objective function, often using the backpropagation technique. Hence, you need two sets of data for every supervised learning task: the data to be learned and the labeled data (i.e. the correct answers to whatever the problem is). The labeled data must be produced and curated by humans specifically for the purpose of training an algorithm, whereas the data to be learned may be acquired from the "environment", e.g. telephone line signals. Most large-scale commercial applications require at least tens of thousands and often millions and billions of rows of labeled training data to achieve satisfactory accuracy in their task.

In contrast, **unsupervised learning** algorithms attempt to learn something from data without being given explicit feedback by a teacher. The most common form of unsupervised learning would be clustering, in which an algorithm attempts to find clusters of related data points in some data set. More modern and sophisticated unsupervised learning techniques include a kind of neural network called an auto-encoder, which can automatically learn fairly complex patterns in data. Nonetheless, unsupervised methods are limited in utility since it's difficult to operationalize them. A fancy unsupervised algorithm might learn some interesting patterns in your data, but unless you tell it what you want it to do (in effect converting it to supervised learning), then it won't be able to say, classify your images into the categories you care about.

Supervised learning has been the dominant form of machine learning and the most successful commercially, however, it is unsatisfactory in our quest for the intelligent machines that Alan Turing dreamed of. Surely one major reason for wanting to develop intelligent machines is so that they can do things we don't already know how to do. This is the major limitation of supervised learning; there must already exist a teacher that knows how to do the task and can teach the machine. If the machine learns how to do the task, then it is of course very useful since it may be able to do it faster, cheaper and with less error than a human, but it still cannot learn how to do anything we don't already know how to do. These supervised learning algorithms are necessarily extremely domain specific.



**Figure 1.3: Supervised learning is the class of algorithms that exhibit a teacher-student relationship where the algorithm learns by being given explicit feedback by a teacher. Unsupervised learning algorithms are able to self-teach to some extent, and learn something about data on their own.**

Clearly, humans have aspects of both supervised and unsupervised learning. Learning a new subject in school for example, is largely an instance of a teacher-student situation. However, it's unsupervised learning that is responsible for human progress: the creation of new knowledge, new skills, art and creativity. There has to be the first one to learn something on their own before being able to teach it. Supervised learning can help us be more efficient by saving us time and resources, thereby giving us more resources to invest in creation, but it is not going to give us artificial general intelligence (AGI) where machines would become our intellectual peers and not just automation tools. But even if you're entirely practically-minded and have no interest in AGI, supervised learning is still unsatisfactory because it is limited by how much training data we can give it; just like the number of students that can go to school is limited by the number of teachers. Sometimes we just don't have the resources to produce the labeled data set, or doing so outweighs the benefit of using the algorithm.

For example, one of the authors' wife is a neuroscientist who uses fancy microscopes to image neurons in living rodents. The microscopes produce gigabytes of video data that must be painstakingly analyzed by manually identifying and delineating neurons to produce

quantitative data about the neurons' characteristics. A machine learning paper was recently published in which the paper's authors developed a neural network that could automatically identify neurons in these videos. However, it was a supervised neural network that required each user to provide a labeled data set of thousands of manually identified neurons. The authors themselves noted that this process may take more than 10 hours (optimistically). While such a significant time investment might pay off in the long term if you have to do this type of experiment many times, no busy scientist wants to risk spending an entire working day building a training data set for an algorithm that is not guaranteed to work for them.

Perhaps even more illustrative of the inadequacy of supervised learning is teaching an algorithm how to drive a car or operate a robot. Let's say you want a car to learn how to drive just by analyzing video feeds (i.e. no other sensors like radar and lidar); this is in fact the strategy that Tesla is currently using to develop its autopilot software. The car must continuously control the steering wheel and continuously adjust the accelerator or breaks. How are you going to collect training data for this task? This is a **control task** (or decision task) not a prediction or classification task which are the bread-and-butter of supervised learning.

In a traditional supervised learning task, the environment is precisely managed and only the data we curated is accessible to the algorithm, and each piece of data is fully independent (i.e. we could arbitrarily remove or add data without significantly affecting the overall performance of the algorithm). In a control task, the environment is largely unmanaged and produces data "on its own" and may be probabilistic. In our self-driving car example, the data the algorithm is receiving from its cameras is not produced by some human curator, it's produced by the natural evolution of the environment around it as a function of time, which is largely unpredictable. In addition, without a bunch of expensive sensors, the car can only make incomplete observations of the surrounding environment. It would be totally unfeasible to teach the algorithm what it should be doing at every instant of time, such as the exact amount the steering wheel should be rotated given what the cameras are currently recording.

### 1.3 Problem Structuring in Control Tasks

This takes us back to the quote at the beginning of this chapter by the pioneering artificial intelligence researcher Marvin Minsky. Many early artificial intelligence algorithms were nothing more than hard-coded rules that could be easily followed by a human. The advent of neural networks and other learning algorithms allowed us to teach an algorithm how to do something just by giving it the correct answers, telling it how wrong it was and let it update itself to be less wrong next time. This freed us from having to work out and program a fixed set of rules to solve a problem and gave us greater flexibility. But it's still a very procedural process: feed in some data, get an output, produce an error value based on our knowledge of the correct answer, then update the algorithm using some technique like backpropagation. What if instead we could just give a high-level objective or goal to the algorithm and let it

figure out the details? We want to be able to do goal-directed programming like Marvin Minsky suggested would be the future.

The field of **reinforcement learning** is starting to realize that imagined future. In reinforcement learning, we don't need to give the algorithm a bunch of labeled data to learn some task, we just need to be able to define what success looks like in the environment. For example, we might give the high-level goal of "don't hit anything, follow all traffic rules, and navigate from point A to point B along the most efficient route" to our self-driving car algorithm and let it figure out how to work out the details of achieving that goal. This may seem like a remarkable feat, but it's exactly the kind of thing we'll be learning how to do in this book.

All human behavior is fundamentally goal-directed, so it makes sense that this is how we should design our learning algorithms. Biological evolution by natural selection has endowed most species with a set of primitive goals or drives, such as minimizing hunger, avoiding pain, seeking pleasure, reducing uncertainty, and in the case of humans and other social animals, achieving social acceptance. Even in the modern world that is far abstracted away from our ancestral environment, every human behavior can be reduced to some combination of these primitive drives and goals. For example, as awkward as it is to admit, the authors writing this book may think that they are doing it because they are excited about reinforcement learning and want to share it with others, but fundamentally we are motivated by our evolutionary primitive drive to be contributing members of our tribe. It's amazing how much has been accomplished in the world by a bunch of people individually maximizing or minimizing a fairly small set of high-level drives and goals.

Let's run with this idea. If a person is hungry (high-level drive) then she must eat to minimize her hunger. But eating can't be done in a single step; she first must make very low level decisions like which direction to walk in. In this sense, in order to solve a high-level objective we can break it down into small sub-problems that together will contribute to solving the high-level problem.

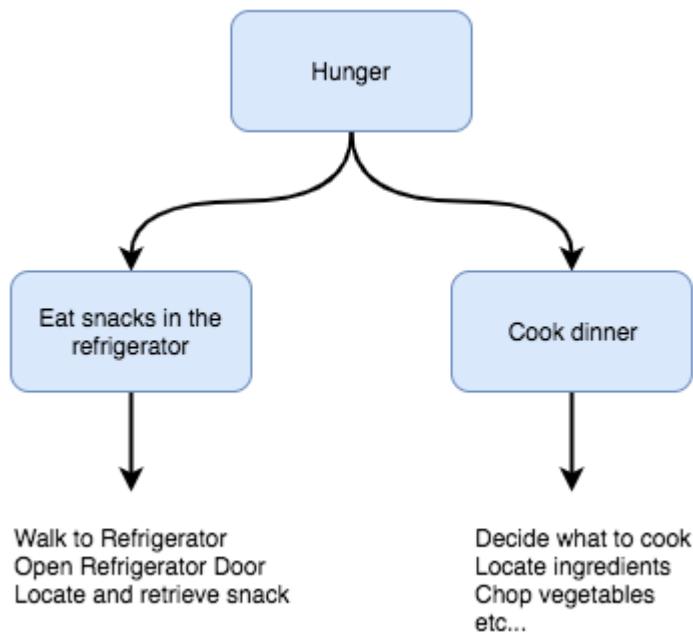


Figure 1.4: A high-level goal like satisfying one's hunger can be broken down into a tree of “local” sub-goals that each must be solved in order to achieve the global goal.

This idea of breaking a high-level goal, in which it is unclear how to solve on its own, into a hierarchy of sub-goals which at the lowest level are all almost trivial decisions was given a rigorous mathematical description by Richard Bellman in 1957. Bellman called this technique **dynamic programming** for reasons that mostly had to do with historical circumstances than for clarity. A more intuitive name might be “goal decomposition.” Bellman’s dynamic programming has one more trick besides goal decomposition; in addition to breaking an objective into a set of easier to solve sub-objectives, we should also store the solutions to all of the solutions of these sub-objectives so that if we need to solve the same sub-objective at a later time point, we can just look up the solution we got last time rather than figuring it out all over again. This technique of storing previously solved sub-problems is called **memoization** (note, it’s not **memorization**, it’s memoization; there’s no “r”).

Dynamic programming (DP) can be used to solve any control task that can be broken down into smaller parts. This has applications from economics to mathematics to computer science and of course to reinforcement learning, as we’ll soon see. But to give a simple example of DP in action (and the example you’ll find most often in textbooks), let’s see how you might use it to improve the efficiency of a computer algorithm that produces the  $n$ -th number in the Fibonacci sequence. If you’re not familiar with the Fibonacci sequence, it’s the sequence of positive whole numbers that follows this pattern: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... where the  $n$ -th

number is the sum of the previous two numbers. In other words,  $n=(n-1)+(n-2)$  where n refers to the n-th number (zero-indexed) in the sequence and the first two numbers are always 0 and 1. Just take our word for it that this sequence of numbers is special and actually has useful applications, hence why we might want to write a computer program to produce it. Using this mathematical definition as our starting point, we might write this function in Python as:

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)

>>> fib(7)
13
```

It works great. The problem is this function is terribly inefficient; look what happens when we call `fib(4)`:

```
fib(4)
fib(3) + fib(2)
(fib(2) + fib(1)) + (fib(1) + fib(0))
```

We get a tree of recursive function calls where some identical computations are performed multiple times, for example, we can see `fib(2)` is computed twice. If we use the dynamic programming approach of memoizing our computations, we could have a much more efficient algorithm at the cost of a relatively mild memory overhead. Here's what the DP approach looks like in Python:

```
mem = {0:0, 1:1}

def fib_mem(n):
    if n not in mem:
        mem[n] = fib(n-1) + fib(n-2)
    return mem[n]

>>> fib_mem(7)
13
```

We defined a Python dictionary to store any computations we make and we initialized it to the first two numbers in the Fibonacci sequence. If the function is called with an input that it has already encountered, it can just use the Python dictionary to look up that previously computed number and return it, otherwise it will compute it (but just once) and store it in the `mem` dictionary.

As we now know, in order to apply Bellman's dynamic programming we have to be able to break our problem into sub-problems that we know how to solve. But even this seemingly innocuous assumption is difficult to realize in the real world. How do you break the high level goal for a self-driving car of "don't crash" into small non-crashing sub-problems? Does a child learn to walk by first solving easier sub-walking problems? In reinforcement learning where we

often have these kinds of nuanced situations that may include some element of randomness, we can't apply dynamic programming exactly as Bellman laid out. In fact DP can be considered one extreme of a continuum of problem solving techniques where the other end would be random trial and error.

Another way to view this learning continuum is that in some situations we have maximal knowledge of the environment and in others we have minimal knowledge of the environment, hence we need to employ different strategies in each case. If you need to use the bathroom in your own house, then you know exactly (well, unconsciously at least) what sequence of muscle movements will get you to the bathroom from any starting position (i.e. dynamic programming-ish). This is of course because you know your house extremely well, or in other words, you have a more or less perfect *model* of your house in your mind. If you go to a party at someone else's house that you've never been to before, then assuming no one tells you, you would just have to look around until you find the bathroom on your own (i.e. trial and error); since you don't have a good model of that person's house.

The trial and error strategy generally falls under the umbrella of Monte Carlo methods. A Monte Carlo method is essentially random sampling from the environment. In many real world problems, we have at least some knowledge of how the environment works, so we end up having to employ a mixed strategy of some amount of random trial and error and some amount of exploiting what we already know about the environment and directly solve the easy sub-objectives.

A silly example of a mixed strategy would be if you were blindfolded, placed in an unknown location in your house and told to find the bathroom by throwing pebbles and listening for the noise. You might start by decomposing the high level goal (find the bathroom) into a more accessible sub-goal, figure out which room you're currently in. To solve this sub-goal, you might throw a few pebbles in random directions and assess the size of the room, which might give you enough information to infer which room you're in, say the bedroom. Then you need to pivot to another sub-goal, navigate to the door so you can enter the hallway. You'd then start throwing pebbles again, but since you memoized (remember) the results of your last random pebble throwing, you could target your throwing to areas of less certainty. Iterating over this process, you might eventually find your bathroom. Hence, in this case, you're applying both the problem structuring (i.e. goal decomposition) of dynamic programming and the random sampling of Monte Carlo methods.

## 1.4 The Standard Model

We learned how Richard Bellman introduced dynamic programming as a general method of solving certain kinds of control or decision problems, but we also know that it occupies an extreme end of the reinforcement learning continuum. Arguably, Bellman's more important contribution was helping develop what we might call the "standard model" for reinforcement learning-type problems. The standard model is essentially the core set of terms and concepts that every reinforcement learning problem can be phrased in. This not only provides a

standardized language in which to communicate to other engineers and researchers, it also forces us to formulate our problems in a way that is amenable to dynamic programming-like problem decomposition, such that we can iteratively optimize over local sub-problems and yet still make progress toward achieving the global high-level objective. Fortunately, it's actually pretty simple too.

To concretely illustrate the standard model, let's consider the task of building a reinforcement learning algorithm that can learn to minimize the energy usage at a big data center. Computers need to be kept cool to function well, so large data centers can incur significant costs from cooling systems. The naïve approach to keeping a data center cool would be just to keep the air conditioning on all the time at some level that results in no servers ever running too hot. You could probably do better than this since it's unlikely that all servers in the center are running hot at the same times and that the data center usage is always at the same level, so if you could target the cooling to where and when it matters most, you could achieve the same result for less money.

Step one of the standard model is to define your overall objective. In this case our overall objective is minimize dollars spent on cooling with the constraint that no server in our center can surpass some threshold temperature. Although this appears to be two objectives, we can bundle these together into a new composite objective function. This will be a function that returns an error value that indicates how off-target we are at meeting the two objectives, given the current cost and the temperature data for the servers. The actual number that our objective function is not important, we just want to make it as low as possible. Hence, we need our reinforcement learning algorithm to minimize this objective (error) function with respect to some input data which will definitely include the running costs and temperature data, but may also include other useful contextual information that can help the algorithm predict the data center usage.

This input data is data that is generated by the environment, a term we've been loosely using so far. In general, the environment of a reinforcement learning (or control) task is whatever dynamic process produces data that is relevant to achieving our objective. Although we use it as a technical term, it's not too far abstracted from its everyday usage. As an instance of a very advanced reinforcement learning algorithm yourself, you are always in some environment and your eyes and ears are constantly consuming information produced by your environment so you can achieve your daily objectives. Since the environment is a dynamic process (i.e. a function of time), it may be producing a continuous stream of data at every instant of time of varied size and type. To make things algorithm-friendly, we need to take this environment-data and bundle it into discrete packets that we call the *state* (of the environment) and can deliver to our algorithm at each of its discrete time steps. The state reflects our knowledge of the environment at some particular time point; just like a digital camera captures a discrete snapshot of a scene at some time (and produces a consistently formatted PNG image or something).

To summarize so far, we defined an objective function (minimize costs and temperature) that is a function of the state (current costs, current temperature data) of the environment

(our data center and any related processes). The last part of our model is the reinforcement learning algorithm itself. This could be *any* algorithm that can learn from data to minimize or maximize some objective function. It does *not* need to be a deep learning algorithm; we want to make clear that reinforcement learning is a field of its own, separate from the concerns of any particular learning algorithm. We'll lay out our reasons for why this book almost exclusively uses deep learning as our learning algorithm in a few sections.

As we noted before, one of the key differences between reinforcement learning (or control tasks generally) and ordinary supervised learning is that in a control task the algorithm needs to make decisions and take actions. These actions will have a causal effect on what happens in the future. Taking an action is indeed another keyword in the standard model and is more or less what you expect it to mean. However, every decision made or action taken is the result of analyzing the current state of the environment and attempting to make the best decision based on that information. The last concept in the standard model is that after each action taken the algorithm is given a *reward*. The reward is a (local) signal of how well the learning algorithm is performing at achieving the global objective. The reward can be a positive signal (i.e. doing well, keep it up) or a negative signal (i.e. this is not working, try something else) even though we call both situations a reward.

The reward signal is the only cue the learning algorithm has to update itself in hopes of performing better at the next state of the environment. In the case of our data center example, we might grant the algorithm a reward of +10 (an arbitrary value) whenever its action reduces the error value. Or more reasonably, we grant a reward proportional to how much it decreases the error. If it increases the error, then we give it a negative reward. Lastly, we give a fancier name to our learning algorithm by calling it the *agent*. The agent is the action-taking or decision-making learning algorithm in any reinforcement learning problem. Finally, we can put this all together as seen in Figure 1.5.

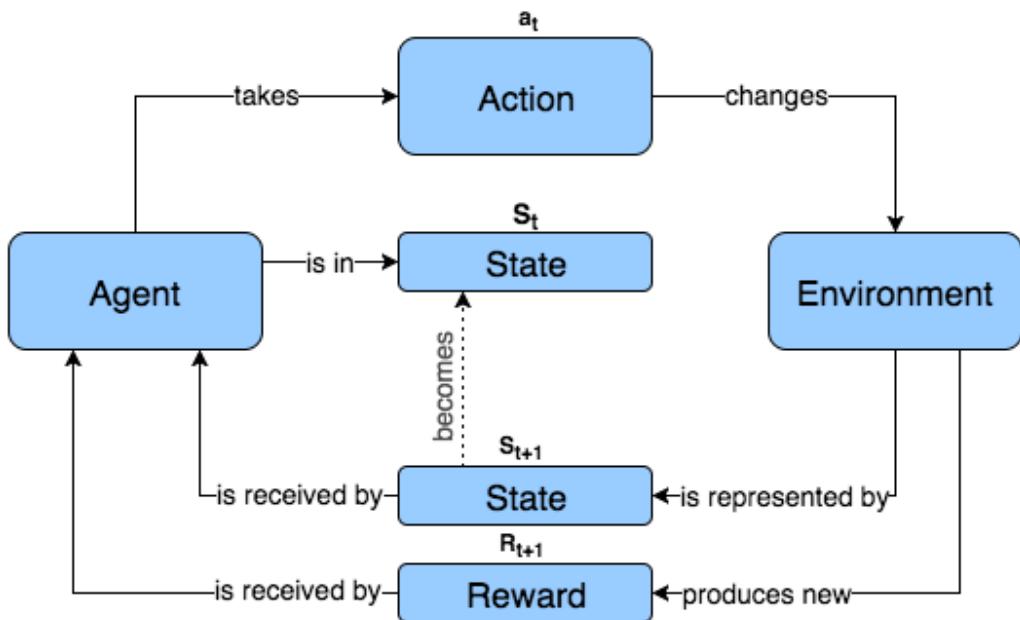
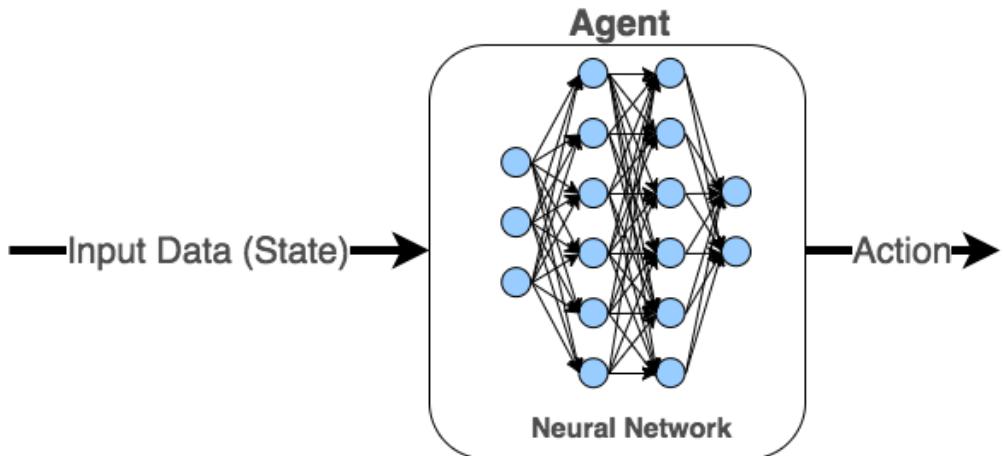


Figure 1.5: The standard model for reinforcement learning algorithms. RL algorithms involve an agent that learns the best way to interact in an environment. The agent takes an action in the environment - such as moving a chess piece - which then updates the state of the environment. For every action we take, we receive a reward (e.g. +1 for winning the game, -1 for losing the game, 0 otherwise). The RL algorithm repeats this process with the objective of maximizing rewards in the long term and eventually learns how the environment works.

In our data center example, we hope that our agent will learn how to decrease our cooling costs. Unless we're able to supply it with complete knowledge of the environment, it will have to employ some degree of trial and error. If we're lucky, the agent might learn so well that it can be used in different environments than the one it was originally trained in. Since the agent is the learner, it is implemented as some sort of learning algorithm. And since this is a book about *deep* reinforcement learning, our agents will be implemented using *deep learning* algorithms (also known as deep neural networks). But remember, reinforcement learning is more about the type of problem and solution than any particular learning algorithm, and one could certainly use alternatives to deep neural networks. In fact, in Chapter 3 we begin using a very simple non-neural network algorithm and then we'll replace it with a neural network by the end of the chapter.



**Figure 1.6:** The input data (which is the state of the environment at some time point) is fed into the agent, which is implemented as a deep neural network in this book, which then evaluates that data to take an action. This process is a little more involved than shown here, but this captures the essence.

The agent's only objective is to maximize its expected rewards in the long term. The agent just repeats this cycle: process the state information, decide what action to take, see if you get a reward, observe the new state, take another action... and so on. And if we set all this up correctly, the agent will eventually learn to understand its environment and make reliably good decisions at every step. This general mechanism can be applied to autonomous vehicles, chatbots, robotics, automated stock trading, healthcare and much more. We'll be exploring some of these applications in the next section and throughout this book.

In this book, we spend most of the time learning how to structure problems into our standard model and how to implement sufficiently powerful learning algorithms (agents) to solve difficult problems. For our cases, you don't need to do construct environments, you'll just be plugging into existing environments (such as game engines or other APIs). For example, OpenAI has released a Python Gym library that provides us with a plethora of environments and an easy interface for our learning algorithm to interact with them (Figure 1.5). The code on the left shows just how simple it is to setup and use one of these environments, a 9x9 Go game in only 5 lines of code.

```
import gym
env = gym.make('Go9x9-v0')
env.reset()
env.step(action)
env.render()
```

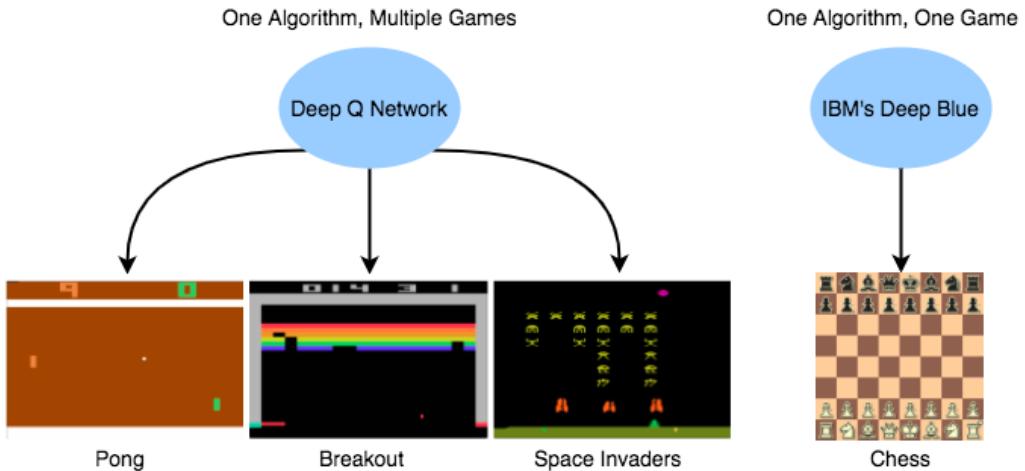
	A	B	C	D	E	F	G	H	J
9		.	.	.	.	X	.	.	
8		X	.	.	.	.	.	.	
7		.	.	.	.	.	.	.	
6		.	.	.	.	.	.	.	
5		.	.	.	.	.	.	.	
4		.	.	.	.	.	.	.	
3		.	.	0	.	.	0	.	
2		.	.	.	.	.	.	.	
1		.	.	.	.	.	.	.	

Figure 1.7. The OpenAI python library comes with many environments and an easy interface for a learning algorithm to interact with them. With just a few lines of code we've loaded up a 9x9 Go board.

## 1.5 What can I do with Reinforcement Learning?

We began this chapter by exploring the history of artificial intelligence and how it leads us to believe that, although recent successes in supervised learning are important and useful, supervised learning is not going to get us to artificial general intelligence (AGI). We ultimately seek general purpose learning machines that can be applied to multiple problems with minimal to no supervision and whose repertoire of skills can be transferred across domains. Large data-rich companies can gainfully benefit from supervised approaches, but smaller companies and organizations may not have the resources to exploit the power of machine learning. General purpose learning algorithms would level the playing field for everyone

Nonetheless, while general purpose learning machines would be more powerful than domain-specific and supervised machines, there are still reasons to consider using current reinforcement learning technology in practice. Reinforcement learning research and applications are still maturing but there have been many exciting developments in recent years. Google's DeepMind research group has showcased some impressive results and garnered international attention. The first was in 2013 with an algorithm that could play a spectrum of Atari games at superhuman levels. Previous attempts at creating agents to solve these games involved fine-tuning the underlying algorithms to understand the specific rules of the game, often called feature engineering. While these feature engineering approaches can work well for a particular game, they are unable to transfer any knowledge or skills to a new game or domain. DeepMind's Deep Q-Network (DQN) algorithm was robust enough to work on seven games without any game-specific tweaks. It had nothing more than the raw pixels from the screen and was merely told to maximize the score, yet the algorithm learned how to play beyond an expert human level.



**Figure 1.8:** DeepMind's DQN algorithm successfully learned how to play seven Atari games with only the raw pixels as input and the players score as the objective to maximize. Previous algorithms, such as IBM's Deep Blue, needed to be fine-tuned to play a specific game.

Even more recently is DeepMind's AlphaGo and AlphaZero algorithms, which beat the world's best players at the ancient Chinese game Go. Experts believed artificial intelligence (AI) would not be able to play Go competitively for at least another decade. Players do not know the best move to make at any given turn and only receive feedback for their actions at the end of the game. Many high level players saw themselves as artists rather than calculating strategists and described winning moves as being beautiful or elegant. These are characteristics that algorithms typically don't handle well. With over  $10^{170}$  legal board positions, brute force algorithms (like what IBM's Deep Blue used to beat Chess) were not feasible. AlphaGo managed this feat largely by playing simulated games of Go millions of times and learning which actions maximized the rewards of playing the game well. Similarly to the Atari case, AlphaGo only had access to the same information a human player would, where the pieces were on the board.

While algorithms that can play games better than humans are remarkable, the promise and potential of RL goes far beyond making better game bots. DeepMind was able to create a model to decrease Google's data center cooling cost by 40%, something we explored earlier in this chapter as just an example. Autonomous vehicles use RL to learn which series of actions (accelerating, turning, breaking, signaling) leads to passengers reaching their destinations on time and how to avoid accidents. And researchers are training robots to complete tasks, such as learning to run, without needing to explicitly program complex motor skills.

Many of these examples are high-stakes, like driving a car. You of course cannot just let a learning machine learn how to drive a car by trial and error. Fortunately, there are an increasing number of successful examples of letting learning machines loose in some harmless

simulator and then once it has mastered the simulator, try it on real hardware in the real world. One instance of this that we will explore in this book is algorithmic trading. A substantial fraction of all stock trading is actually executed by computers with little to no input from human operators. Most of these algorithmic traders are wielded by huge hedge funds managing billions of dollars. In the last few years, however, we've seen more and more interest by individual traders in building trading algorithms. Indeed, Quantopian is a company that provides a platform where individual users can write trading algorithms in Python and test them in a safe, simulated environment. If their algorithms perform well, they can use them to trade real money. Many traders have achieved relative success with simple heuristics and rule-based algorithms, however, equity markets are dynamic and unpredictable, so a continuously learning reinforcement learning algorithm has the advantage of being able to adapt to changing market conditions in real-time.

One practical problem we tackle early on in this book is advertisement placement. Many web businesses derive significant revenue from advertisements, and the revenue from ads is often tied to the number of clicks those ads can garner. Hence there is a big incentive to place advertisements that maximize clicks. The only way to do this, however, is by utilizing knowledge about the users to place the most appropriate ads. Unfortunately, we generally don't know what characteristics of the user are related to the right ad choices. However, we can employ reinforcement learning techniques to make some headway. If we give an RL algorithm some potentially useful information about the user (what we would call the environment, or state of the environment) and just tell it to maximize ad clicks, then it will learn how to associate its input data to its objective, and will eventually learn which ads will produce the most clicks given a particular user.

## 1.6 Why Deep Reinforcement Learning?

Okay, so reinforcement learning sounds pretty interesting, but why *deep* reinforcement learning? RL has existed long before the popular rise of deep learning. In fact, some of the earliest methods (which we will introduce for learning purposes) involved nothing more than storing experiences in a lookup table (e.g. a Python dictionary) and updating that table on each iteration of the algorithm.

## Game Play Lookup Table

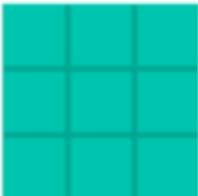
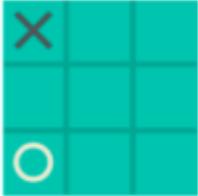
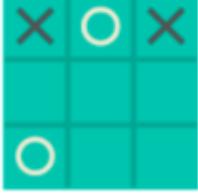
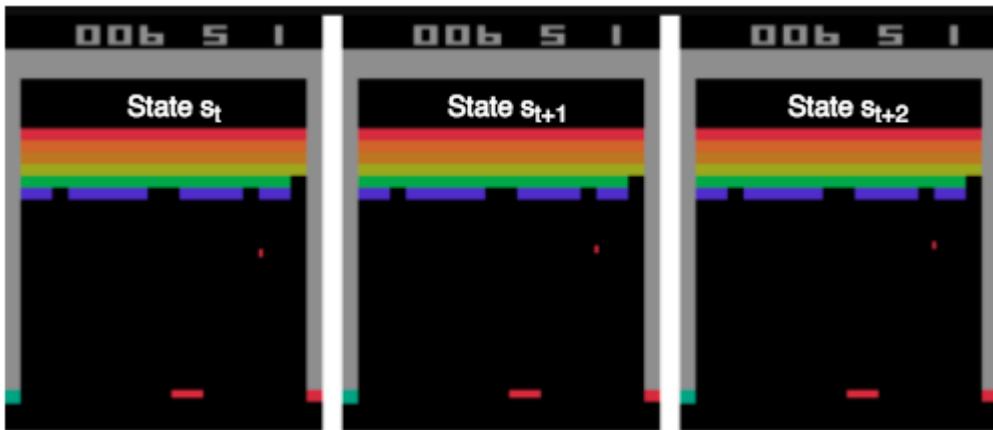
<b>Key</b>	<b>Value</b>
<b>Current State</b>	<b>Action to Take</b>
	<b>Place X in Top Left</b>
	<b>Place X in Top Right</b>
	<b>Place X in Bottom Right</b>

Figure 1.9: An action lookup table for Tic-Tac-Toe with only three entries where the player plays X. When the player is given a board position, the lookup table dictates the move that they should make next. There will be an entry for every possible state in the game.

The idea was to let the agent play around in the environment and just see what happens, but store its experiences of what happens in some sort of database. After awhile, you can look back on your database of knowledge and just observe what worked and what didn't. No neural networks or any other fancy algorithms. For very simple environments this actually works fairly well. For example, in Tic-Tac-Toe there are 255,168 valid board positions. The lookup table (also called a memory table) would have that many entries which mapped from each state to a specific action (Figure 1.5). During training, the algorithm can learn which move leads towards more favorable positions and update that entry in the memory table.

However, once the environment gets more complicated, using a memory table becomes intractable. For example, every screen configuration of a video game can be considered a different state. Imagine trying to store every possible combination of valid pixel values that

may present on screen in a video game! DeepMind's DQN algorithm that played Atari was fed four  $84 \times 84$  grey-scaled images at each step, which would lead to  $256^{28228}$  unique game states (256 different shades of grey per pixel,  $4 \times 84 \times 84 = 28228$  pixels). This number is much larger than the number of atoms in the observable universe and would definitely not fit in computer memory. And this was after the images were scaled down to reduce their size from the original  $210 \times 160$  pixel color images.



**Figure 1.10.** A series of three frames of breakout. The placement of the ball is slightly different in each frame. If using a lookup table, this would equate to storing three unique entries in the lookup table. A lookup table would be impractical as there are way too many game states to store.

If one pixel is changed, the game is considered to be in a different state and would require another entry in a lookup table. However, we could try to limit the possibilities. In the game Breakout you control a paddle at the bottom of the screen that can move right or left; the objective of the game is to deflect the ball and break as many blocks located on the top of the screen. In that case we could define constraints - only look at the states as the ball is returning back to the paddle since our actions are not important while we are waiting for the ball at the top of the screen - or provide our own features - instead of providing the raw image, just provide the position of the ball, paddle, and the remaining blocks. However, these methods require the programmer to understand the underlying strategies of the game and would not generalize to other environments.

That's where deep learning comes in. A deep learning algorithm can learn to abstract away the details of specific arrangements of pixels and learn the important features of a state. Since a deep learning algorithm only has a finite number of parameters, we can use it to compress any possible state into something we can efficiently process, and then use that new representation to make our decisions. By using neural networks, the Atari DQN only had 1792 parameters (convolutional neural network with 16  $8 \times 8$  filters, 32  $4 \times 4$  filters, 256 node fully

connected hidden layer) as opposed to the  $256^{28228}$  key/values that would be needed to store the entire state space.

In the case of the Breakout game, a deep neural network might learn on its own to recognize the same high level features a programmer would have to hand engineer in a lookup table approach. That is, it might learn how to “see” the ball, the paddle, the blocks and recognize the direction of the ball. That’s pretty amazing given that it’s only being given raw pixel data. And even more interesting is that the learned high level features may be transferable to other games or environments.

Deep learning is the secret sauce that makes all the recent successes in reinforcement learning possible. No other class of algorithms has demonstrated the representational power, efficiency and flexibility as deep neural networks. Moreover, neural networks are actually fairly simple!

## 1.7 Why this book?

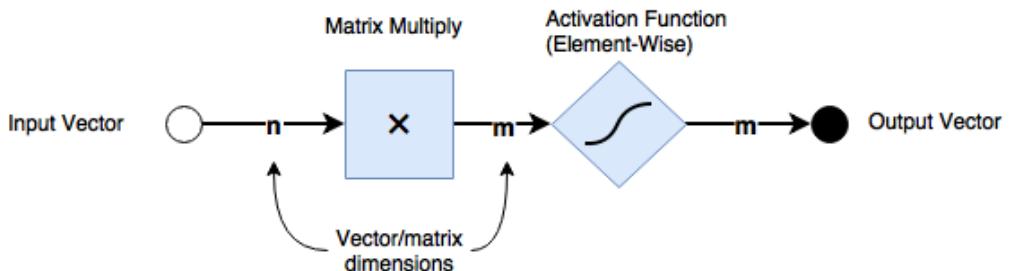
There are a number of resources available that cover reinforcement learning, why choose this book? The fundamental concepts of reinforcement learning have been well-established for decades, but the field is moving very fast, so teaching any particular new result is almost surely going to be short-lived. That’s why this book focuses on teaching skills not details with short half-lives. We do cover some recent advances in the field that will surely be supplanted in the not too distant future, however, we do so only to build new skills, not because the particular topic we’re covering is necessarily a time-tested technique. We’re confident that even if some of our examples become dated, the skills you learn will not and you’ll be prepared to tackle reinforcement learning problems for a long time to come.

Moreover, reinforcement learning is a huge field with a lot to learn. We can’t possibly hope to cover all of it in this book. Rather than be an exhaustive RL reference or comprehensive course, our goal is to teach you the foundations of RL with applications and sample a few of the most exciting recent developments in the field. We expect that you will be able to take what you’ve learned here and easily get up to speed on many other areas of RL that we left out. Plus, we have a section at the end that gives you a roadmap for what areas you might want to check out after finishing this book.

We also aim to write a book that is focused on teaching well, but also rigorously. Reinforcement learning and deep learning are both fundamentally mathematical. If you read any primary research articles in these fields, you will encounter potentially unfamiliar mathematical notation and equations. Mathematics allows us to make precise statements about what’s true, how things are related, and offers rigorous explanations for how and why things work. We could teach reinforcement learning without any math and just use Python, however, that approach would handicap you in understanding future advances.

So we think the math is important, but as our editor noted, there’s a common saying in the publishing world: “for every equation in the book, the readership is halved,” which probably has some truth to it. There’s an unavoidable cognitive overhead in deciphering complex math

equations, unless you're a professional mathematician who reads and writes math all day. Faced with wanting to present a rigorous exposition of deep reinforcement learning to give our readers a top-rate understanding and yet wanting to reach as many people as possible, we came up with what we think is a very distinguishing feature of this book. As it turns out, even professional mathematicians are becoming tired of traditional math notation with its huge array of symbols, and so within a particular branch of advanced mathematics called category theory, mathematicians have developed a purely graphical language called *string diagrams*. String diagrams look very similar to flow-charts and circuit diagrams and have a fairly intuitive meaning, yet they are just as rigorous and precise as traditional mathematics notation largely based on Greek and Latin symbols.



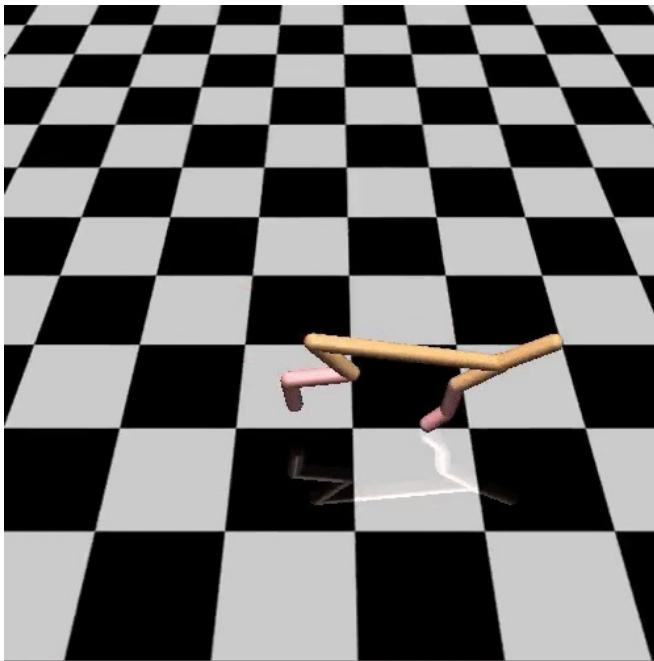
**Figure 1.11:** A string or flow diagram for a single layer neural network. Reading from left to right, this represents a function that accepts an input vector of dimension  $n$ , multiplies it by a matrix of dimensions  $n \times m$ , returning a vector of dimension  $m$ . Then a non-linear (activation) function is applied to each element in the vector and the resulting vector is returned from the function as the output.

Figure 1.11 shows a simple example of the type of diagrams we will frequently use throughout the book to communicate everything from complex mathematical equations to the architectures of deep neural networks. We will describe this graphical syntax in the next chapter and we'll continue to refine and build it up throughout the rest of the book. In some cases this graphical notation is overkill for what we're trying to explain, so we'll use a combination of clear prose and Python or pseudocode. We also include traditional math notation in most cases, so the bottom line is you will be able to learn the underlying mathematical concepts one way or another, whether diagrams, code or normal notation most connect with you.

## 1.8 What's next?

In the next chapter, we dive right into the real meat of reinforcement learning, covering many of the core concepts such as the tradeoff between exploration and exploitation, Markov Decision Processes, value functions, and policies (these words will make sense soon). But first, in the beginning of the next chapter we'll introduce some of the teaching methods we'll

employ for the rest of the book. The rest of the book will cover core deep reinforcement learning algorithms that much of the latest research is built upon starting with Deep Q Networks, followed by Policy Gradient approaches then to Model-Based algorithms. We will primarily be utilizing OpenAI's Gym (mentioned earlier) to train our algorithms to understand nonlinear dynamics, control robots (Figure 1.7) and play Go.



**Figure 1.12. The Half-Cheetah Mujoco environment from the OpenAI Gym. We will be implement a Policy Gradient algorithm to solve them in Chapter 6.**

In each chapter, we will open with a major problem or project that we will use to illustrate the important concepts and skills for that chapter. As each chapter progresses, we may add complexity or nuance to the starting problem to go deeper into some of the principles. For example, in chapter 2 we start with a problem of maximizing rewards at a casino slot machine and by solving that problem we cover much of foundational reinforcement learning. Later we add some complexity to that problem and change the setting from a casino to a business that needs to maximize advertisement clicks, which allows us to round out a few more of the core concepts.

Although this book is for those who already have experience with the basics of deep learning, we expect to not only teach you fun and useful reinforcement learning techniques but also to hone your deep learning skills. In order to solve some of the more challenging

projects, we'll need to employ some of the latest advances in deep learning such as generative adversarial networks, evolutionary methods, meta-learning and transfer learning. Again, this is all in line with our skills-focused mode of teaching, so the particulars of any of these advances is not what's important.

## 1.9 Summary

- Reinforcement learning is a subclass of machine learning algorithms that learn by maximizing rewards in some environment. These algorithms are useful when the problem involves making decisions or taking actions. Reinforcement learning algorithms can, in principle, employ any statistical learning model, however, it has become increasingly popular and effective to use deep neural networks.
- The agent is the core of any reinforcement learning problem. It is the part of the reinforcement learning algorithm that processes input information to take an action. In this book we are primarily focused on agents implemented as deep neural networks.
- The environment is the potentially dynamic conditions in which the agent operates. More generally, the environment is whatever process generates the input data for the agent. For example, we might have an agent flying a plane in a flight simulator, so the simulator is the environment.
- The state is a snapshot of the environment that an agent has access to and uses to make decisions. The environment is often a set of constantly changing conditions, however, we can sample from the environment and these samples at particular time points are the state information of the environment we give to the agent.
- An action is a decision made by an agent that produces a change in its environment. Moving a particular chess piece is an action and so is pressing the gas pedal in a car.
- A reward is a positive signal given to an agent by the environment after taking a "good" action. The rewards are the only learning signal the agent is given. The objective of a reinforcement learning algorithm (i.e. the agent) is to maximize rewards.
- The general pipeline for an RL algorithm is a loop in which the agent receives input data (the state of the environment), the agent evaluates that data and takes an action among a set of possible actions given its current state, the action changes the environment, the environment then sends a reward signal and new state information to the agent and the cycle repeats. When the agent is implemented as a deep neural network, then at each iteration we are evaluating a loss function based on the reward signal and backpropagating to improve the performance of the agent.

### REFERENCES:

- "On Computable Numbers, with an Application to the Entscheidungsproblem" Alan Turing, 1936
- "The History of Artificial Intelligence" University of Washington  
[< https://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf >](https://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf)

- "Neural Networks - History: The 1940's to the 1970's" <  
[>](https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html)
- [>](https://en.wikipedia.org/wiki/Hebbian_theory)
- [>](http://www.psych.utoronto.ca/users/reingold/courses/ai/cache/neural4.html)
- [>](https://en.wikipedia.org/wiki/ADALINE)
- "Reinforcement Learning: An Introduction" 2<sup>nd</sup> Edition, Sutton and Barto, 2012 draft
- [>](http://www.scholarpedia.org/article/Reinforcement_learning#Background_and_History)
- "RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING"
- <[>](http://smo.sogang.ac.kr/doc/dy_birth.pdf)
- [>](https://en.wikipedia.org/wiki/Dynamic_programming)

# 2

## *Modeling Reinforcement Learning Problems: Markov Decision Processes*

### This chapter covers:

- String diagrams and our teaching methods
- The PyTorch deep learning framework
- Solving N-armed bandit problems
- Balancing exploration versus exploitation
- Modeling a problem as a Markov decision process (MDP)
- Implementing a neural network to solve an advertisement selection problem

### 2.1 String Diagrams and our teaching methods

This chapter covers some of the most fundamental concepts in all of reinforcement learning and will be the basis for the rest of the book. But before we get into that, we want to first go over some of the recurring teaching methods we'll employ in this book, most notably, the string diagrams we mentioned last chapter.

From our experience, when most people try to teach something complicated they tend to teach it in the reverse order in which the topic itself was developed. They'll give you a bunch of definitions, terms, descriptions and perhaps theorems and then they'll say, "great, now that we've covered all the theory, let's go over some practice problems." In our opinion, that's exactly the opposite order in which things should be presented. Most good ideas arise as solutions to real problems in the world, or at least imagined problems. The problem-solver

stumbles across a potential solution, tests it, improves it, and then eventually it gets formalized and possibly mathematized. All the terms and definitions come *after* the solution to the problem was thought of.

We think learning something is most motivating and effective when you take the place of that original idea-maker, who was thinking of how to solve a particular problem. Only once the intuition of the solution crystalizes does it warrant formalization, which is indeed necessary to establish its correctness and to faithfully communicate it to others in the field. There is a powerful urge to engage in this reverse chronological mode of teaching, but we will do our best to resist it and develop the topic as we go. In that spirit, we will introduce new terms, definitions, and mathematical notation as we need them. For example, we will uses “callout” boxes like this:

### **DEFINITION (NEURAL NETWORK):**

A neural network is a kind of machine learning model composed of multiple “layers” that perform a matrix-vector multiplication followed by the application of a non-linear “activation” function. The matrices of the neural network are the model’s learnable parameters and are often called “weights” of the neural network.

You will only see these callout boxes once per term, but we will often repeat the definition in different ways in the text to make sure you really understand it and remember it. This is a course on reinforcement learning, not a textbook or reference, so we won’t shy away from repeating ourselves when we think it’s something important to remember.

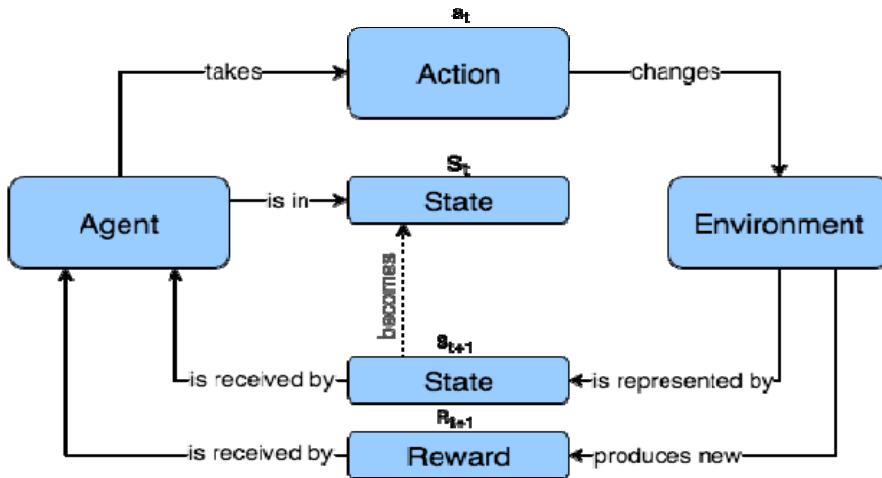
Whenever we need to introduce some math, we typically use a box showing the math and a pseudo-Python version of the same underlying concept. Sometimes it’s easier to think in terms of code or the math, and in any case, we think it’s good to get familiarized with both. As a super simple example, if we were introducing the equation of a line, we would do it like this:

Math	Pseudocode
$y=mx+b$	<pre>def line(x,m,b):     return m*x + b</pre>

We also include plenty of inline code (short snippets) and code-listings (longer code examples) and the code for complete projects. All of the code in the book is provided in Jupyter Notebooks categorized by chapter on the book’s GitHub repository <<https://github.com/DeepReinforcementLearning/>>.

Since reinforcement learning involves a lot of interconnecting concepts that can become confusing when left just in words, we include a lot of diagrams and figures of varying sorts. The two more important kinds of figures we use are **string diagrams** and ologs (“oh-log”). They perhaps have odd names, but they’re both really simple ideas and are adapted from category theory, that branch of math we mentioned in the first chapter where they tend to use a lot of diagrams to supplement or replace traditional symbolic notation.

We've already come across an olog when we introduced the general framework for reinforcement learning in chapter 1:

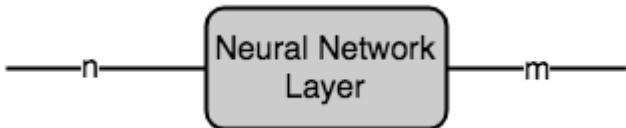


The idea is that the boxes contain nouns or noun-phrases while the arrows are labeled with verbs or verb-phrases. It's a small difference from typical flow-diagrams, but it makes it easy to translate the olog diagram into English prose and vice versa and also makes very clear what exactly the arrows are *doing* functionally.

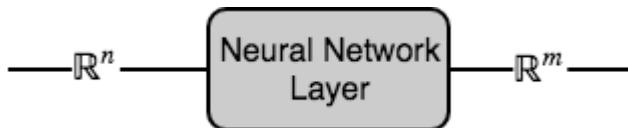
Lastly, string diagrams (sometimes referred to as wiring diagrams in other sources) are also a flow-like diagrams that represent the flow of typed data along strings (i.e. directed or undirected arrows) into processes (computations, functions, transformations, etc.) represented as boxes. The important difference between string diagrams and similar looking flow diagrams is that all data on the wires is explicitly typed (e.g. a numpy array with shape or simply a floating point number) and that they are fully compositional. By compositional, we mean that we can "zoom in" or out on the diagram in a principled way, to see the bigger more abstract picture or drill down to the computational details.

If we're showing a more high-level depiction, the process boxes may just be labelled with a word or short phrase indicating the kind of process that happens, but we can also show a zoomed-in view of that process box that reveals all its internal details, composed of its own set of sub-strings and sub-processes. The compositional nature of these diagrams also means that we can plug parts of diagrams into other diagrams, forming more complex diagrams, as long as all the strings' types are compatible.

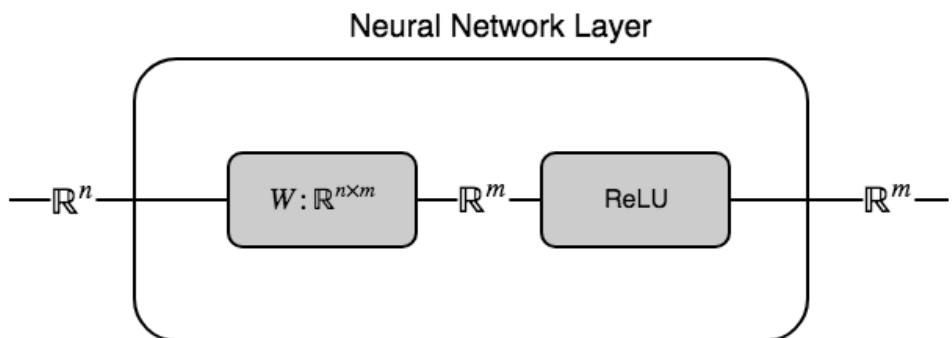
For example, here's a single layer of a neural network as a string diagram:



Reading from left to right, we see some data of type "n" flows into a process box called "Neural Network Layer" and produces an output of type "m." Since neural networks typically take vectors as inputs and produce vectors as outputs, these types refer to the dimensions of the input and output vectors, respectively. That is, this neural network layer accepts a vector of length or dimension  $n$  and produces a vector of dimension  $m$ . It's possible that  $n=m$  for some neural network layers. This manner of "typing" the strings is simplified and we do it only when it's clear what the types mean from the context. In other cases, we may employ mathematical notation such as  $\mathbb{R}$  for the set of all real numbers, which in programming languages basically translates to floating point numbers. So for a vector of floating point numbers with dimension  $n$  we could type the strings like this:



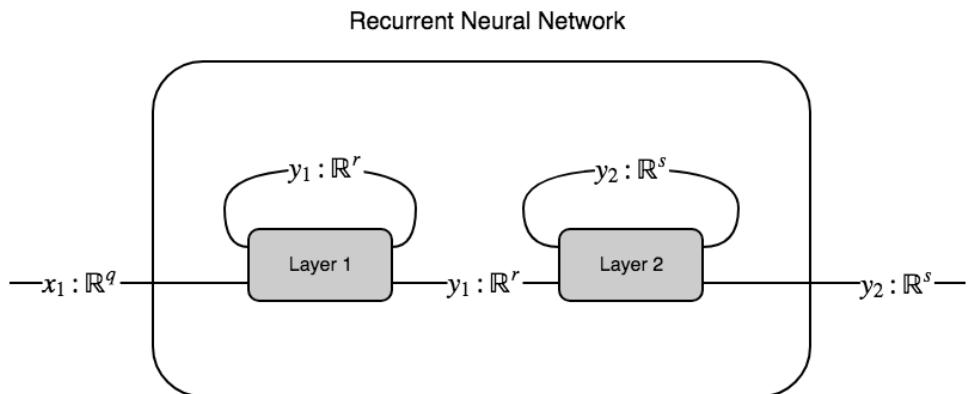
Now the typing is richer, we not only know the dimensions of the input and output vectors, we know that they're real/floating point numbers. While this is almost always the case, sometimes we may be dealing with integers or binary numbers. In any case, our process box "Neural Network Layer" is left as a black-box, we don't know exactly what's going on in there other than the fact that it transforms a vector into another vector of possibly different dimensions. We can decide to zoom-in on this process to see what specifically is happening:



Now we can see the inside of the original process box, and it is composed of its own set of sub-processes. We can see that our  $n$ -dimensional vector gets multiplied by a matrix of dimension  $n \times m$ , which produces an  $m$ -dimensional vector product. This vector then passes

through some process called “ReLU,” which you may recognize as the standard neural network activation function, the rectified linear unit. We could continue to zoom-in on the ReLU sub-sub-process if we wanted. Hence, anything that deserves the name string diagram must be able to be scrutinized at any level of abstraction and remain well-typed (meaning the types of data entering and exiting processes are compatible and make sense, e.g. a process that is supposed to produce sorted lists should not be hooked up to another process that expects integers) at any level.

As long as the strings are well-typed, then we can string together a bunch of processes into a complex system. This allows us to build components once and re-use them wherever they’re type matched. At a somewhat high level, we might depict a simple two-layer recurrent neural network (RNN) like this:



So this RNN takes in a  $q$ -vector and produces an  $s$ -vector. However, we can see the inside processes. There are two layers, each one looks identical in its function. They each take in a vector and produce a vector, except that the output vector gets copied and fed back into the layer process as part of the input, hence the recurrence.

String diagrams are a very general type of diagram; in addition to diagramming neural networks, we could use them to diagram how to bake a cake. A special kind of string diagram is a **computational graph**. A computational graph is a string diagram where all the processes represent concrete computations that a computer can perform, or that can be described in some programming language like Python. If you’ve ever visualized a computational graph in TensorFlow’s TensorBoard, then that’s what we mean.

## 2.2 Solving the Multi-Arm Bandit

Okay, we’re ready to get started with a real reinforcement learning problem and learn the relevant concepts and skills needed to solve this problem as we go. But before we get too fancy building AlphaGo, let’s first consider a simple problem (that is easily translated into a practical problem). Let’s say you’re at a casino and in a section with some slot machines. In

front of you are 10 slot machines in a row with a flashy sign that says "Play for free! Max payout is \$10!" Wow, not bad right! Intrigued, you ask one of the employees what's going on here, it seems too good to be true, and she says "It's really true, play as much as you want, it's free. Each slot machine is guaranteed to give you a reward between \$0 and \$10. Oh, by the way, keep this to yourself, but those 10 slot machines each have a different average payout, so try to figure out which one gives out the most rewards on average and you'll be making tons of cash!"

What kind of casino is this?! Who cares, let's just figure out how to make the most money! Oh by the way, here's a joke: What's another name for a slot machine? .... A one-armed bandit! Get it? It has one arm (a lever) and it generally steals your money! Huh, well I guess we could call our situation a 10-armed bandit problem, or an  $n$ -armed bandit problem more generally, where  $n$  is the number of slot machines. While this problem sounds pretty fanciful so far, later we'll see that these so-called  $n$ -armed bandit (or multi-armed bandit) problems do have some very practical applications.

Let me restate our problem more formally. We have  $n$  possible actions (here  $n = 10$ ) where an action means pulling the arm/lever of a particular slot machine and at each play ( $k$ ) of this game we can choose a single lever to pull. After taking an action  $a$  we will receive a reward  $R_k$  (reward at play  $k$ ). Each lever has a unique probability distribution of payouts (rewards). For example, if we have 10 slot machines and play many games, slot machine #3 may give out an average reward of \$9 whereas slot machine #1 only gives out an average reward of \$4. Of course, since the reward at each play is probabilistic, it is possible that lever #1 will by chance give us a reward of \$9 on a single play. But if we play many games, we expect on average slot machine #1 is associated with a lower reward than #3.

In words, our strategy should be to play a few times, choosing different levers and observing our rewards for each action. Then we want to only choose the lever with the largest observed average reward. Thus we need a concept of expected reward for taking an action  $a$  based on our previous plays. We'll call this expected reward  $Q_k(a)$  mathematically; you give the function an action (given we're at play  $k$ ) and it returns the expected reward for taking that action. Formally,

Math	Pseudocode
$Q_k(a) = \frac{R_1 + R_2 + \dots + R_k}{k_a}$	<pre>def exp_reward(action, history):     rewards_for_action = history[action]     return sum(rewards_for_action) / len(rewards_for_action)</pre>

That is, the expected reward at play  $k$  for action  $a$  is the arithmetic mean of all the previous rewards we've received for taking action  $a$ . Thus our previous actions and observations influence our future actions, we might even say some of our previous actions reinforce our current and future actions. We'll come back to this later. The function  $Q_k(a)$  is called a value

function because it tells us the value of something. In particular, it is an actionvalue function because it tells us the value of taking a particular action. Since we typically denote this function with the symbol  $Q$ , it also often called a  $Q$  function. We'll come back to value functions later and give a more sophisticated definition, but this will suffice for now.

That is, the expected reward at play for action  $a$  is the arithmetic mean of all the previous rewards we've received for taking action  $a$ . Thus our previous actions and observations influence our future actions, we might even say some of our previous actions *reinforce* our current and future actions. We'll come back to this later. The function  $\pi$  is called a *value function* because it tells us the value of something. In particular, it is an *action value function* because it tells us the value of taking a particular action. Since we typically denote this function with the symbol  $Q$ , it also often called a  $Q$  *function*. We'll come back to value functions later and give a more sophisticated definition, but this will suffice for now.

**EXPLORATION AND EXPLOITATION.** When we first start playing, we need to play the game and observe the rewards we get for the various machines, we can call this strategy *exploration*, since we're just exploring the results of our actions essentially randomly. This is in contrast to a different strategy we could employ called *exploitation*, which means we use our current knowledge about which machine seems to produce the most rewards and just keep playing that machine. So our overall strategy needs to include some amount of exploitation (simply choosing the best lever based on what we know so far) and some amount of exploration (choosing random levers so we can learn more). The proper balance of exploitation and exploration will be important to maximizing our rewards.

So how can we come up with an algorithm to figure out which slot machine has the largest average payout? Well, the simplest algorithm would be to select action  $a$  for which this equation is true:

Math	Pseudocode
$a = \arg\max_a (Q_k(a))$ $\forall a \in A_k$	<pre>def get_best_action(actions, history):     exp_rewards = [exp_reward(action, history)     for action in actions]     return argmax(exp_rewards)</pre>

Or as legitimate Python 3 code:

```
def get_best_action(actions):
    best_action = 0
    max_action_value = 0
    for i in range(len(actions)): #loop through all possible actions
        cur_action_value = get_action_value(actions[i]) #get the value of the current action
        if cur_action_value > max_action_value:
            best_action = i
            max_action_value = cur_action_value
    return best_action
```

This equation states that the expected reward for the current play  $k$  for taking action  $a$  is equal to the maximum average reward of all previous actions taken. In other words, we use our above reward function  $Q_k(a)$  on all the possible actions and select the action that returns the maximum average reward. Since  $Q_k(a)$  depends on a record of our previous actions and their associated rewards, this method will not evaluate actions that we haven't already explored. Thus we might have previously tried lever 1 and lever 3, and noticed that lever 3 gives us a higher reward, but with this method, we'll never think to try another lever, say #6, which, unbeknownst to us, actually gives out the highest average reward. This method of simply choosing the best lever that we know of so far is called a "greedy" method (or as we discussed, exploitation)

**$\epsilon$ -GREEDY STRATEGY.** Obviously, we need to have some exploration of other levers (slot machines) going on to discover the true best action. One simple modification to our above algorithm is to change it to an  $\epsilon$ (epsilon)-greedy algorithm, such that, with a probability  $\epsilon$ , we will choose an action  $a$  at random, and the rest of the time (probability  $1-\epsilon$ ) we will choose the best lever based on what we currently know from past plays. So most of the time we play greedy, but sometimes we take some risks and choose a random lever and see what happens. This will of course influence our future greedy actions.

Let's see if we can solve this in code with Python.

```
import numpy as np
from scipy import stats
import random
import matplotlib.pyplot as plt

n = 10
arms = np.random.rand(n) #hidden probs associated with each arm
eps = 0.1
```

Per our casino example, we will be solving a 10-armed bandit problem, hence  $n = 10$ . We've also defined a numpy array of length  $n$  filled with random floats that can be understood as probabilities. The way we've chosen to implement our reward probability distributions for each arm/lever/slot machine is this: Each arm will have a probability, e.g. 0.7. The maximum reward is \$10. We will setup a `for` loop to 10 and at each step, it will add +1 to the reward if a random float is less than the arm's probability. Thus on the first loop, it makes up a random float (e.g. 0.4). 0.4 is less than 0.7, so `reward += 1`. On the next iteration, it makes up another random float (e.g. 0.6) which is also less than 0.7, thus `reward += 1`. This continues until we complete 10 iterations and then we return the final total reward, which could be anything between 0 and 10. With an arm probability of 0.7, the *average* reward of doing this to infinity would be 7, but on any single play, it could be more or less.

```
def reward(prob, n=10):
    reward = 0;
    for i in range(n):
        if random.random() < prob:
            reward += 1
```

```
    return reward
```

The next function we define is our greedy strategy of choosing the best arm so far. This function will accept a list of tuples that contains a history of actions and rewards.

```
[(1,8), (2,4), (1,7)]
```

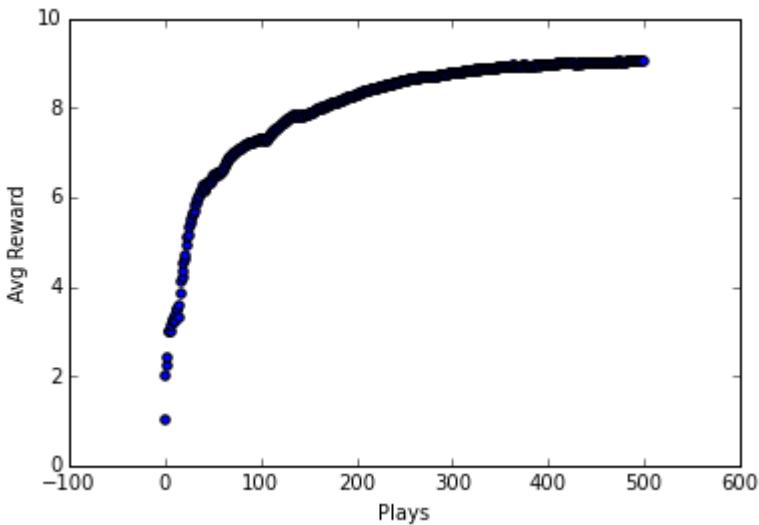
In the above example, there are 3 samples (we have pulled three arms). We first pulled arm 1 and received \$8. We then pulled arm 2 and received \$4. For our last action we decided to pull arm 1 again and we received \$7 this time.

Provided this experience array, we can now create a function that will determine the arm that has provided us the highest average return and thus should be pulled again. In the above case, arm 1 has provided us the highest average return  $(8 + 7) / 2 = 7.5$ .

```
def get_best_arm(pastRewards, actions):
    bestArm = 0 #just default to 0
    bestMean = 0
    for action in actions:
        avg = np.mean(pastRewards[np.where(pastRewards[:,0] == action)][:, 1])
        if avg > bestMean:
            bestMean = avg
            bestArm = action
    return bestArm
```

And here is the main loop for each play. If a random number is greater than epsilon parameter, then we just calculate the best action using the `get_best_arm` function and take that action, otherwise we take a random action. I've set it to play 500 times and display a matplotlib scatter plot of the mean reward against plays. Hopefully we'll see that the mean reward increases as we play more times.

```
plt.xlabel("Plays")
plt.ylabel("Avg Reward")
for i in range(500):
    if random.random() > eps:
        choice = get_best_arm(pastRewards, actions)
    else:
        choice = np.where(arms == np.random.choice(arms))[0][0]
    thisAV = np.array([[choice, reward(arms[choice])]])
    av = np.vstack((av, thisAV))
    percCorrect = 100*(len(av[np.where(av[:,0] == np.argmax(arms))])/len(av))
    runningMean = np.mean(av[:,1])
    plt.scatter(i, runningMean)
```



As you can see, the average reward does indeed improve after many plays. Our algorithm is *learning*, it is getting reinforced by previous good plays! And yet it is such a simple algorithm.

The problem we've considered here is a *stationary* problem because the underlying reward probability distributions for each arm do not change over time. We certainly could consider a variant of this problem where this is not true, a non-stationary problem. In this case, a simple modification would be to weight more recent action-value pairs greater than distant ones, thus if things change over time, we will be able to track them. Beyond this brief mention, we will not implement this slightly more complex variant here.

**SOFTMAX SELECTION POLICY.** Imagine another type of bandit problem: A newly minted doctor specializes in treating patients with heart attacks. She has 10 treatment options of which she can choose only one to treat each patient she sees. For some reason, all she knows is that these 10 treatments have different efficacies and risk-profiles for treating heart attacks, and she doesn't know which one is the best yet. We could still use our same  $\epsilon$ -greedy algorithm from above, however, we might want to reconsider our  $\epsilon$  policy of completely randomly choosing a treatment once in awhile. In this new problem, randomly choosing a treatment could result in patient death, not just losing some money. So we really want to make sure to not choose the worst treatment but still have some ability to explore our options to find the best one.

This is where a softmax selection might be the most appropriate. Instead of just choosing an action at random during exploration, softmax gives us a probability distribution across our options. The option with the largest probability would be equivalent to our best arm action from above, but then we have some idea about what are the 2nd and 3rd best actions for

example. This way, we can randomly choose to explore other options while avoiding the very worst options. Here's the softmax equation:

### Math

$$\Pr(A) = \frac{e^{Q_k(A)/\tau}}{\sum_{i=1}^n e^{Q_k(i)/\tau}}$$

### Pseudocode

```
def softmax(vals, tau):
    softm = pow(e, vals / tau) / sum( pow(e,
    vals / tau))
    return softm
```

$\Pr(A)$  is a function that accepts an action-value vector (array) and returns a probability distribution over the actions, such that higher value actions have higher probabilities. For example, if your action-value array has 4 possible actions and they all currently have the same value, say  $A = [10, 10, 10, 10]$ , then  $\Pr(A) = [0.25, 0.25, 0.25, 0.25]$ , i.e. all the probabilities are the same and must sum to 1.

The numerator of the fraction just exponentiates the action-value array divided by a parameter  $\tau$ , yielding a vector of the same size as the input. The denominator sums over the exponentiation of each individual action-value divided by  $\tau$ , yielding a single number.

$\tau$  is a parameter called temperature the scales the probability distribution of actions. A high temperature will tend the probabilities to be very similar, whereas a low temperature will exaggerate differences in probabilities between actions. Selecting this parameter requires an educated guess and some trial and error.

When we implement the slot machine 10-armed bandit problem from above using softmax, we don't need our `get_best_arm` function anymore. Since softmax produces a weighted probability distribution across our possible actions, we will just randomly (but weighted) select actions according to their relative probabilities. That is, our best action will get chosen more often because it will have the highest softmax probability, but other actions will be chosen at lesser frequency.

To implement this, we need to create a new numpy array that stores the softmax probabilities of our actions. We'll initialize all the actions to have a uniform small probability of  $1/n$  (since we have 10 actions, each action will be assigned a probability of 0.1). Hence, on our first play of the game, all the actions have an equal probability of being chosen, but from then onward the initial uniform probability distribution will become highly skewed toward actions that yielded the highest rewards.

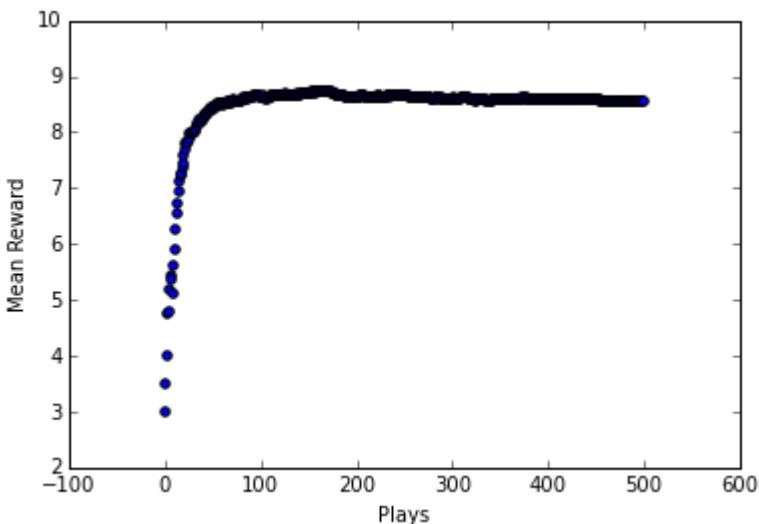
```
n = 10
arms = np.random.rand(n)
av = np.ones(n)
counts = np.zeros(n)
av_softmax = np.zeros(n)
av_softmax[:] = 1.0/n
```

The mathematical exponential is a function call to `np.exp(...)` in numpy. It will apply the function element-wise across the input vector.

```
def softmax(av, tau=1.12):
    softm = ( np.exp(av / tau) / np.sum( np.exp(av / tau) ) )
    return softm
```

Our main for loop is a bit different now. In order to choose an action randomly from our softmax probability distribution, we use numpy's `random.choice` function. `random.choice(arr, p=None)` accepts an array (e.g. `arr`) of values and another array of probabilities (e.g. `p`) for each of those values, then it will randomly select a value in `arr` using the probability distribution in `p`. We will give it `arr = arms`, so it will randomly select an index value corresponding to a particular action, given the probability distribution in `av_softmax`. The update rule we developed in the previous sections remains the same, but we need to recalculate the softmax probabilities stored in the `av_softmax` array after we have a new running mean value for the action we just took.

```
for i in range(500):
    choice = np.random.choice(arms, p=av_softmax)
    counts[choice] += 1
    k = counts[choice]
    rwd = reward(arms[choice])
    old_avg = av[choice]
    new_avg = old_avg + (1/k)*(rwd - old_avg)
    av[choice] = new_avg
    av_softmax = softmax(av)
```



Softmax action selection seems to do better than the epsilon-greedy method; it looks like it converges on an optimal policy faster. The downside to softmax is having to manually select

the  $\tau$  parameter. Softmax here was pretty sensitive to  $\tau$  and it takes awhile of playing with it to find a good value for it. Obviously with epsilon-greedy we had the parameter epsilon to set, but choosing that parameter was much more intuitive.

## 2.3 Applying Bandits to Optimize Ad Placements

The slot machine example doesn't seem to be a particularly real-world problem, but if we just add in one additional element to it, then it does become a practical business problem, one big example being advertisement placement. Whenever you visit a website with ads, the company placing the ads would like to maximize the probability that you will click them.

Let's say we manage 10 e-commerce websites, each focuses on selling a different broad category of retail items such as computers, shoes, jewelry, etc. We would like to increase sales by referring customers who shop on one of our sites to another site that they might be interested in. When a customer checks out on a particular site in our network, we will display an advertisement to one of our other sites in hopes they'll go there and buy something too. Alternatively, we can also place an ad to another product on the same site. Our problem is that we don't know which sites we should refer users to. We could try placing random ads, but we suspect there's a more targeted approach.

**CONTEXTUAL BANDITS.** Perhaps you can see how this is just a new layer of complexity to the N-armed bandit problem we considered at the beginning. At each "play of the game" (each time a customer checks out on a particular website) we have  $N = 10$  possible actions we can take, corresponding to the 10 different types of advertisements we could place. The twist is that the best ad to place may depend on which site in the network the current customer is on. For example, a customer checking out on our jewelry site may be more in the mood to buy a new pair of shoes to go with their new diamond necklace than they would be to buy a new laptop. Thus our problem is to figure out how a particular site relates to a particular advertisement.

This leads us to the introduction of state-spaces. The N-armed bandit problem we started with had an N-element action space (the space or set of all possible actions), but there was no concept of state. That is, there was no information we could use "in the environment" that would help us choose a good arm. The only way we could figure out which arms were good is by trial-and-error. In the ad problem, we know the user is buying something on a particular site, which may give us some information about that user's preferences that we could use to help guide our decision of which ad to place. We call this contextual information a state, and this new class of problems *contextual* bandits.

**KEY POINT:** A state in a game (or reinforcement learning problem more generally) is the set of information available in the environment that can be used to make decisions.

**STATES, ACTIONS, REWARDS.** Before we move on, let's consolidate some of the terms and concepts we've introduced so far. Reinforcement learning algorithms attempt to model the

world in a way that computers can understand and calculate. In particular, RL algorithms model the world as if it merely involved a set of **states**  $S$  ("state-space"), which are a set of features about the environment, a set of **actions**  $A$  ("action-space") than can be taken in a given state, and **rewards**  $r$  given for taking an action in a given state. When we speak of taking a particular action in a particular state, we often call it a **state-action pair**  $(s, a)$ .

**KEY POINT:** The objective of any RL algorithm is to maximize the rewards over the course of an entire episode.

Since our original N-armed bandit problem did not have a state space, only an action space, we just needed to learn the relationship between actions and rewards. We learned the relationship by using a lookup table to store our experience of receiving rewards with particular actions, i.e. we stored action-reward pairs  $(a_k, r_k)$  where the reward at play  $k$  was an average over all past plays associated with taking action  $a_k$ .

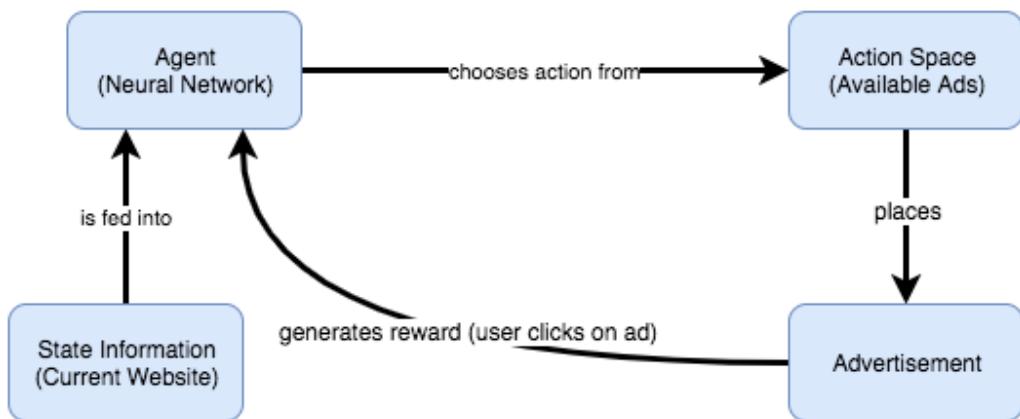


Figure 2.1: Overview of contextual bandit for advertisement placement. The agent (which is a neural network algorithm) receives state information, in this case, the current website the user is on, which it uses to decide which of several advertisements it should place at the checkout step. The users will click on the advertisement or not, resulting in reward signals that get relayed back to the agent for learning.

In our N-armed bandit problem, since we only had 10 actions, a lookup table of 10 rows was very reasonable. When we introduce a state space with contextual bandits, we now start to get a "combinatorial explosion" of possible state-action-reward tuples. For example, if we have a state space of 100 states, and each state is associated with the 10 actions, we now have 1000 different pieces of data we need to store and re-compute over. And in most of the problems we consider in this book, the state space is intractably large, thus a simple lookup table is just not feasible.

That's where deep learning comes in. When properly trained, neural networks are great at learning abstractions that get rid of the details of little value. They can learn composable patterns and regularities in data such that they can effectively compress a large amount of data while retaining the important information. Hence, neural networks can be used to learn complex relationships between state-action pairs and rewards without us having to store all such experiences as raw memories. We often call the part of an RL algorithm that makes the decisions based on some information the **agent**. In order to solve the contextual bandit we've been discussing, we'll employ a neural network as our agent. First though we will take a moment to introduce PyTorch, the deep learning framework we will be using to implement these networks.

## 2.4 Building Networks with PyTorch

There are many Deep Learning frameworks available today, with TensorFlow, MXNet and PyTorch probably being the most popular. We chose to use PyTorch for this book because of its simplicity, it allows you to write native-looking Python code and still get all the goodies of a good framework like automatic differentiation and built in optimization. We'll just give a quick up-front introduction to PyTorch here, but we'll explain more as we go along. If you need to brush up on basic Deep Learning, see the Appendix, we have a fairly detailed Deep Learning review section and more thorough coverage of PyTorch.

If you're comfortable with the NumPy multi-dimensional array, then you can almost just replace everything you do with numpy with PyTorch. For example, here we instantiate a 2x3 matrix in numpy:

```
>>> import numpy

>>> numpy.array([[1, 2, 3], [4, 5, 6]])
array([[1, 2, 3],
       [4, 5, 6]])
```

And here is how you instantiate the same matrix with PyTorch:

```
>>> import torch

>>> torch.Tensor([[1, 2, 3], [4, 5, 6]])
1 2 3
4 5 6
[torch.FloatTensor of size 2x3]
```

**TENSORS.** It's basically the same except in PyTorch we call multi-dimensional arrays **tensors**. Unsurprisingly, this is also the case for TensorFlow and other frameworks, so get used to seeing multi-dimensional arrays referred to as tensors. We can and do refer to the **tensor order**, which is basically how many indexed dimensions it has. This gets a little confusing because sometimes we speak of the dimension of a vector, in which case we just mean the length of the vector. But when we speak of the order of a tensor, we mean how many indices

it has. A vector has one index, meaning every element can be “addressed” by a single index value, and hence is an order 1 tensor or 1-tensor for short. A matrix has two indices, one for each dimension, and is a 2-tensor. Higher order tensors may just be referred to as a  $k$ -tensor where  $k$  is the order, a non-negative integer. On the other end, a single number is actually a 0-tensor, or also called a *scalar*, since it has no indices.

**AUTOMATIC DIFFERENTIATION.** The most important features of PyTorch that we need and that NumPy doesn’t offer is automatic differentiation and optimization. Let’s say we want to setup a simple linear model to predict some data of interest, we can easily define the model using ordinary numpy-like syntax:

```
>>> x = torch.Tensor([2,4]) #input data
>>> m = torch.randn(2, requires_grad=True) #parameter 1
>>> b = torch.randn(1, requires_grad=True) #parameter 2
>>> y = m*x+b #linear model
>>> loss = (torch.sum(y_known - y))**2 #loss function
>>> loss.backward() #calculate gradients
>>> m.grad
tensor([ 0.7734, -90.4993])
```

You simply supply the `requires_grad=True` argument to PyTorch Tensors that you want to compute gradients for, and then call the `backward()` method on the last node in your computational graph, which will backpropagate gradients through all the nodes with `requires_grad=True`. You can then do gradient descent with the automatically computed gradients.

**BUILDING MODELS.** For most of this book, we’ll be too lazy to even deal directly with automatically computed gradients. Instead we’ll use PyTorch’s `nn` module to easily setup a feedforward neural network model and then use the built in optimization algorithms to automatically train the neural network without every having to manually specify the mechanics of backpropagation and gradient descent. Here’s a simple two layer neural network with an optimizer setup:

```
model = torch.nn.Sequential(
    torch.nn.Linear(10, 150),
    torch.nn.ReLU(),
    torch.nn.Linear(150, 4),
    torch.nn.ReLU(),
)
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

We’ve setup a two-layer model with ReLU activation functions, defined a mean-squared error loss function, and setup an optimizer. All we have to do to train this model, given that we have some labeled training data, is start a training loop:

```
for step in range(100):
    y_pred = model(x)
```

```

loss = loss_fn(y_pred, y_correct)
optimizer.zero_grad()
loss.backward()
optimizer.step()

```

The `x` variable is the input data to the model. The `y_correct` variable is a tensor representing the labeled, correct output. We make the prediction using the model, calculate the loss, and then compute the gradients using the `backward()` method on the last node in the computational graph (which is almost always the loss function). Then we just run the `step()` method on the optimizer and it will run a single step of gradient descent. If we need to build more complex neural network architectures than the Sequential model, we can write our own Python class and inherit from PyTorch's module class, and then use that instead.

```

class MyNet(Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.fc1 = Linear(784, 50)
        self.fc2 = Linear(50, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return x

model = MyNet()

```

That's all you need to know about PyTorch for now to be productive with it.

## 2.5 Solving Contextual Bandits

We've built a simulated environment for a contextual bandit as we've described. The simulator includes the state (a number 0-9 representing 1 of the 10 websites in the network), reward generation (ad clicks), and a method to choose an action (which of 10 ads to serve). Below we demonstrate how to use the environment. The only part we need to build is the agent, which is generally the crux of any RL problem, since building an environment is usually just setting up input/output with some data source or plugging into an existing API.

```

from bandit_sim import *

env = ContextBandit(arms=10)
state = env.get_state()
reward = env.choose_arm(1)
print(state)
>>> 2
print(reward)
>>> 8

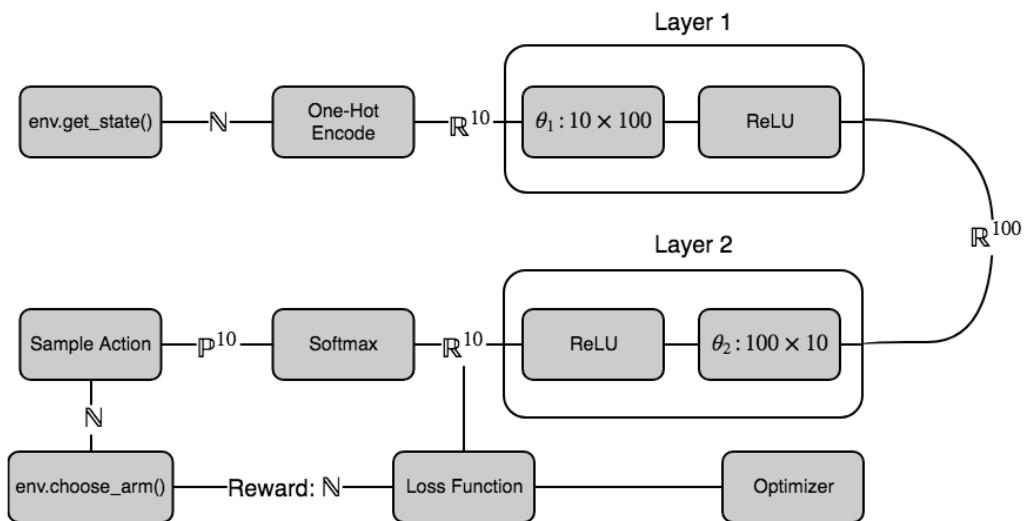
```

The simulator consists of a simple Python class called `ContextBandit` that can be initialized to a specific number of arms. For simplicity, the number of states equals the number of arms, but in general the state-space is often much larger than the action-space. The class has two

methods, one is `get_state()` which is called with no arguments and will return a state sampled randomly from a uniform distribution. In most problems your state will come from a much more complex distribution. Calling `choose_arm(...)` will simulate placing an advertisement and returns a reward (e.g. proportional to the number of ad clicks). We need to always call `get_state` then `choose_arm`, in that order, to continually get new data to learn from. The `bandit_sim` module also includes a few helper functions such as the softmax function and a one-hot encoder. Recall that a one-hot encoded vector is a vector where all but 1 element is set to 0. The only non-zero element is set to 1 and indicates a particular state in the state-space.

Rather than a single static reward probability distribution over N actions like our original bandit problem, the contextual bandit simulator sets up a different reward distribution over the actions for each state. That is, we will have N different softmax reward distributions over actions for each of N states. Hence we need to learn the relationship between the states and their respective reward distributions, then learn which action has the highest probability mass for a given state.

As with all of our projects in this book, we'll be using PyTorch to build the neural network. In this case, we're going to build a 2 layer feedforward neural network that uses rectified linear units (ReLU) as the activation function. The first layer accepts a 10-element one-hot (1-of-K) encoded vector of the state and the final layer returns a 10-element vector representing the predicted reward for each action given the state.



**Figure 2.2:** A computational graph for a simple 10-armed contextual bandit. The `get_state()` function returns a state value, which is transformed into a one-hot vector that becomes the input data for a 2 layer neural network. The output of the neural network is the predicted reward for each possible action, which is a dense vector that is run through a softmax to sample an action from the resulting probability distribution over the actions. The chosen action will return a reward and updates the state of the environment.  $\theta_1, \theta_2$  represent the weight

parameters for each layer. The operation `mul` refers to matrix multiplication, which then gets passed to the activation function  $\sigma$ . Symbol reference:  $N, \mathbb{R}, \mathbb{P}$  denote the natural numbers (i.e. 0, 1, 2, 3, ...), the real numbers (a floating point number, for our purposes), and a probability, respectively. The superscript indicates the length of the vector, so for example,  $P^{10}$  represents a 10-element vector where each element is a probability, such that all the elements sum to 1.

Figure 2.3 shows the forward pass of the algorithm as we've described. Unlike the lookup table approach, our neural network agent is going to learn to predict the rewards that each action will result in for a given state. Then we use the softmax function to give us a probability distribution over the actions and sample from this distribution to choose an arm. Choosing an arm will give us a reward, which we use to train our neural network.

For example, initially our neural network will produce a random vector such as [1.4, 50, 4.3, 0.31, 0.43, 11, 121, 90, 8.9, 1.1] when in state 0. We will run softmax over this vector and sample an action, most likely action 6 (from actions 0 - 9) since that is the biggest number in the vector. Choosing action 6 will generate a reward of say 8. So then we train our neural network to produce the vector [1.4, 50, 4.3, 0.31, 0.43, 11, 8, 90, 8.9, 1.1] since that is the true reward we received for action 6, leaving the rest of the values unchanged. So next time when the neural network sees state 0, it will produce a reward prediction for action 6 closer to 8. As we continually do this over many states and actions, the neural network will eventually learn to predict accurate rewards for each action given a state. Thus our algorithm will be able to choose the best action each time, maximizing our rewards.

```
import numpy as np
import torch as th
from torch.autograd import Variable
from bandit_sim import *

arms = 10
N, D_in, H, D_out = 1, arms, 100, arms
```

$N$  is batch size,  $D_{in}$  is input dimension,  $H$  is hidden dimension and  $D_{out}$  is output dimension. Now we need to set up our neural network model. It is a simple sequential (feedforward) neural network with 2 layers as we described earlier.

```
model = th.nn.Sequential(
    th.nn.Linear(D_in, H),
    th.nn.ReLU(),
    th.nn.Linear(H, D_out),
    th.nn.ReLU(),
)
```

We'll use the mean squared error loss function here, but others will do.

```
loss_fn = th.nn.MSELoss(size_average=False)
```

Now we setup a new environment by instantiating the `ContextBandit` class, supplying the number of arms to its constructor. Remember, we've setup the environment such that the number of arms will be equal to the number of states.

```
env = ContextBandit(arms)
```

The main for-loop of the algorithm is very similar to our original N-armed bandit, however, we had the added step of needing to run a neural network and use the output to select an action. We'll define a function called `train` that accepts the environment instance we created above, the number of epochs we want to train for, and the learning rate. In the function we set a PyTorch variable for the current state, which we need to one-hot encode using the `one_hot(...)` encoding function that we have omitted.

Once we enter the main training for-loop, we run our neural network model with the randomly initialized current state vector. It will return a vector that represents its guess for the values of each of the possible actions. At first, the model will output a bunch of random values since it is not trained. In any case, we run the softmax function over the model's output to generate a probability distribution over the actions and select an action using it. We select an action using the environment's `choose_arm(...)` function, which will return the reward generated for taking that action; it will also update the environment's current state. We turn the reward (which is a non-negative integer) into a one-hot vector that we use as our training data. So we run one step of backpropagation with this reward vector given the state we gave the model. Since we're using a neural network model as our action-value function, we no longer have any sort of action-value array storing "memories," everything is being encoded in the neural network's weight parameters.

### **Listing 2.1 The Main Training Loop**

```
def train(env, epochs=5000, learning_rate=1e-2):
    cur_state = Variable(th.Tensor(one_hot(arms, env.get_state()))) #A
    optimizer = th.optim.Adam(model.parameters(), lr=learning_rate)
    for i in range(epochs):
        y_pred = model(cur_state) #B
        av_softmax = softmax(y_pred.data.numpy(), tau=2.0) #C
        av_softmax /= av_softmax.sum() #D
        choice = np.random.choice(arms, p=av_softmax) #E
        cur_reward = env.choose_arm(choice) #F
        one_hot_reward = y_pred.data.numpy().copy() #G
        one_hot_reward[choice] = cur_reward #H
        reward = Variable(th.Tensor(one_hot_reward))
        loss = loss_fn(y_pred, reward)

        optimizer.zero_grad()

        loss.backward()

        optimizer.step()
        cur_state = Variable(th.Tensor(one_hot(arms, env.get_state()))) #I
```

#A Get current state of the environment; convert to PyTorch Variable

#B Run neural net forward to get reward predictions

#C Convert reward predictions to probability distribution with softmax

#D Normalize distribution to make sure it sums to 1

#E Choose new action probabilistically

#F Take action, receive reward

```
#G Convert PyTorch tensor data to numpy array
#H Update one_hot_reward array to use as labeled training data
#I Update current environment state
```

When we train this network for 5000 epochs, we can plot the moving average of rewards earned over the training time (we omitted the code to produce such a graph). Our neural network indeed learns a fairly good understanding of the relationship between states, actions and rewards for this contextual bandit. The maximum reward payout for any play is 10, and our average is topping off around 8.5, which is close to the mathematical optimum for this particular bandit. Our first deep reinforcement learning algorithm works! Okay... it's not a very deep network, but still!

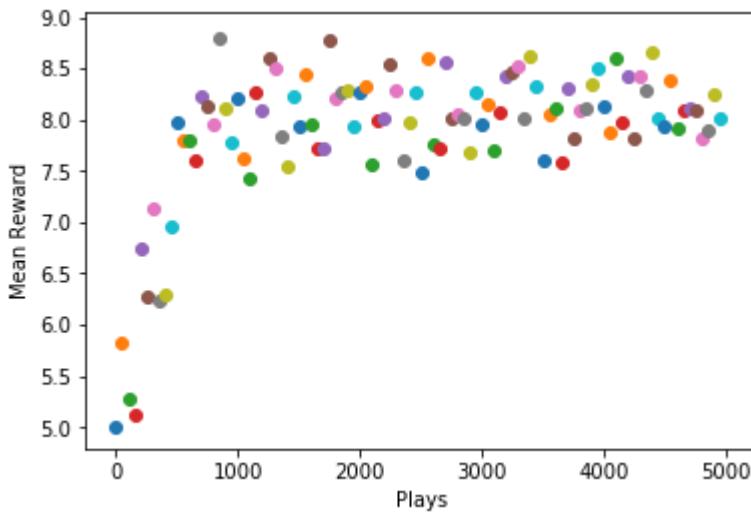


Figure 2.3: Training graph showing the average rewards for playing the contextual bandit simulator using a 2-layer neural network as the action-value function. We can see the average reward rapidly increases during training time, demonstrating our neural network is successfully learning.

## 2.6 The Markov Property

In our contextual bandit problem, our neural network led us to choose the best action given a state without reference to any other prior states. We just gave it the current state we're in and it produced the expected rewards for each possible action. This is an important property in reinforcement learning called the **Markov property**. A game (or any other control task) that exhibits the Markov property is said to be a **Markov Decision Process (MDP)**. With an MDP, the current state alone contains enough information to choose optimal actions to maximize future rewards. Modeling a control task as an MDP is a key concept in reinforcement learning.

The MDP model simplifies an RL problem dramatically as we do not need to take into account all previous states or actions, i.e. we don't need to have memory, we just need to analyze the present situation. Hence, we always attempt to model a problem as a (at least approximately) Markov decision processes. The card game Blackjack (also known as 21) is an MDP because we can play the game successfully by just knowing our current state (i.e. what cards we have and the dealer's one face-up card).

To test your understanding of the Markov property, consider each control problem/decision task in the list below and see if it has the Markov property or not:

- Driving a car
- Deciding whether to invest in a stock or not
- Choosing a medical treatment for a patient
- Diagnosing a patient's illness
- Predicting which team will win in a football game
- Choosing the shortest route (by distance) to some destination
- Aiming a gun to shoot a distant target

Okay, so let's see how you did, here are our answers and brief explanations:

- Driving a car can generally be considered to have the Markov property because you don't need to know what happened 10 minutes ago to be able to optimally drive your car. You just need to know where everything is right now and where you want to go.
- Deciding whether to invest in a stock or not does not meet the criteria of the Markov property since you would want to know the past performance of the stock in order to make a decision.
- Choosing a medical treatment seems to have the Markov property because you don't need to know the biography of a person to choose a good treatment for what ails them right now.
- In contrast, *diagnosing* (rather than treating) would definitely require knowledge of past states. It is often very important to know the historical course of a patient's symptoms in order to make a diagnosis.
- Predicting which football team will win does not have the Markov property since, like the stock example, you need to know the past performance of the football teams to make a good prediction.
- Choosing the shortest route to a destination has the Markov property because you just need to know the distance to the destination for various routes, which doesn't depend on what happened yesterday.
- Aiming a gun to shoot a distant target also has the Markov property since all you need to know is where the target is and perhaps current conditions like wind velocity and the particulars of your gun. You don't need to know the wind velocity of yesterday.

I hope you can appreciate that for some of those examples you could make arguments for or against it having the Markov property. For example, in diagnosing a patient, you may need

to know the recent history of their symptoms but if that is documented in their medical record and we consider the full medical record as our current state, then we've effectively induced the Markov property. This is an important thing to keep in mind: many problems may not naturally have the Markov property, but often times we can induce it by jamming more information into the state.

DeepMind's deep Q-learning algorithm learned to play Atari games from just raw pixel data and the current score. Do Atari games have Markov property? Not exactly. Say the game is Pacman, if our state is the raw pixel data from our current frame, we have no idea if that enemy a few tiles away is approaching us or moving away from us, and that would strongly influence our choice of actions to take. This is why DeepMind's implementation actually feeds in the last 4 frames of gameplay, effectively changing a non-Markov decision process into an MDP. With the last 4 frames, the agent has access to the direction and speed of all players.

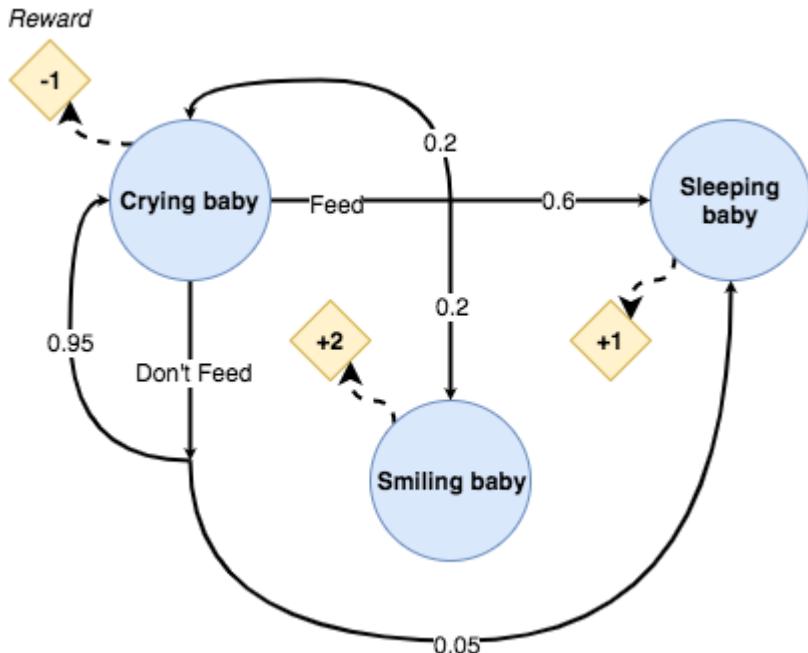


Figure 2.4: A simplified MDP diagram with 3 states and 2 actions. Here we model the parenting decision process for taking care of an infant. If the baby is crying, we can either administer food or not, and with some probability the baby will transition into a new state and receive a reward of -1, +1, or +2 (e.g. the baby's satisfaction).

Figure 2.2 gives a lighthearted example of a Markov decision process using all the concepts we've discussed so far. You can see there is a 3-element state space  $S = \{\text{crying baby}, \text{sleeping baby}, \text{smiling baby}\}$ , and a 2-element action space  $A = \{\text{feed}, \text{don't feed}\}$ . In

addition, we have the transition probabilities noted, which are the maps from an action to the probability of an outcome state (we'll go over this again in the next section). Of course in real life, you as the "agent" have no idea what the transition probabilities are. If you did, then you would have a *model* of the environment. As we'll learn later, sometimes an agent does have access to a model of the environment, and sometimes not. In the cases where our agent does not have access to the model, we may want our agent to learn a model of the environment (which may just approximate the true, underlying model).

## 2.7 Predicting Future Rewards: Value and Policy Functions

Believe it or not, we actually smuggled in a lot of knowledge in the previous sections. The way we setup our solutions to the N-armed bandit and the contextual bandit are standard reinforcement learning methods, and as such, there is a whole bunch of established terminology and mathematics behind what we did. We introduced a few terms already such as state and action spaces, but we mostly just described things in natural language. In order to give you the ability to understand the latest RL research papers and make future chapters less verbose, it's important to become acquainted with the jargon and the mathematics.

So let's review and formalize what we've learned so far. A reinforcement learning algorithm essentially constructs an agent which acts in some environment. The environment is often a game, but is more generally whatever process produces states, actions and rewards. The agent has access to the current state of the environment, which is all the data about the environment at a particular time point,  $s_t \in S$ . Using this state information, the agent takes an action  $a_t \in A$  which may deterministically or probabilistically change the environment to be in a new state  $s_{t+1}$ .

The probability associated with mapping a state to a new state by taking an action is called the **transition probability**. At the same time, the agent receives a reward  $r_t$  for having taken action  $a_t$  in state  $s_t$  leading to a new state  $s_{t+1}$ . And we know the ultimate goal of the agent (our reinforcement learning algorithm) is to maximize its rewards. It's really the state transition  $s_t \rightarrow s_{t+1}$  that produces the reward, not the action per se, since the action may probabilistically lead to a bad state. If you're in an action movie (no pun intended) and you jump off a roof onto another roof, whether you land gracefully onto the other roof or miss it completely and fall to your peril is what's important (the two possible resulting states), not the fact that you jumped (the action).

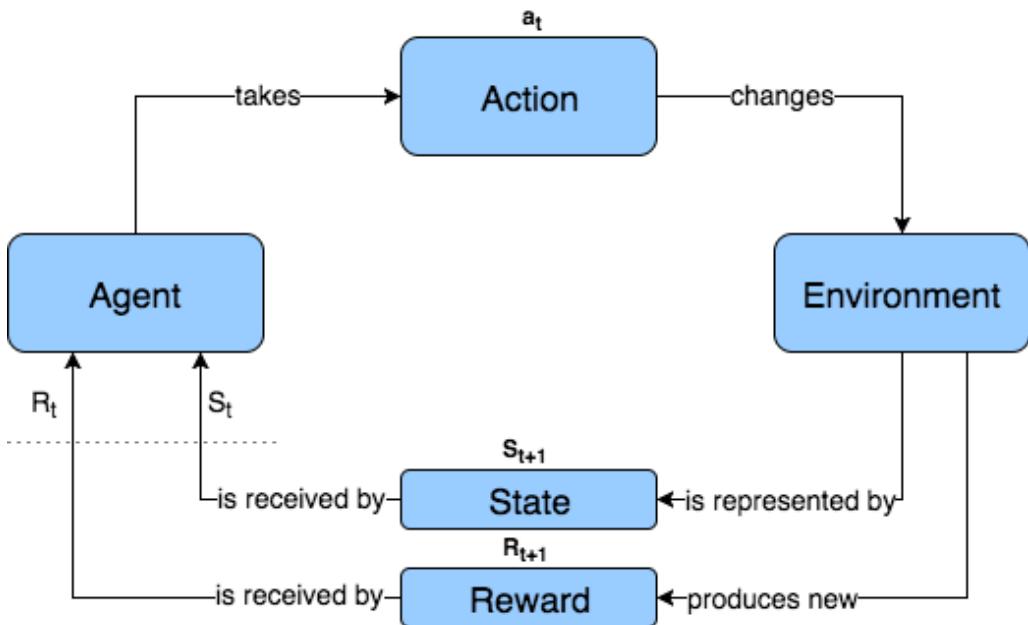


Figure 2.5: The general process of a reinforcement learning algorithm. The environment produces states and rewards. The agent takes an action  $a_t$  given a state  $s_t$  at time  $t$  and then receives a reward  $r_t$ . The agent's goal is to maximize rewards by learning to take the best actions in a given state.

**POLICY FUNCTIONS.** How exactly do we use our current state information to decide what action to take? This is where the key concepts of value functions and **policy functions** come into play, which we already have a bit of experience with. Let's first tackle policies. In words, a policy  $\pi$  is the “strategy” of an agent in some environment. For example, the strategy of the dealer in Blackjack is to always hit until she reaches a card value of 17 or greater.

#### Math

$$\pi: s \rightarrow \Pr(A | s), \text{ where } s \in S$$

#### English

A policy  $\pi$  is a mapping from states to the (probabilistically) best actions for those states.

Where  $s$  is a state and  $\Pr(A | s)$  is a probability distribution over the set of actions  $A$ , given state  $s$ . The probability of each action in the distribution is the probability that that action will produce the greatest reward.

**OPTIMAL POLICY** The policy is the part of our reinforcement learning algorithm that chooses actions given its current state. We can then formulate the *optimal policy*. The optimal policy is the strategy that maximizes rewards.

Math	English
$\pi^* = \operatorname{argmax} E[R   \pi]$	If we know the expected rewards for following any possible policy $\pi$ , then optimal policy $\pi^*$ is a policy that, when followed, produces the maximum possible rewards.

Remember, a particular policy is a map/function, so we have some sort of set of possible policies and the optimal policy is just an *argmax* (select the maximum) over this set of possible policies as a function of their expected rewards.

Again, the whole goal of a reinforcement learning algorithm (our agent) is to choose the actions that lead to the maximal expected rewards. But there are two ways we can train our agent to do this, directly or indirectly. That is, we can teach our agent to learn what actions are best directly given what state it is in, or we can teach it to learn which states are most valuable and then to take actions that lead to the most valuable states. This indirect method leads us to the idea of value functions.

**VALUE FUNCTIONS** are functions that map a state or a state-action pair to the **expected value** (i.e. expected reward) of being in some state or taking some action in some state. You may recall from statistics that the expected reward is just the long-term average of rewards received after being in some state or taking some action. When we speak of *the* value function, we usually mean a state-value function:

Math	English
$V_\pi: s \rightarrow E[R   s, \pi]$	A value function $V_\pi$ is a function that maps a state $s$ to the expected rewards given that we start in state $s$ and follow some policy $\pi$ .

This is a function that accepts a state  $s$  and returns the expected reward of starting in that state and taking actions according to our policy  $\pi$ . It may not be immediately obvious why the value function depends on the policy. Consider that in our contextual bandit problem, if our policy was just to choose totally random actions (i.e. sample actions from a uniform distribution), then the value (expected reward) of a state would probably be pretty low since we're definitely not choosing the best possible actions. We want to use a policy that is not a uniform distribution over the actions, but is the probability distribution that would produce the maximum rewards when sampled.

With our first N-armed bandit problem we were already introduced to state-action value functions. These functions often go by the name **Q-function** or **Q-value**, which is where Deep Q-learning comes from since, as we'll see in the next chapter, deep learning algorithms can be used as Q-functions.

Math	English
$Q_\pi(s, a) \rightarrow E[R   a, s, \pi]$	$Q_\pi$ is a function that maps a pair $(s, a)$ of a state $s$ and an action $a$ to the expected reward of taking action $a$ in state $s$ given that we're using the policy (i.e. 'strategy') $\pi$

In fact, we sort of implemented a deep Q-network to solve our contextual bandit problem (except it was a pretty shallow neural network) since it was essentially acting as a Q-function. We trained it to produce accurate estimates of the expected reward of taking an action given a state. Our policy function was the softmax function over the output of the neural network.

## 2.8 Chapter Summary

We will be using the tools and terminology we developed in this chapter throughout the rest of the book so make sure you're comfortable with them before moving on.

- **State-spaces and Action-spaces:** State-spaces are the set of all possible states a system can be in. In Chess, this would be the set of all possible valid board configurations. An action is a function that maps a state  $s$  to a new state  $s'$ . An action may be stochastic such that it maps a state sprobabilistically to a new state  $s'$ . There may be some probability distribution over the set of possible new states from which one is selected. The action-space is the set of all possible actions for a particular state.
- **Environment and Model:** The source of states, actions, and rewards. If we're building an RL algorithm to play a game, then the game is the environment. A model of an environment is an approximation of the state-space, action-space, and transition probabilities.
- **Reward and Expected Reward:** Rewards are signals produced by the environment that indicate the relative success of taking an action in a given state. Expected reward is a statistical concept that informally refers to the long-term average value of some random variable  $X$  (in our case, the reward), denoted  $E[X]$ . For example, in the  $N$ -armed bandit case,  $E[R | a]$  ('the expected reward given action  $a$ ') is the long-term average reward of taking each of the  $N$ -actions. If we knew the probability distribution over the actions  $a$  then we could calculate the precise value the expected reward for a game of  $N$  plays as  $E[R | a] = \sum_{i=1}^N a_i p_i r$  where  $N$  is the number of plays of the game,  $p_i$  refers to the probability of action  $a_i$ , and  $r$  refers to the maximum possible reward.
- **Agent:** An RL algorithm that learns to behave optimally in a given environment. Often implemented as a deep neural network. The goal of the agent is to maximize expected rewards, or equivalently, navigate to the highest value state.
- **Policy / Policy Function:** Informally, a particular strategy. Formally, a function that accepts a state and produces an action to take or produces a probability distribution

over the action-space given the state. A common policy is the epsilon-greedy strategy where with probability  $\epsilon$  we take a random action in the action-space, and with probability  $\epsilon-1$  we choose the best action we know of so far.

- **Value functions:** In general, any function that returns expected rewards given some relevant data. Without additional context, it typically refers to a state-value function, which is a function that accepts a state and returns the expected reward of starting in that state and acting according to some policy. The Q-value is the expected reward given a state-action pair, and the Q-function is a function that produces Q-values when given a state-action pair.
- **Markov Decision Process:** A decision-making process where it is possible to make the best decisions without reference to a history of prior states.

## 2.9 What's next?

We covered many of the foundational concepts in reinforcement learning just by using N-armed and contextual bandits as examples. We also got our feet wet with deep reinforcement learning in this chapter, but in the next chapter we will implement a full-blown Deep Q-network similar to the algorithm DeepMind used to play Atari games at superhuman levels. It will be a natural extension of what we've covered here.

# 3

## *Predicting the Best States and Actions: Deep Q-Networks*

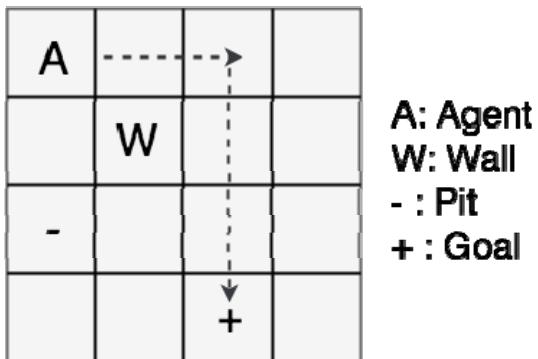
### This chapter covers:

- Implementing the Q-function as a neural network
- Building a Deep Q-network using PyTorch to play Gridworld
- Counteracting “catastrophic forgetting” with experience replay
- Improving learning stability with target networks

### 3.1 The Q-function

In this chapter we start off where the deep reinforcement learning revolution began: DeepMind’s Deep Q-networks that learned to play Atari games. Although we won’t be using Atari games as our testbed quite yet, we will be building virtually the same system DeepMind did. We’ll be using a simple console-based game called Gridworld as our game environment. Gridworld is actually a family of similar games, but they all generally involve a grid board with a player (or agent), an objective tile (the “goal”), and possibly one or more special tiles that may be barriers or may grant negative or positive rewards. The player can move up, down, left or right and the point of the game is to get the player to the goal tile where the player will receive a positive reward. The player must not only reach the goal tile but must do so following the shortest path and may need to navigate through various obstacles.

We will be using a very simple Gridworld engine included in the GitHub repository for this book, which you can download at <<https://github.com/DeepReinforcementLearning/>> in the Chapter 3 folder.



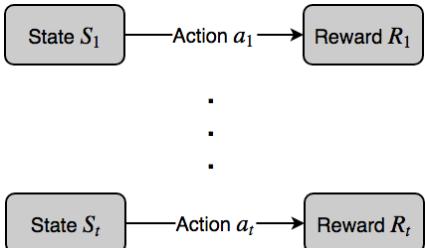
**Figure 3.1:** This is a simple Gridworld game setup. The agent (A) must navigate along the shortest path to the Goal (+) tile and avoid falling into the Pit (-).

The Gridworld game depicted in Figure 3.1 shows the simplest version of Gridworld we'll start with, and we'll progressively tackle more difficult variants of the game. Our goal is to train a deep RL agent to navigate the Gridworld board to the goal following the most efficient route every time. But before we get too far into that, let's review the key terms and concepts from the previous chapter that we will continue to use here.

The **state** is the information that our agent receives to make a decision on what action to take. It could be the raw pixels of a video game, sensor data from your autonomous vehicle, or, in the case of Gridworld, a tensor representing the position of all the objects on the grid.

The **policy**, denoted  $\pi$ , is the strategy our agent follows when provided a state. For example, a policy in blackjack might be to look at our hand (the state) and hit or stay randomly. Although this would be a terrible policy, the important point to stress is that the policy confers which actions we take. A better policy would be to always hit until we have 19.

The **reward** is the feedback our agent gets after taking an action, leading us to a new state. For a game of chess, we could reward our agent +1 when it performs the action that leads to a checkmate of the other player and -1 for an action that leads our agent to be checkmated. Every other state could be rewarded 0 since we do not know if the agent is winning or not.



Our agent makes a series of actions based upon its policy  $\pi$  and repeats this process until the episode ends. We call the weighted sum of the rewards while following a policy from the starting state  $S_1$  the **value** of that state, or a state-value. We can denote this by the **value function**  $V_\pi(s)$  that accepts an initial state and returns the expect total reward.

$$V_\pi(s) = \sum_{i=1}^t w_i R_i = w_1 R_1 + w_2 R_2 + \dots + w_t R_t$$

The coefficients  $w_1, w_2$ , etc. are the weights we apply to the rewards before summing them. For example, we often want to weight more recent rewards greater than distant future rewards. This weighted sum is an expected value, a common statistic in many quantitative fields, and is often concisely denoted  $E[R|\pi,s]$ , read as “the expected rewards given a policy  $\pi$  and a starting state  $s$ ”. Similarly, there is an **action-value function**  $Q_\pi(s, a)$  that accepts a state  $S$  and an action  $A$  and returns the value of taking that action given that state, or in other words  $E[R + | \pi, s, a]$ . Some RL algorithms or implementations will use one or the other. Importantly, if we base our algorithm on learning state-values (as opposed to action-values), we must keep in mind that the value of a state depends completely on our policy  $\pi$ . Using blackjack as an example, if we’re in the state of having a card total of 20, and have two possible actions, hit or stay, the value of this state is only high if our policy says to stay when we have 20. If our policy said to hit when we have 20, we would probably bust and lose the game, thus the value of that state would be low. In other words, the value of a state is equivalent to the value of the highest action taken in that state.

## 3.2 Navigating with Q-learning

In 2013, DeepMind published a paper entitled “Playing Atari with Deep Reinforcement Learning” that outlined their new approach to an old algorithm, which gave them enough performance to play 6 of 7 Atari 2600 games at record levels. Crucially, the algorithm they used only relied on analyzing the raw pixel data from the games just like a human. This paper really set off the field of deep reinforcement learning.

The old algorithm they modified is called Q-learning and it has been around for decades. Why did it take so long to make such significant progress? Well of course a large part is due to the general boost that artificial neural networks (deep learning) got a few years prior with the use of GPUs that allowed training much larger networks. But a significant amount is due to the specific novel features DeepMind implemented to address some of the other issues that reinforcement learning algorithms struggled with. We'll be covering it all in this chapter.

**WHAT IS Q-LEARNING.** Okay so what is Q-learning, you ask? If you guessed it has something to do with the action-value function  $Q_\pi(s, a)$  we previously described, you are right, but it's only a small part of the story. Many RL algorithms involve updating a value function (whether it's a state value or action-value) to achieve the objective, but not all such algorithms are Q-learning algorithms. Q-learning is a particular method to learn optimal action-values, but there are others. That is to say, value-functions and action-value functions are general concepts in RL that appear in many places, but Q-learning is a particular algorithm. Just like the goal of any investor is to make money, but there are many different kinds of investment strategies even though there are concepts common to all of them.

Believe it or not, we sort of implemented a Q-learning algorithm in the last chapter when we built a neural network to optimize the ad placement problem. The main idea of Q-learning is that your algorithm predicts the value of a state-action pair, and then you compare this prediction to the observed accumulated rewards at some later time point and then update the parameters of your algorithm so that next time it will make better predictions. That's essentially what we did last chapter when we our neural network predicted the expected reward (value) of each action given a state, and then we would observe the actual reward we get and update our network accordingly. But that was a particular and simple implementation of a broader class of Q-learning algorithms that is described by the following update rule:

#### Math

---

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

#### Pseudocode

---

```
def get_updated_q_value(old_q_value, reward, state, step_size, discount):
    term2 = (reward + discount * max([Q(state, action) for action in actions]))
    term2 = term2 - old_q_value
    term2 = step_size * term2
    return (old_q_value + term2)
```

## English

---

The Q-value at time  $t$  is updated to be the current predicted Q-value plus the amount of value we expect in the future given that we play optimally from our current state.

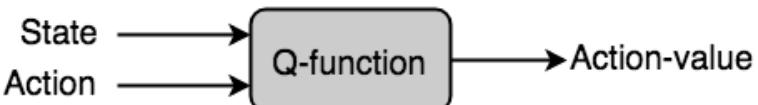
### TACKLING GRIDWORLD.

Okay, so we've seen the formula for Q-learning. Let's take a step back and apply this formula to our Gridworld problem. Our goal in this chapter is to train a neural network to play a simple game, Gridworld, from scratch. All the agent will have access to is what the board looks like, the same as a human player would; the algorithm has no informational advantage. Moreover, we're starting with an un-trained algorithm, so it literally knows nothing at all about the world. It has no prior information about how games work or anything. The only thing we provided is the reward for reaching the goal. The fact that we will (trust us) be able to teach the algorithm to learn to play from nothing is actually quite impressive.

Unlike us humans who live in what appears to be a continuous flow of time, the algorithm lives in a discrete world, so something needs to happen at each discrete time step. First, at time step 1, the algorithm will "look" at the game board and make a decision about what action to take, then the game board will be updated, and so on.

Let's sketch out the details of this process now. Here's the sequence of events for a game of Gridworld.

1. We start the game in some state, we'll call it  $S_t$ . The state includes all the information about the game that we have. For our Gridworld example, the game state is represented as a 4x4x4 tensor. We will go into more detail about the specifics of the board when we implement the algorithm in a later section.
2. We feed the  $S_t$  data and a candidate action into a deep neural network (or some other fancy machine learning algorithm) and it produces a prediction of how valuable taking that action in that state is.



Remember, the algorithm is not predicting the reward we will get after taking a particular action, it's predicting the expected value or expected rewards, which is the long-term average reward we will get from taking an action in a state and then continuing to behave according to our policy  $\pi$ . We do this for a bunch (perhaps all) possible actions we could take in this state.

3. We decide to take an action, which we'll label  $A_t$ , e.g. because our neural network predicted it is the highest value action. After we take the action, we are now in a new state of the game, which we'll call  $S_{t+1}$ , and we receive or observe a reward, labelled  $R_{t+1}$ . Now we want to update our learning algorithm to reflect the actual reward we

received after taking the action it predicted was the best. Perhaps we got a negative reward or a really big reward, so we want to improve the accuracy of its predictions.

4. Next we run the algorithm using  $S_{t+1}$  as input and figure out which action our algorithm predicts has the highest value, we'll call this  $\max Q(S_{t+1}, a)$ . To be clear, this is a single value that reflects the highest predicted q-value given our new state and all possible actions.
5. Now we have all the pieces of our update term. We'll perform one iteration of training using some loss function, such as mean-squared error, to minimize the prediction error of our algorithm and the target prediction of  $Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * \max Q(S_{t+1}, A) - Q(S_t, A_t)]$ .

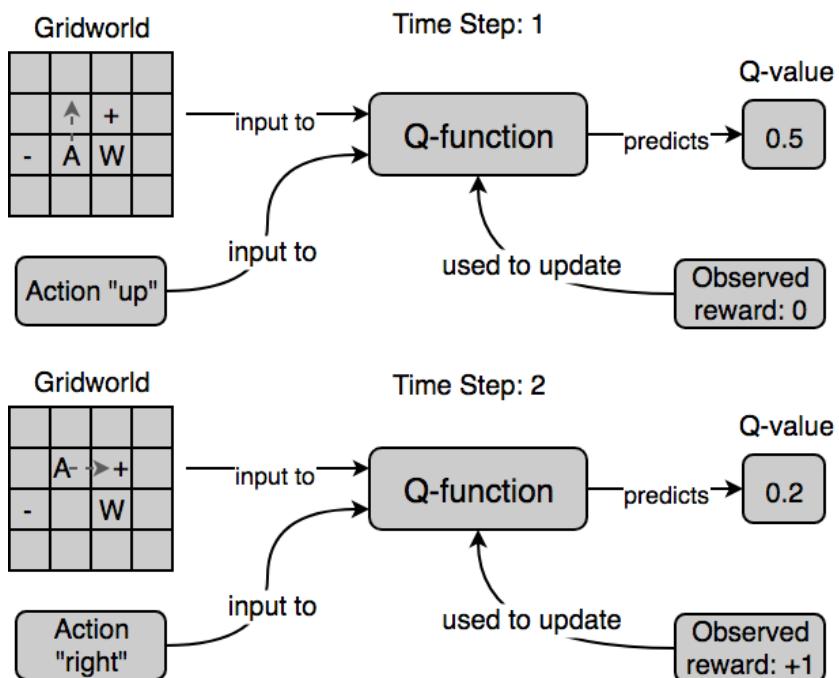


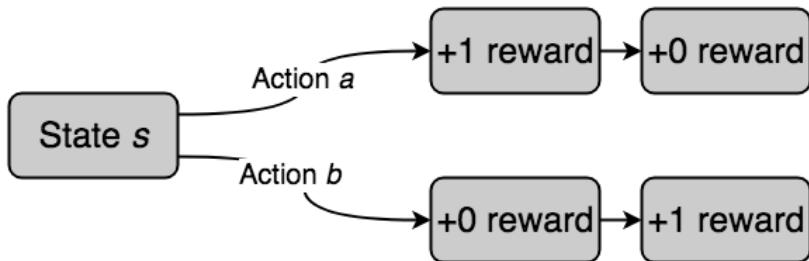
Figure 3.2: Schematic of Q-learning with Gridworld. The Q-function accepts a state and an action, and returns the predicted expected reward (i.e. value) of that state-action pair. After taking the action, we observe a reward, and using the update formula shown earlier, we use this observation to update the Q-function to make better predictions.

## HYPERPARAMETERS.

The parameters and are called hyperparameters since they're parameters that influence how the algorithm learns but are not involved in the actual learning. The parameter is the **learning-rate** and is the same hyperparameter used to train many machine learning

algorithms. It controls how quickly we want the algorithm to learn from each move; a small value means it will only make small updates at each step whereas a large value means the algorithm will potentially make large updates.

**DISCOUNT FACTOR.** The parameter  $\gamma$ , the **discount factor**, is a variable between 0 and 1 and controls how much our agent discounts future rewards when making a decision. Let's take a simple example. Our agent has a decision between picking an action that leads to 0 reward then +1 reward or an action that leads to +1 and then 0 rewards.



Previously, we defined the value of a trajectory as the expected reward. Both trajectories provide +1 overall reward though, which sequence of actions should the algorithm prefer, or in other words, how can we break the tie? Well if the discount factor  $\gamma$  is less than 1, we will discount future rewards relative to recent rewards. In this simple case, even though both paths lead to a total of +1 rewards, action  $b$  gets the +1 reward later than action  $a$ , and since we will discount the future action, we prefer action  $a$ . In other words, we multiply the +1 reward observed with action  $b$  by a weighting factor less than 1, so we lower the reward from +1 to say only 0.8, hence now the decision is clear which action to take.

The discount factor comes up in real life as well as RL. Let's say that someone offers you \$100 now or \$110 one month from now. Most people would prefer to receive the money now because we discount the future to some degree, which makes sense since the future is someone uncertain (e.g. what if the person offering you the money dies in 2 weeks?).

In Q-learning, we face the same decision; how much do we consider future observed rewards when learning to predict Q-values? Unfortunately, there's no definitive answer to this, or to setting pretty much any of the hyperparameters we have control over. We just have to play around with these knobs and see what works best empirically.

It's worth pointing out that most games are *episodic*, meaning that there's multiple chances to take actions before the game is over, and many games like Chess don't naturally assign points to anything other than winning or losing the game. Hence the reward signal in these games is sparse, making it difficult for trial-and-error based learning to reliably learn anything, which would require seeing a reward fairly frequently. In Gridworld, we've designed the game so that any move that doesn't win the game receives a reward of 0, the winning move gets a reward of +1, and the losing move rewards -1. So it's really only the final move of the game where the algorithm can say "aha! Now I get it!" Since each episode of a

Gridworld game can be won in a fairly small number of moves, the sparse reward problem isn't too bad, but in other games it is such a significant problem that even the most advanced reinforcement learning algorithms have yet to reach human level performance. While beyond the scope of this book, one proposed method of dealing with this is to stop merely relying on the objective of maximizing expected rewards and instead instruct the algorithm to seek novelty and by doing so it will learn about its environment.

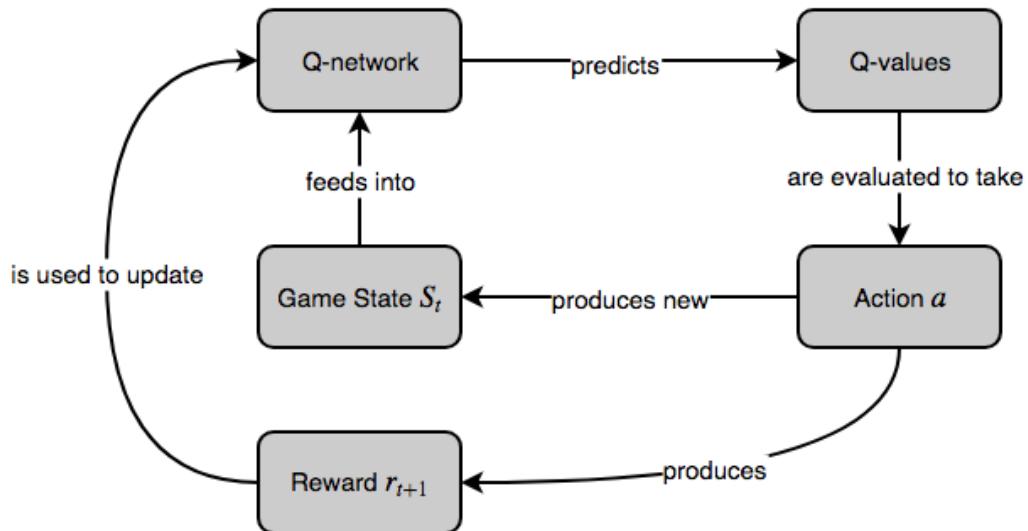


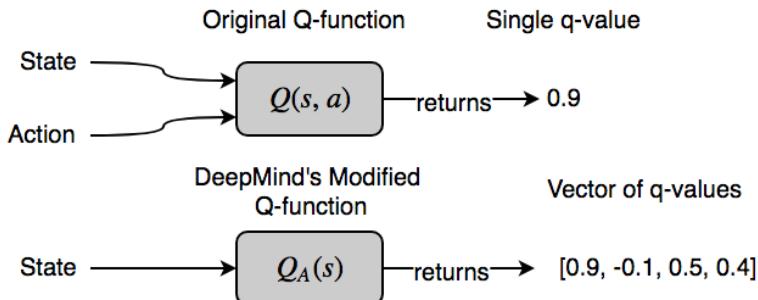
Figure 3.3: The general overview of Q-learning. We start in a game state  $S_t$ , which the Q-network (i.e. a neural network acting as a Q-function) uses to predict Q-values for the 4 available actions in Gridworld. The Q-values are used to select an action (e.g. we might just select the action associated with the highest predicted Q-value), which then results in a new game state  $S_{t+1}$  and a reward signal  $r_{t+1}$ . The reward is used to update the Q-network but minimizing the loss between the predicted Q-values and the observed reward.

**BUILDING THE NETWORK** Let's dig into how we will build our deep learning algorithm for this game. Recall that a neural network has a particular kind of architecture or network topology. When you build a neural network, you have to decide how many layers it should have, how many parameters each layer (i.e. the "width" of the layer) has, and how the layers are connected. Gridworld is simple enough that we don't need to build anything fancy. We can get away with a fairly straightforward feedforward neural network with only a few layers, using the typical rectified linear activation unit (ReLU).

The only parts that require some more careful thought are how we will represent our input data and how we will represent the output layer. We'll cover the output layer first. In our discussion of Q-learning, we said that the Q-function is a function that takes some state and some action and computes the value of that state-action pair, i.e.  $Q(s, a)$ . This is how the Q-

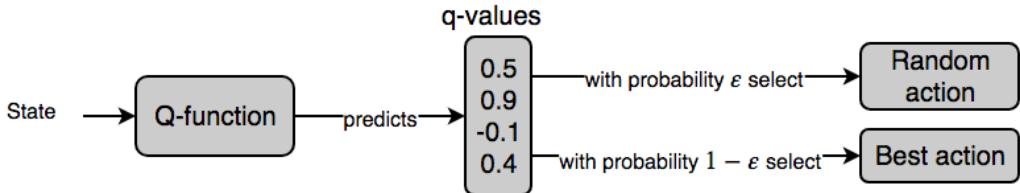
function was originally defined. As we noted in the previous chapter, there is also just a state-value function usually denoted  $V_\pi(s)$  that computes the value of some state given you're following a particular policy  $\pi$ .

Generally, we want to use the Q-function because it can tell us the value of taking an action in some state, and therefore we can take the action that has the highest predicted value. But it would be rather wasteful if we separately computed the Q-values for every possible action given the state, even though the Q-function was originally defined that way. A much more efficient procedure, and the one that DeepMind employed in its implementation of Deep Q-Learning, is to instead recast the Q-function as a vector-valued function, meaning that instead of computing and returning a single Q-value for a single state-action pair, it will compute the Q-values for all actions given some state and return the vector of all those Q-values. So now we might represent this new version of the Q-function,  $Q_A(s)$ , where the subscript A denotes the set of all possible actions.



**Figure 3.4:** The original Q-function accepts a state-action pair and returns the value of that state-action pair, i.e. a single number. DeepMind used a modified vector-valued Q-function that accepts a state and returns a vector of state-action values, one for each possible action given the input state. The vector-valued Q-function is more efficient since you only need to compute the function once for all the actions.

Now it's quite easy to employ a neural network as our version of the Q-function; the last layer will simply produce an output vector of Q-values, one for each possible action. In the case of Gridworld there are only 4 possible actions ("up", "down", "left", "right") so the output layer will produce 4-dimensional vectors. We can then directly use the output of the neural network to decide what action to take using a simple -greedy approach or softmax. Just like DeepMind, we'll use the -greedy approach, and instead of using a static value like we did in the last chapter, we will initialize it to a large value (i.e. 1, so completely random selection of actions to start) and slowly decrement it so that after a certain number of iterations, the value will rest at some small value. In this way, we will allow the algorithm to explore and learn a lot in the beginning but then it will settle into maximizing rewards by exploiting what it has learned. Hopefully we set the decrementing process so that it will not under-explore or over-explore, but that will have to be tested empirically.



**Figure 3.5:** In an epsilon-greedy action selection method, we set a parameter  $\epsilon$  to some value, e.g. 0.1, and with probability  $\epsilon = 0.1$  we will randomly select an action (completely ignoring the predicted q-values) or with probability  $1 - \epsilon = 0.9$ , we select the action associated with the highest predicted q-value (the best known action). An additional helpful technique is to start with a high  $\epsilon$  value, e.g. 1 and then slowly decrement it over the training iterations.

Okay, we have the output layer figured out, now to tackle the rest. In this book, we construct a network of just three layers with widths of 164 (input layer), 150 (hidden layer), 4 (output layer). You are welcome and encouraged to add more hidden layers and/or play with the size of the hidden layer as you will likely be able to achieve better results with a deeper network. We chose to implement a fairly shallow network here so that you can train the model with just your CPU (it takes our MacBook Air 1.7 GHz Intel Core i7 8GB RAM only a few minutes to train). We already discussed why the output layer is of width 4, but we haven't talked about the input layer yet. We need to first introduce the Gridworld game engine we will be using. We specifically developed a Gridworld game for this book and it is included in the GitHub repository for this chapter < <http://github.com/DeepReinforcementLearning/> >.

**INTRODUCING THE GRIDWORLD GAME ENGINE.** In the GitHub repository, you'll find a single file, `Gridworld.py`. Just copy and paste this file into whatever folder you'll be working out of. You can include it in your Python session by just running `from Gridworld import *`. The `Gridworld` module contains some classes and helper functions to run a Gridworld game instance. To create a Gridworld game instance, run:

### Listing 3.1 Creating a Gridworld Game

```
from Gridworld import *
game = Gridworld(size=4, mode='static')
```

The Gridworld board is always square, so the size refers to one side dimension, i.e. in this case a 4x4 grid will be created. There are three ways to initialize the board. The first is to initialize it statically, so that the objects on the board are initialized at the same pre-determined locations. Second, you can set `mode='player'` so that just the player is initialized at a random position on the board. Lastly, you can also initialize so that all the objects are placed randomly (which is harder to learn for the algorithm) using `mode='random'`. We'll use all 3 eventually.

Now that we've created the game, let's play it. You call the `display` method to display the board, and the `makeMove` method to make a move. Moves are encoded with a single letter 'u'

for 'up', 'l' for 'left' and so on. After each move, you should display the board to see the effect. Additionally, after each move, you'll want to observe the reward/outcome of the move by calling the `reward` method. In Gridworld, every non-winning move receives a 0 reward. The winning move (reaching the goal) receives a +1 reward, or a -1 reward for the losing move (landing on the pit).

```
>>> game.display()
array([[ '+', ' -', ' ', 'P'],
       [ ' ', 'W', ' ', ' '],
       [ ' ', ' ', ' ', ' '],
       [ ' ', ' ', ' ', ' ']], dtype='<U2')

>>> game.makeMove('d')
>>> game.makeMove('d')
>>> game.makeMove('l')
>>> game.display()
array([[ '+', ' -', ' ', ' '],
       [ ' ', 'W', ' ', ' '],
       [ ' ', ' ', 'P', ' '],
       [ ' ', ' ', ' ', ' ']], dtype='<U2')
>>> game.reward()
0
```

Now let's take a look at how the game state is actually represented since we will need to feed this into our neural network. Run:

```
>>> game.board.render_np()
array([[[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 0]],

      [[1, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]],

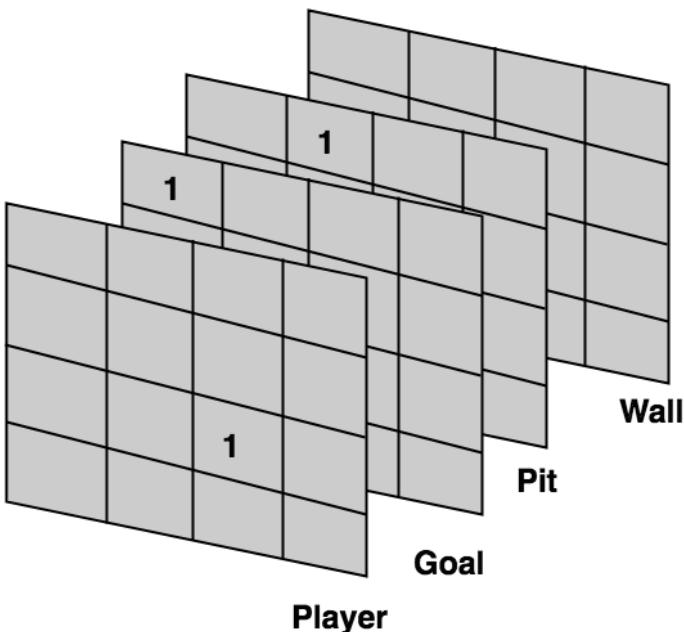
      [[0, 1, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]],

      [[0, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]]], dtype=uint8)>>> game.board.render_np().shape
(4, 4, 4)
```

Each matrix is a 4x4 grid of zeros and a single 1 where a 1 indicates the position of a particular object. Each matrix encodes the position of one of the 4 objects: the player, the goal, the pit and the wall. If you compare the result from `display` with the game state, you can see that the first matrix encodes the position of the player, the second matrix encodes position

of the goal, the third matrix encodes the position of the pit, and the last matrix encodes the position of the wall.

In other words, the 1st dimension of this 3-tensor is divided into 4 separate grid planes, where each plane represents the position of each element. So below is an example where the player is at grid position (2,2), the goal is at (0,0), the pit is at (0,1) and the wall is at (1,1) where (row, column). All other elements are 0s.



**Figure 3.6:** This is how the Gridworld board is represented as a numpy array. It is a  $4 \times 4 \times 4$  tensor, composed of 4 “slices” of a  $4 \times 4$  grid. Each grid slice represents the position of an individual object on the board and contains a single 1 whereas all other elements are 0s. The position of the 1 indicates the position of that slice’s object.

While we could in principle build a neural network that can operate on a  $4 \times 4 \times 4$  tensor, it is easier to just flatten it into a 1-tensor (vector). A  $4 \times 4 \times 4$  tensor has  $4^3 = 64$  total elements, so the input layer of our neural network must be accordingly shaped. The neural network will have to learn what this data means and how it relates to maximizing rewards. Remember, the algorithm will know absolutely nothing to begin with.

**NEURAL NETWORK AS THE Q-FUNCTION.** Now for the fun part. Let’s build our neural network that will serve as our Q function. As you know, in this book we’ll be using PyTorch for all our deep learning models, but if you’re more comfortable with another framework such as TensorFlow or MXnet, then it should be fairly straightforward to port the models. Here’s the general architecture for the model we will build:

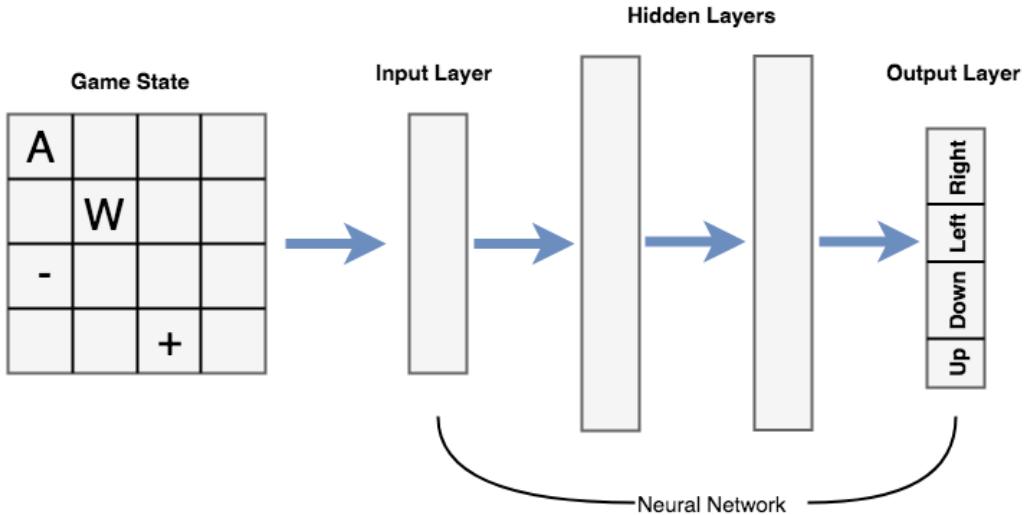


Figure 3.7: The neural network model we will use to play Gridworld. The model has an input layer that can accept a 64-length game state vector, some hidden layers, and an output layer that produces a 4-length vector of Q-values for each action, given the state.

To implement this with PyTorch, we'll use the `nn` module, which is the higher-level interface for PyTorch, similar to Keras for TensorFlow.

### **Listing 3.2 Neural Network Q-function**

```
import numpy as np
import torch
from torch.autograd import Variable
from Gridworld import *
from IPython.display import clear_output
import random
from matplotlib import pylab as plt

l1 = 64
l2 = 150
l3 = 100
l4 = 4

model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3,l4)
)
loss_fn = torch.nn.MSELoss(size_average=True)
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
gamma = 0.9
epsilon = 1.0
```

So far all we've done is setup the neural network model, define a loss function, the learning rate, setup an optimizer, and define a couple parameters. If this were a simple classification neural network, then we'd almost be done. We'd just setup a for-loop to iteratively run the optimizer to minimize the model error with respect to the data. It's more complicated with reinforcement learning, which is probably why you're reading this book. We covered the main steps we'd need to run through earlier, but let's zoom in a little.

Below is the implementation for the main loop of the algorithm. In broad strokes:

1. Setup a for-loop to number of epochs
2. In the loop, setup while loop (while game is in progress)
3. Run Q network forward.
4. We're using an epsilon greedy implementation, so at time  $t$  with probability  $\epsilon$  we will choose a random action. With probability  $1-\epsilon$  we will choose the action associated with the highest Q value from our neural network.
5. Take action  $a$  as determined in (4), observe new state  $s'$  and reward  $r_{t+1}$
6. Run the network forward using  $s'$ . Store the highest Q value,  $\max Q$ .
7. Our target value to train the network is  $R_{t+1} + \gamma * \max Q_A(S_{t+1})$  where  $\gamma$  (gamma) is a parameter between 0 and 1.
8. Given that we have 4 outputs and we only want to update/train the output associated with the action we just took, our target output vector is the same as the output vector from the first run, except we change the one output associated with our action to the result we compute using the Q-learning formula.
9. Train the model on this 1 sample. Repeat process 2-9

[insert blog diagram of steps 1-9]

Just to be clear, when we first run our neural network and get an output of action-values like this

```
array([[-0.02812552, -0.04649779, -0.08819015, -0.00723661]])
```

our target vector for one iteration may look like this:

```
array([[-0.02812552, -0.04649779, 1, -0.00723661]])
```

where we just changed a single entry to the value we want to update. One detail to

One detail to include in the code before we move on. The Gridworld game engine's makeMove method expects a character such as 'u' to make a move, however, our Q-learning algorithm only knows how to generate numbers, so we need a simple map from numeric keys to our action characters:

```
action_set = {
    0: 'u',
```

```

1: 'd',
2: 'l',
3: 'r',
}

```

Okay, let's actually get to coding the main training loop.

### **Listing 3.3 Q-learning: Main Training Loop**

```

epochs = 1000
losses = [] #A
for i in range(epochs): #B
    game = Gridworld(size=4, mode='static') #C
    state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0 #D
    state = Variable(torch.from_numpy(state_).float()) #E
    status = 1 #F
    while(status == 1): #G
        qval = model(state) #H
        qval_ = qval.data.numpy()
        if (random.random() < epsilon): #I
            action_ = np.random.randint(0,4)
        else:
            action_ = (np.argmax(qval_))

        action = action_set[action_] #J
        game.makeMove(action) #K
        new_state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
        new_state = Variable(th.from_numpy(new_state_).float()) #L
        reward = game.reward()
        newQ = model(new_state.reshape(1,64)).data.numpy()
        maxQ = np.max(newQ) #M
        y = np.zeros((1,4))
        y[:] = qval_[:]
        if reward == -1: #N
            update = (reward + (gamma * maxQ))
        else:
            update = reward
        y[0][action_] = update #O
        y = Variable(torch.from_numpy(y).float())
        loss = loss_fn(qval, y) #P
        print(i, loss.item())
        optimizer.zero_grad()
        loss.backward()
        losses.append(loss.data[0])
        optimizer.step()
        state = new_state
        if reward != -1: #Q
            status = 0
            clear_output(wait=True)
    if epsilon > 0.1: #R
        epsilon -= (1/epochs)

```

#A Create a list to store loss values so we can plot the trend later

#B The main training loop

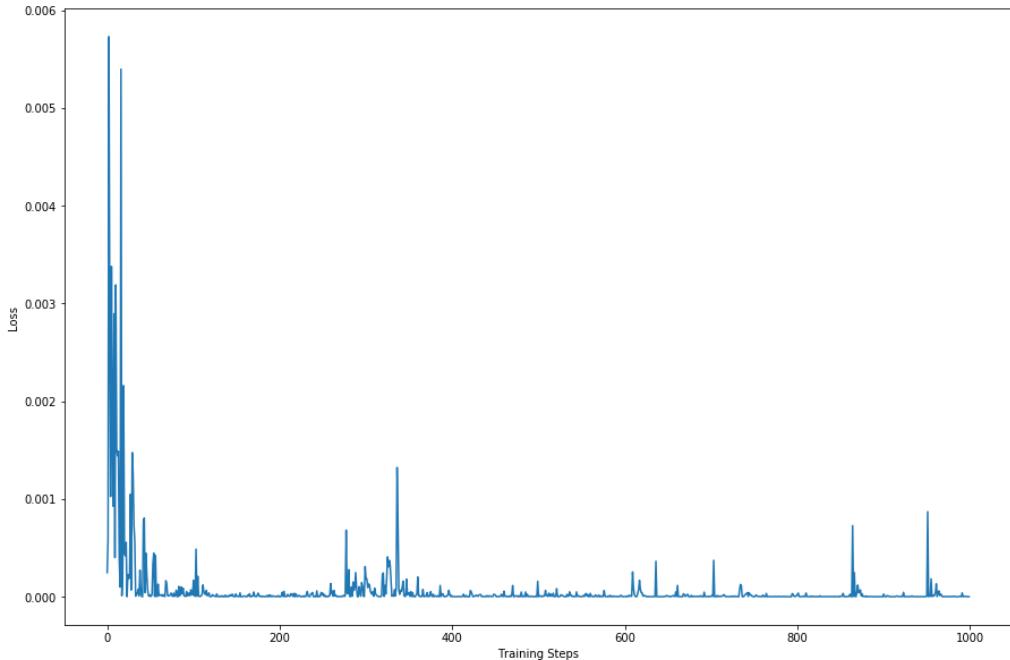
```

#C For each epoch, we start a new game
#D After we create the game, we extract the state information and add a small amount of noise
#E We need to convert the numpy array into a PyTorch Tensor and then into a PyTorch Variable
#F Use the status variable to keep track of whether or not the game is still in progress
#G While this game is still in progress, we'll play to completion then start a new epoch
#H Run the Q-network forward to get its predicted q-values for all the actions
#I Select an action using the epsilon-greedy method
#J We have to translate the numerical action into one of the action characters that our Gridworld game expects
#K After selecting an action using the epsilon greedy method, we take the action
#L After making a move, we get the new state of the game
#M Find the maximum q-value predicted from the new state
#N Calculate the target q-value
#O Create a copy of the qval array and then update the one element corresponding to the action we took
#Q If reward is -1 then we know the game hasn't been won or lost and is still in progress
#R Decrement the epsilon value each epoch

```

**NOTE:** Why did we add noise to the game state? Because it helps prevent “dead neurons” that can happen with the use of rectified linear units (ReLU) as our activation function. Basically, since most of the elements in our game state array are 0s they won’t play nice with ReLU, which is technically non-differentiable at 0. Hence, we add just a tiny bit of noise so that none of the values in the state array are exactly 0. It also might help with overfitting.

Go ahead and run the training loop, 1000 epochs will be enough. Once it’s done, you can plot the losses to see if the training looks successful (i.e. the loss more or less decreases over training time). Here’s the plot we get:



The loss plot is pretty noisy but clearly a moving average of the plot is significantly trending toward zero. This gives us some confidence the training worked, but we'll never know until we test it. We've written up a simple function that allows us to test the model on a single game.

#### **Listing 3.4 Testing the Q-network**

```
def test_model(model, mode='static'):
    i = 0
    test_game = Gridworld(mode=mode)
    state_ = test_game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
    state = Variable(torch.from_numpy(state_).float())
    print("Initial State:")
    print(test_game.display())
    status = 1
    #while game still in progress
    while(status == 1):
        qval = model(state)
        qval_ = qval.data.numpy()
        action_ = np.argmax(qval_) #take action with highest Q-value
        action = action_set[action_]
        print('Move #: %s; Taking action: %s' % (i, action))
        test_game.makeMove(action)
        state_ = test_game.board.render_np().reshape(1,64) +
        np.random.rand(1,64)/10.0
        state = Variable(torch.from_numpy(state_).float())
        print(test_game.display())
        reward = test_game.reward()
```

```

if reward != 0:
    status = 0
    print("Reward: %s" % (reward,))
i += 1
if (i > 15):
    print("Game lost; too many moves.")
    break

```

The test function is essentially just the same as the code in our training loop except we don't do any loss calculation or backpropagation, we're just running the network forward to get the predictions. Let's see if it learned how to play Gridworld!

```

>>> test_model(model, 'static')

Initial State:
[[ '+' '-' ' ' 'P' ]
 [ ' ' 'W' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 0; Taking action: d
[[ '+' '-' ' ' ' ' ' ]
 [ ' ' 'W' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 1; Taking action: d
[[ '+' '-' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' ]
 [ ' ' ' ' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 2; Taking action: l
[[ '+' '-' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' ]
 [ ' ' ' ' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 3; Taking action: l
[[ '+' '-' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' ]
 [ ' ' ' ' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 4; Taking action: l
[[ '+' '-' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' ]
 [ ' ' 'P' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 5; Taking action: u
[[ '+' '-' ' ' ' ' ' ]
 [ 'P' 'W' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 6; Taking action: u
[[ '+' '-' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

```

```
Reward: 10
```

Can we get a round of applause for our Gridworld player here? Clearly it knows what it's doing; it went straight for the goal! But let's not get too excited; this was the static version of the game, which is really easy. If you use our test function with the `mode='random'`, we'll find some disappointment:

```
>>> testModel(model, 'random')

Initial State:
[[ '+' ' ' 'P' ]
 [ ' ' 'W' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 0; Taking action: d
[[ '+' ' ' ' ' ' ' ]
 [ ' ' 'W' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 1; Taking action: d
[[ '+' ' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 2; Taking action: l
[[ '+' ' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 3; Taking action: l
[[ '+' ' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 4; Taking action: l
[[ '+' ' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' 'P' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 5; Taking action: u
[[ '+' ' ' ' ' ' ' ]
 [ 'P' 'W' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

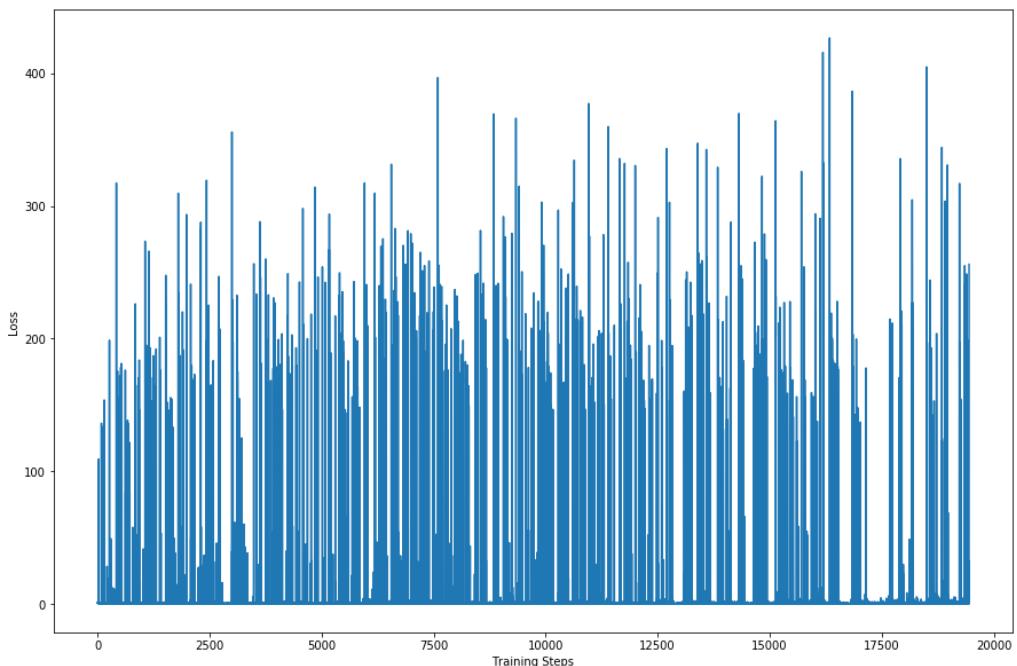
Move #: 6; Taking action: u
[[ 'P' '+' ' ' ' ' ' ]
 [ ' ' 'W' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

Move #: 7; Taking action: d
[[ '+' ' ' ' ' ' ' ]
 [ 'P' 'W' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]
 [ ' ' ' ' ' ' ' ' ]]

# we omitted the last several moves to save space
```

```
Game lost; too many moves.
```

This is actually really interesting, look carefully at the moves that the network is making. The player starts off the game only 2 tiles to the right of the goal, if it *really* knew how to play the game, it would just take the shortest path to the goal. Instead it starts moving down and to the left, just like it would in the static game mode. It seems like the model just memorized the particular board it was trained on and didn't generalize at all. Ah, but maybe we just need to train it with the game mode set to random, and then it will really learn right? Try it, go ahead. Re-train it with random mode. Maybe you'll be luckier than us, but this is what our loss plot looks like with random mode and 1000 epochs:



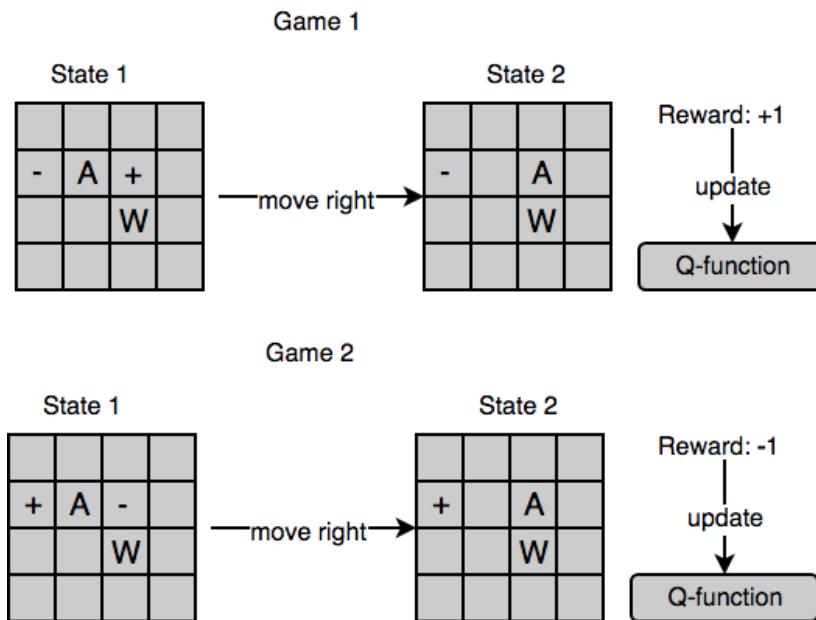
Well, that doesn't look pretty. There's no sign that any significant learning is happening at all with random mode. We didn't show these results, but for us, the model *did* seem to learn how to play with "player" mode where only the player is randomly placed on the grid. In any case, this is a big problem, reinforcement learning wouldn't be worth anything if all it could do is learn how to memorize or weakly learn. But indeed, this is a problem that the DeepMind team faced and one they solved.

### 3.3 Preventing Catastrophic Forgetting: Experience Replay

We're slowly building up our skills and we want our algorithm to train on the harder variant of the game where every new game the all the board pieces are randomly placed on the grid. It can't just memorize a sequence of steps to take as before, it needs to be able to take the shortest path to the goal (without stepping into the pit) from what the initial board configuration is. It needs to develop a more sophisticated representation of its environment.

**CATASTROPHIC FORGETTING** The main problem we encountered in the previous section when we tried to train our model on random mode has a name, it's called **catastrophic forgetting**. It's actually a very important issue that is associated with gradient descent-based training methods in *online* training. Online training is what we've been doing: we backpropagate after each move as we play the game.

Imagine that in game #1 that our algorithm is training on (learning Q-values for) the player is placed in between the pit and the goal such that the goal is on the right and the pit is on the left.



**Figure 3.8: Catastrophic forgetting.** The idea is that when two game states are very similar and yet lead to very different outcomes, the q-function will get “confused” and won’t be able to learn what to do. In this example, the catastrophic forgetting happens because the q-function learns from game state 1 that taking action “right” leads to a +1 reward, but then game state 2, which looks very similar, gets a reward of -1 after moving right, so the algorithm then forgets what it previously learned about game state 1, overall resulting in essentially no significant learning at all.

Using an epsilon-greedy strategy, the player takes a random move and by chance takes a step to the right and hits the goal. Great, the algorithm will try to learn that this state-action pair is associated with a high value by updating its weights in such a way that the output will more closely match the target value (i.e. via backpropagation). Now, the second game gets initialized and the player is again in between the goal and pit but this time the goal is on the *left* and the pit is on the right. Perhaps to our naive algorithm, the state *seems* very similar to the last game. Since last time moving right gave a nice positive reward, the player chooses to make one step to the right again, but this time it ends up in the pit and gets -1 reward. The player is thinking "what is going on, I thought going to the right was the best decision based on my previous experience." So now it may do backpropagation again to update its state-action value but because this state-action is very similar to the last learned state-action it may override its previously learned weights.

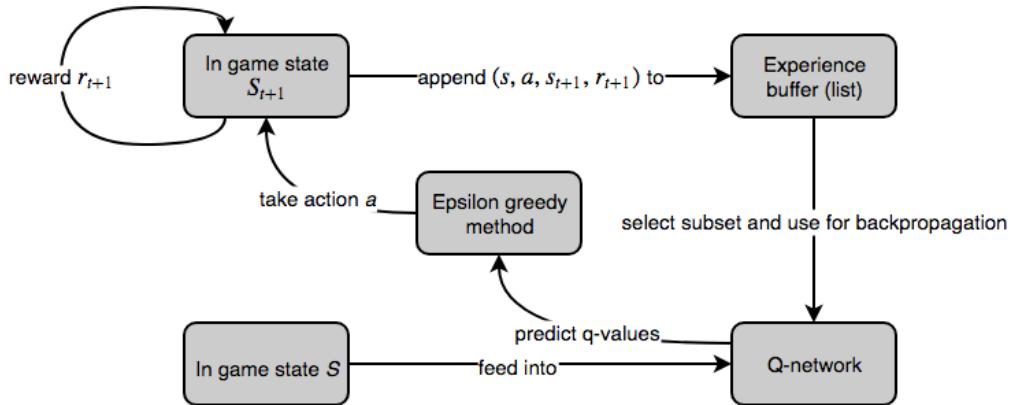
This is the essence of catastrophic forgetting. There's a push-pull between very similar state-actions (but with divergent targets) that results in this inability to properly learn anything. We generally don't have this problem in the supervised learning realm because we do randomized batch learning, where we don't update our weights until we've iterated through some random subset of our training data and compute the sum or average gradient for the batch, and this sort of averages over the targets and stabilizes the learning.

**EXPERIENCE REPLAY.** Catastrophic forgetting is probably not something we have to worry about with the first variant of our game because the targets are always stationary, and indeed the model successfully learned how to play it. But with the random mode, it's something we need to consider, and that is why we need to implement something called **experience replay**. Experience replay basically gives us batch updating in an online learning scheme. It's actually not a big deal to implement; here's how it works.

Experience replay:

1. In state  $s$ , take action  $a$ , observe new state  $s_{t+1}$  and reward  $r_{t+1}$
2. Store this as a tuple  $(s, a, s_{t+1}, r_{t+1})$  in a list.
3. Continue to store each experience in this list until we have filled the list to a specific length (up to you to define)
4. Once the experience replay memory is filled, randomly select a subset (again...you need to define the subset size)
5. Iterate through this subset and calculate value updates for each; store these in a target array (e.g.  $y_{\text{train}}$ ) and store the state  $s$  of each memory in  $X_{\text{train}}$
6. Use  $X_{\text{train}}$  and  $y_{\text{train}}$  as a minibatch for batch training. For subsequent epochs where the array is full, just overwrite old values in our experience replay memory array.

## Experience Replay



**Figure 3.9:** This is the general overview of experience replay, a method to mitigate a major problem with online training algorithms: catastrophic forgetting. The idea is to employ minibatching by storing past experiences and then using a random subset of these experiences to update the Q-network, rather than just the single most recent experience.

Thus, in addition to learning the action-value for the action we just took, we're also going to use a random sample of our past experiences to train on to prevent catastrophic forgetting.

So here's the same training algorithm from above except with experience replay added. Remember, this time we're training it on the harder variant of the game where all the board pieces are randomly placed on the grid.

### Listing 3.5 Experience Replay

```

epochs = 3000
losses = []
batchSize = 100 #A
buffer = 500 #B
replay = [] #C
max_moves = 50 #D
h = 0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
    state = Variable(torch.from_numpy(state_).float())
    status = 1
    mov = 0
    while(status == 1):
        mov += 1
        qval = model(state)
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_.flatten())
        reward = game.step(action)
        next_state = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
        next_state = Variable(torch.from_numpy(next_state).float())
        done = game.isDone()
        status = game.status
        losses.append(model.loss_fn(reward, qval_, next_state))
        replay.append((state, action_, reward, next_state, done))
        if len(replay) > batchSize:
            minibatch = random.sample(replay, batchSize)
            for state, action, reward, next_state, done in minibatch:
                loss = model.loss_fn(reward, qval_[action], next_state)
                model.optimizer.zero_grad()
                loss.backward()
                model.optimizer.step()
            replay = []
    if i % 100 == 0:
        print("Epoch: ", i)
        print("Loss: ", np.mean(losses[-100:]))
        print("Average moves: ", np.mean(movs[-100:]))
        print("Status: ", status)
        print("Buffer size: ", len(replay))
  
```

```

else:
    action_ = (np.argmax(qval_))

action = action_set[action_]
game.makeMove(action)
new_state_ = game.board.render_np().reshape(1,64) +
np.random.rand(1,64)/100.0
new_state = Variable(torch.from_numpy(new_state_).float())
reward = game.reward()

if (len(replay) < buffer): #E
    replay.append((state, action_, reward, new_state))
else: #F
    replay.pop(0)
replay.append((state, action_, reward, new_state))
minibatch = random.sample(replay, batchSize) #G
X_train = Variable(torch.empty(batchSize, 4, dtype=torch.float))
y_train = Variable(torch.empty(batchSize, 4, dtype=torch.float))
h = 0
for memory in minibatch: #H
    old_state, action_m, reward_m, new_state_m = memory
    old_qval = model(old_state)
    newQ = model(new_state_m).data.numpy()
    maxQ = np.max(newQ)
    y = torch.zeros((1,4))
    y[:] = old_qval[:]
    if reward == 0:
        update = (reward_m + (gamma * maxQ))
    else:
        update = reward_m
    y[0][action_m] = update
    X_train[h] = old_qval
    y_train[h] = Variable(y)
    h+=1

loss = loss_fn(X_train, y_train)
print(i, loss.item())
optimizer.zero_grad()
loss.backward()
losses.append(loss.data[0])
optimizer.step()

state = new_state
if reward != 0 or mov > max_moves:
    status = 0
    mov = 0
if epsilon > 0.1:
    epsilon -= (1/epochs)

```

#A The number of samples from the experience replay buffer used as minibatches  
#B The size of the experience replay buffer  
#C The experience replay buffer, a list that will store tuples of state and reward data  
#D The maximum number of moves allowed before we decide to start a new game  
#E If the buffer has not been filled up, just add to it  
#F If the buffer is full, remove first item and add new item  
#G Sample a minibatch from the replay buffer

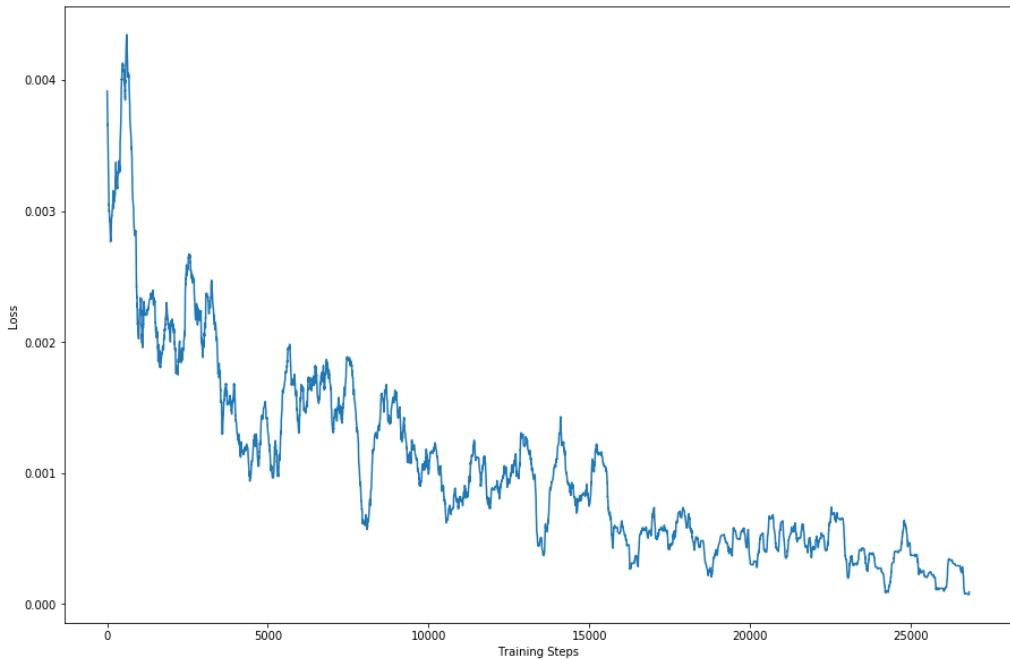
```
#H Iterate through each experience in the minibatch
```

We ran for 3000 epochs this time since it's a more difficult game, but otherwise the Q-network model is the same as before. When we test the algorithm, it seems to play most of the games correctly. We wrote an additional testing script to see the percentage of games it wins out of 1000 plays.

### **Listing 3.6 Testing the performance**

```
max_games = 1000
wins = 0
for i in range(max_games):
    win = test_model(model, mode='random', display=False)
    if win:
        wins += 1
win_perc = float(wins) / float(max_games)
print("Games played: {0}, # of wins: {1}".format(max_games,wins))
print("Win percentage: {}".format(win_perc))
```

When we run Listing 3.6 on our trained model, we got 86.5% wins out of 1000 games, which is certainly a lot better than random but is not nearly what we would expect if the algorithm really knew what it was doing (although you could probably improve the accuracy with a much longer training time). Once you actually know how to play, you should be able to win every single game. There's a small caveat that some of the initialized games may actually be impossible to win and thus the win percentage may not ever reach 100%. The Gridworld game engine does eliminate most of the impossible board configurations but there's a small number that can still get through. What's holding us back from getting into the 95%+ accuracy territory? Well let's have a look at our loss plot. Here we plotted the running average loss plot (yours may vary significantly):



You can see it's definitely trending downward but looks pretty unstable. This is the type of plot you'd be a bit surprised to see in a supervised learning problem, but is quite common in bare deep reinforcement learning. While the experience replay mechanism helps with training stabilization by reducing catastrophic forgetting, there are still other related sources of instability.

### 3.4 Improving Stability with a Target Network

So far we've been able to successfully train a deep reinforcement learning algorithm to learn and play Gridworld with a deterministic static initialization and a slightly harder version where the player is placed randomly on the board each game. Unfortunately, even though the algorithm appears to learn how to play, it is quite possible it is just memorizing all the possible board configurations since on a 4x4 board it's not that many. The hardest variant of the game is where the player, goal, pit and wall are all initialized randomly each game, making it much more difficult for the algorithm to memorize. This ought to enforce some amount of actual learning, but as we saw, we're still experiencing difficulty with learning this variant. To help address this, we're going to add another dimension to the updating rule that will smooth out the value updates.

**LEARNING INSTABILITY.** One potential problem that DeepMind identified when they published their deep Q-network paper was that if you just keep updating the Q-network's (i.e. the neural

network serving the role as Q-function) parameters after each move, you might cause instabilities to arise. The idea is that, since the actual rewards may be sparse (you only get a significant positive reward when you win the game or significant penalty upon losing), if you update on every single step where most steps you're not seeing any significant reward, you may cause the algorithm to start behaving erratically. For example, the Q-network might predict a high value for the "up" action in some state, then it moves up, and by chance it lands on the goal and wins, so we update the Q-network to reflect the fact that it was rewarded +1. The next game, however, it thinks "up" is a really fantastic move and predicts a high q-value, but then it moves up and gets a -1 reward, so we update and now it thinks "up" is not so great after all, but then a few games later moving up leads to winning again. You can see how this might lead to a kind of oscillatory behavior where the predicted q-value never settles to a reasonable value but just keeps getting jerked around. Of course this is very similar to the catastrophic forgetting problem.

This is not just a theoretical issue, it's something that DeepMind observed in their own training and had to come up with a solution. The solution they devised is to split the Q-network into two copies each with their own model parameters: the "regular" Q-network and a copy called the target network (symbolically denoted  $\theta$ -network, read "Q hat"). The target network is identical to the Q-network in the beginning before we've done any training, but its own parameters lag behind the regular Q-network in terms of how they're updated. Let's run through the sequence of events again with the target network in play (we'll leave out the details of experience replay).

#### Target Network:

1. Initialize Q-network with parameters (weights)  $\theta_Q$  (read "theta Q")
2. Initialize target network as copy of Q-network, but with separate parameters  $\theta_T$  (read "theta T"), and we set  $\theta_T = \theta_Q$ .
3. Use epsilon greedy strategy with Q-network q-values to select action  $a$
4. Observe reward and new state  $r_{t+1}, s_{t+1}$
5. Target network q-value will be set to  $r_{t+1}$  if the episode has just been terminated (i.e. the game is won or lost) or to  $r_{t+1} + \gamma \max Q_{\theta_T}(s_{t+1})$  otherwise (notice use of target network here)
6. Backpropagate the target q-value through the Q-network (not the target network)
7. Every Cnumber of iterations, set  $\theta_T = \theta_Q$  (i.e. set the target network parameters equal to the Q-network's parameters)

## Q-Learning with a Target Network

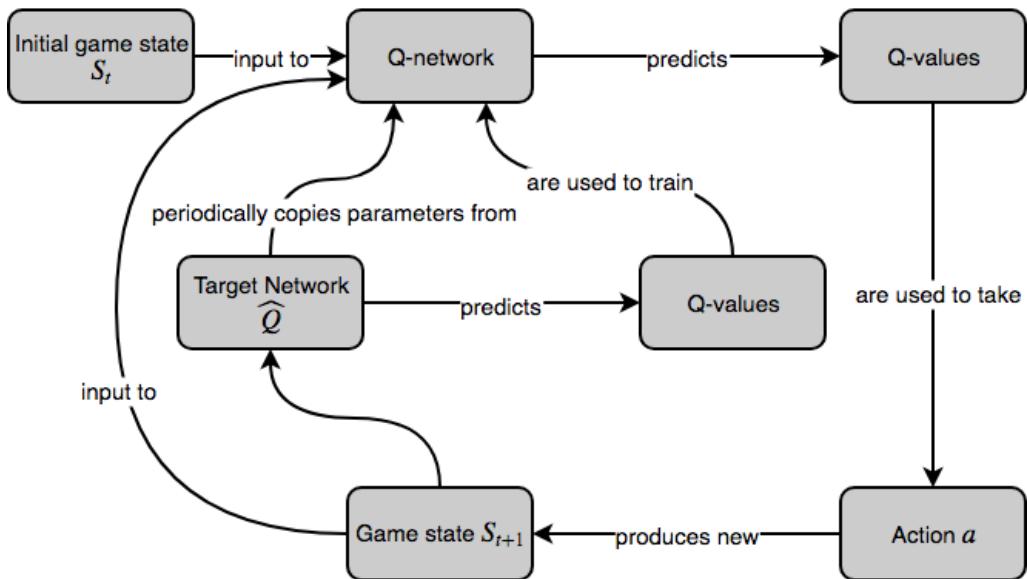


Figure 3.10: This is the general overview for q-learning with a target network. It's a fairly straightforward extension of the normal q-learning algorithm, except that you have a second q-network called the target network whose predicted q-values are used to backpropagate through and train the main Q-network. The target network's parameters are not trained, but are periodically synchronized with the Q-network's parameters. The idea is that using the target network's q-values to train the Q-network will improve the stability of training.

Notice that that only time we use the target network  $\hat{Q}$  is to calculate the target q-value for backpropagation through the Q-network. The idea is that we update the Q-network's parameters on each training iteration, but we decrease the effect that recent updates have on the action selection, hopefully improving stability. The code is getting a bit long now with both experience replay and a target network, so we're just going to show a portion of the full code and leave it to you to checkout the book's GitHub <<http://github.com/DeepReinforcementLearning/>> where you'll find all the code for this chapter. The following code is identical to Listing 3.5 except for a few lines that add in the target network capability.

### **Listing 3.7 Target Network**

```
model3 = torch.nn.Sequential(
    torch.nn.Linear(11, 12),
    torch.nn.ReLU(),
    torch.nn.Linear(12, 13),
    torch.nn.ReLU(),
```

```

        torch.nn.Linear(13,14),
)

model3_ = copy.deepcopy(model3) #A

#B

epochs = 5000
losses = []
batchSize = 250
buffer = 2000
replay = []
max_moves = 100
c = 500 #C
c_step = 0
h = 0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
    state = Variable(torch.from_numpy(state_).float())
    status = 1
    mov = 0
    #while game still in progress
    while(status == 1):
        c_step += 1
        if c_step > c:
            model3_.load_state_dict(model3.state_dict()) #D
            c_step = 0
        mov += 1
        qval = model3(state)

#E

        for memory in minibatch:
            old_state, action_m, reward_m, new_state_m = memory
            old_qval = model3(old_state)
            newQ = model3_(new_state_m).data.numpy() #F
            maxQ = np.max(newQ) #G

```

#A We create a second model by making an identical copy of the original Q-network model.

#B Code omitted

#C This is the parameter that controls how often we will synchronize model3\_ (target network) with model3 (Q-network)

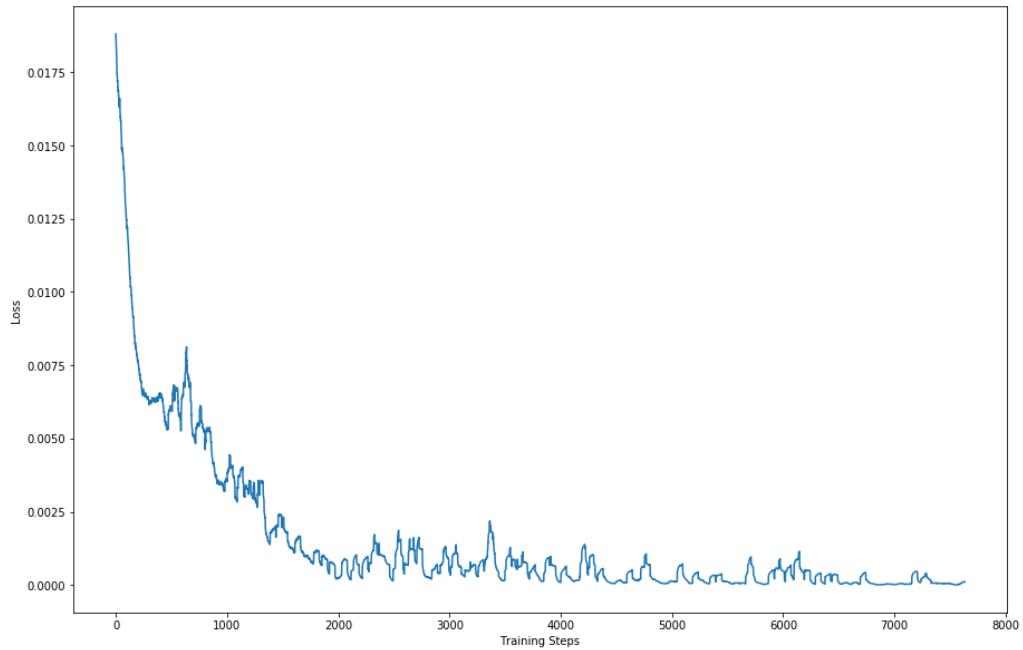
#D Every 50 training steps we synchronize the target network's parameters with the q-network

#E Code omitted

#F We use the target network's predicted q-values to calculate the target q-value for training the Q-network

#G Rest of code omitted

When we plot the loss for a target network approach plus experience replay, we get a nicely decreasing trend. In any case, you'll want to experiment with the hyperparameters such as the experience replay buffer size, the batch size, the target network update frequency, and the learning rate. Unfortunately, the performance can be quite sensitive to these.



And if you run the test script, you should be getting over 90% accuracy. We're only training up to 5000 epochs, where each epoch is a single game. The number of possible game configurations (i.e. the size of the state-space) is approximately 43,680 ( $16 \times 15 \times 14 \times 13$ ), so we're only sampling about  $5000/43680 = 0.11$  or 11% of the total number of possible starting game states. If the model can successfully play games it has never seen before, then we have some confidence it has generalized. If you're getting good results with the 4x4 board, you should try training an agent to play a 5x5 board or larger by changing the size parameter when creating a Gridworld game instance, e.g.

```
>>> game = Gridworld(size=6, mode='random')
>>> game.display()

array([[' ', '+', ' ', ' ', ' ', ' '],
       [' ', ' ', ' ', ' ', ' ', ' '],
       [' ', ' ', 'W', ' ', ' ', ' '],
       [' ', ' ', ' ', ' ', ' ', ' '],
       [' ', ' ', ' ', ' ', 'P', ' '],
       [' ', ' ', ' ', ' ', ' ', ' ']], dtype='<U2')
```

### DeepMind's Deep Q-Network

Believe it or not, but in this chapter we basically built the Deep Q-Network (DQN) that DeepMind originally introduced in 2015 that learned to play old Atari games at superhuman performance levels. DeepMind's DQN used an epsilon-greedy action-selection strategy, experience replay, and a target network. Of course the details of our implementation are different since we are playing a custom Gridworld game and DeepMind was training on raw pixels from real video games. For example, one difference worth noting is that they actual input the last 4 frames of a game into their Q-network. The reason is that a single frame in a video game is not enough information to determine the speed and direction of the objects in the game, which is important to know when deciding what action to take.

You can read more about the specifics of DeepMind's DQN by searching for their paper "Human-level control through deep reinforcement learning." One thing to note is that they used a neural network architecture consisting of two convolutional layers followed by two fully connected layers. In our case we used 3 fully connected layers. It would be a worthwhile experiment to build a model with a convolutional layer at first and try training with Gridworld. One huge advantage of convolutional layers is that they are independent of the size of the input tensor. When we use a fully connected layer, for example, we had to make the first dimension 64, in our case we used a parameter matrix for the first layer. With a convolutional layer, however, it can be applied to input data of any length. This would allow you to train a model on a 4x4 grid and see if it generalizes enough to be able to play on a 5x5 or bigger grid. Go ahead, try it!

## 3.5 Summary

We've covered a lot in this chapter. And once again we've smuggled in a lot of fundamental reinforcement learning concepts. We could have pushed a bunch of academic definitions in your face to start, but we resisted the temptation, and decided to get to coding as quickly as possible and then tell you the definitions afterward. Let's review what we've accomplished and fill in a few terminological gaps.

In this chapter we covered a particular RL algorithm called Q-learning. Q-learning has nothing to do with deep learning or neural networks on its own, it is only an abstract mathematical construct. Q-learning refers to solving a control task by learning a function called a Q-function. You give the Q-function a state (e.g. a game state) and it predicts how valuable all the possible actions are that you could take given the input state, which we call q-values. You decide what to do with these q-values. You might decide to just take the action that corresponds to the highest q-value (a greedy approach), or you might opt for a more sophisticated selection process. As we learned in chapter 2, you have to balance exploration (trying new things) versus exploitation (taking the best action you know of). In this chapter, we used the standard epsilon-greedy approach to select actions, where we initially take random actions to explore, and then progressively switch our strategy to taking the highest value actions.

Back to the Q-function. The Q-function must be learned from the data we give it. It has to learn how to make accurate q-value predictions of states. The Q-function could be anything really, anything from an unintelligent database to a complex deep learning algorithm. Since deep learning is the best class of learning algorithms we have at the moment, we choose to employ neural networks as our Q-functions. This means that "learning the Q-function" is the same as training a neural network with backpropagation.

One important concept about Q-learning that we held until now is that it is an **off-policy** algorithm, in contrast to **on-policy** algorithms. You already know what a policy is from last chapter: it's the strategy our algorithm uses to maximize rewards over time. If a human is learning to play Gridworld, they might employ a policy that first scouts all possible paths toward the goal and selects the one that is shortest. Another policy might be to just randomly take actions until you land on the goal. An off-policy reinforcement learning algorithm like Q-learning means that the choice of policy does not affect the ability to learn accurate q-values. Indeed, our Q-network could learn accurate q-values if we selected actions at random since eventually it would experience a number of winning and losing games and infer the values of states and actions. Of course, this is terribly inefficient, the policy matters only insofar as it helps us learn with the least amount of data. In contrast, an on-policy algorithm will explicitly depend on the choice of policy or will directly aim at learning a policy from the data.

Another key concept we've saved until now is the notion of **model-based** versus **model-free** algorithms. To make sense of this, we first need to understand what a model is. We sort of use this term informally to refer to a neural network, for example, or it's often used to mean any kind of statistical model, others being a linear model or a Bayesian graphical model. In another context, we might say a model is a mental or mathematical representation of how something in "the real world" works. If we understand exactly how something works (i.e. what it's composed of and how those components interact) then we can not only explain data we've already seen, but we can predict data we haven't yet seen.

For example, weather forecasters build very sophisticated models of the climate that take into account many relevant variables and are constantly measuring real-world data. Then they can use their models to predict the weather to some degree of accuracy. There's an almost cliché statistics mantra that "all models are wrong, but some are useful," meaning that it is impossible to build a model that 100% corresponds to reality (if that were the case, you'd basically have built a reality simulator), there will always be data or relationships that we're missing, but nonetheless, many models capture enough truth about some system we're interested in that they're useful for explanation and prediction.

If we could build an algorithm that could figure out how Gridworld works, then it would have inferred a model of Gridworld, and it would be able to perfectly play it. In Q-learning, all we gave the Q-network was a numpy tensor. It had no *a priori* model of Gridworld, but it still learned to play by trial and error. We did not task the Q-network with figuring out how Gridworld works, its only job was to predict expected rewards. Hence, Q-learning is a model-free algorithm.

As the human architects of algorithms, we may be able to engineer in some of our own domain knowledge about a problem as a model. We could then supply this model to a learning algorithm and then let it figure out the details to optimize our problem. This would be a model-based algorithm. For example, most Chess playing algorithms are model-based, that is, they know how Chess works, they know the rules of the games and what the result of taking certain moves will be. The only part that isn't known (and we would want the algorithm to

figure out) is what sequence of moves will win the game. With a model in hand, the algorithm can actually make long-term plans in order to achieve its aim.

In many cases, we want to employ algorithms that can progress from being model-free to planning with a model. For example, a robot learning how to walk may start to learn by trial and error (model-free), but once it's figured out the basics of walking, it can start to infer a model of its environment and then plan a sequence of steps to get from point A to B (model-based). We'll continue to explore on and off-policy and model-based and free algorithms in the rest of the book.

### 3.6 What's next?

Q-learning is about training an agent to learn the most accurate Q-function for an environment. We can then use this learned Q-function to figure out what action to take. In the next chapter we'll learn about an algorithm that will help us build a network that can approximate the policy function.

# 4

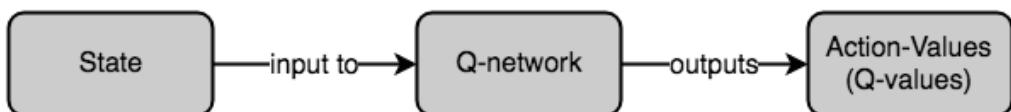
## *Learning to Pick the Best Policy: Policy Gradient Methods*

### This chapter covers:

- Implementing the policy function as a neural network
- Introduction to OpenAI Gym API
- Applying the REINFORCE algorithm on the OpenAI Cartpole problem

### 4.1 Policy Function using Neural Networks

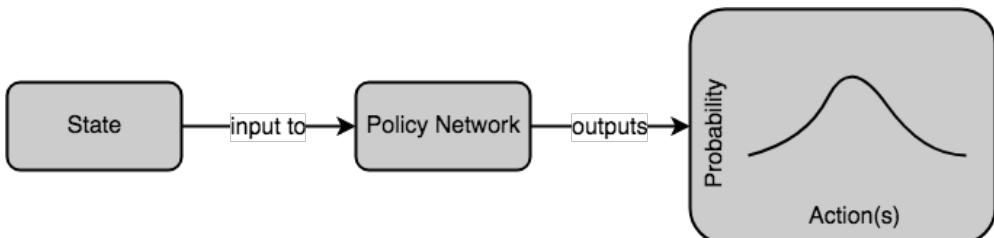
In the previous chapter we discussed Deep Q-networks, an off-policy algorithm where we approximated the Q-function with a neural network. The output of the Q-network were q-values corresponding to each action for a given state, and recall that value means the expected (i.e. weighted average or sum) of rewards.



Using these predicted Q-values from the Q-network, we could select actions to perform using some strategy. The strategy we employed in the last chapter was the epsilon-greedy approach where we select an action at random with probability  $\epsilon$  and with probability  $1-\epsilon$  we select the action associated with the highest q-value (the action the Q-network predicts is the best given its experience so far). There are other policies that we could have followed, such as using a softmax layer on the q-values, or any number of other ways of selecting actions. What if we

skip selecting a policy and instead train a neural network to output an action directly? If we do that, then our neural network ends up being a policy function, or a policy network. Remember from chapter 3, a policy function  $\pi: \text{State} \rightarrow P(\text{Action} | \text{State})$  accepts a state and returns the best action, or more precisely, it will return a probability distribution over the actions and we can sample from this distribution to select actions. If a probability distribution is an unfamiliar concept to you, don't worry, we'll discuss it more throughout the chapter and the book.

**Neural Network as the policy function.** In this chapter we introduce a class of algorithms where we approximate the policy function  $\pi(s)$  instead of the value function  $V_\pi$  or  $Q$ . That is, instead of training a network that outputs action-values, we are training a network to output (the probability of) actions.

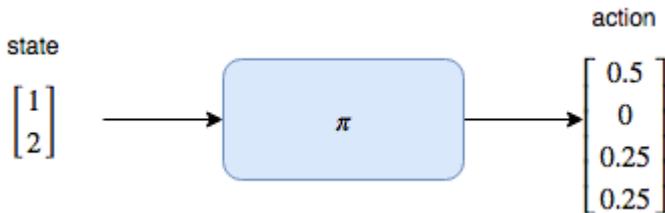


So in contrast to the Q-network, the policy-network tells us exactly what to do given the state we're in. No further decisions necessary. All we need to do is randomly sample from the probability distribution  $P(A | s)$ , and we get an action to take. The actions that are most likely to be beneficial will have the highest chance of being selected from random sampling. Imagine the probability distribution  $P(A | s)$  as a jar filled with little notes with an action written on them. In a game with 4 possible actions, there will be notes with labels 1-4. If our policy network predicts that action 2 is the most likely to result in the highest reward, then it will fill this jar with a lot of little notes labeled 2, and less notes labeled 1, 3 and 4. In order to select an action then, all we do is close our eyes and grab a random note from the jar. We're most likely to choose action 2, but sometimes we'll grab another action and that gives us the opportunity to explore. So using this analogy, every time the state of the environment changes, we give the state to our policy network and it uses that to fill the jar with a new set of labeled notes representing the actions in different proportions and then we randomly pick from the jar.

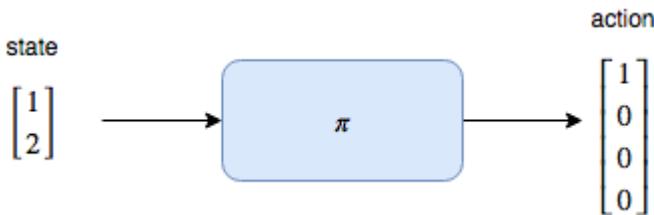
This class of algorithms is called policy gradient methods and has a few important differences from the DQN algorithm which we will explore in this chapter. There are a few advantages of this approach over value prediction methods. One is that, as we already discussed, we no longer have to worry about devising an action-selection strategy like epsilon-greedy, we just sample from the actions. Remember we spent a lot of time cooking up methods to improve the stability of training our DQN, we had to use experience replay and target networks and there are a number of other methods in the academic literature that we

could have used. A policy network tends to simplify some of the complexity we dealt with when using a Q-network.

**Stochastic Policy Gradient.** There are many different flavors of policy gradient methods. We will be starting with the stochastic policy gradient method, which is what we just described. Stochastic policy gradient means that the output of our neural network is an action vector that represents a probability distribution.



The policy we follow is to select an action from this probability distribution. This also means that if our agent ends up in the same state twice, we may not end up taking the same action every time. In the above example, we feed our function the state, which is  $(1,2)$ , and the output is an action vector. Our agent will have a 0.50 probability of going left, no chance of going up, 0.25 probability of going right, and 0.25 probability of going down.



If the environment is static and there is a deterministic strategy, we expect the probability distribution to converge to a more deterministic action vector as shown above.

### I forget...What's a probability distribution?

Let's be concrete. In Gridworld, we had 4 possible actions: up, down, left, right. We call this our **action-set** or **actions-space**, since we can describe it mathematically as a set, e.g.  $A=\{\text{up},\text{down},\text{left},\text{right}\}$  where the curly braces indicate a set (a set in mathematics is just an abstract un-ordered collection of things with certain operations defined). So what does it mean to apply a probability distribution over this set of actions?

Probability is actually a very rich and even controversial topic in its own right. There are varying philosophical opinions on what exactly probability means. To some people, the probability means that, if you were to flip a coin a very large number of times (ideally an infinite number of times, mathematically speaking) then the probability of a fair coin turning up heads is equal to the proportion of heads in that infinitely long sequence of flips. That is, if we flip a fair coin 1,000,000 times, we expect about half of the flips to be heads and the other half tails, and hence the probability is

equal to that proportion. This is a frequentist interpretation of probability since probability is interpreted as the long-term frequency of some event repeated many times.

In contrast, another school of thought interprets probability only as a degree of belief, a subjective assessment of how much I can predict an event given the knowledge I currently possess. This degree of belief is often called a credence. The probability of a fair coin turning up heads is 0.5 or 50% because, given what I know about the coin, I don't have any reasons to predict heads more than tails or tails more than heads, and hence I split my belief evenly across the two possible outcomes. Hence, anything that we can't predict deterministically (i.e. with probability 0 or 1, nothing in between) results from a lack of knowledge.

You're free to interpret probabilities however you want since it doesn't affect our calculations, but in this book we tend to implicitly use the credence interpretation of probability. So for our purposes, applying a probability distribution over the set of actions in Gridworld,  $A=\{up,down,left,right\}$  means to assign a degree of belief (a real number between 0 and 1) to each action in the set such that all the probabilities sum to 1. We interpret these probabilities as the probability that an action is the best action to maximize the expected rewards, given that we're in a certain state.

Concretely, a probability distribution over our action set  $A$  is denoted  $P(A):A \rightarrow [0, 1]$ , meaning that  $P(A)$  is a map from a set  $A$  to a set of real numbers between 0 and 1. In particular, each element  $a_i \in A$  is mapped to a single number between 0 and 1 such that the sum of all these numbers for each action is equal to 1. We might represent this map for our Gridworld action set as just a vector, where we identify each position in the vector with an element in the action-set, e.g.  $[up, down, left, right] \rightarrow [0.25, 0.25, 0.10, 0.4]$ . This map is called a probability mass function (PMF).

What we just described is actually a discrete probability distribution, since our action-set was discrete (a finite number of elements). If our action-set was infinite, i.e. a continuous variable like velocity, then we would call this a continuous probability distribution and instead we would need to define a probability density function (PDF).

The most common example of a PDF is the normal (also known as Gaussian, or just bell-curve) distribution. If we have a probability with a continuous action, say a car game where we need to control the velocity of the car from 0 to some maximum value, which is a continuous variable, how might we do this with a policy network? Well, we could drop the idea of probability distribution and just train the network to produce the single value of velocity that it predicts is best, but then we risk not exploring enough (and it is difficult to train such a network). A lot of power comes from a little bit of randomness. The kind of neural networks we employ in this book only produce vectors (or tensors more generally) as output, so they can't produce a continuous probability distribution—we have to be more clever. A PDF like a normal distribution is defined by two parameters, the mean and variance. Once we have those, we have a normal distribution that we can sample from. So we can just train a neural network to produce mean and standard deviation values that we then plug into the normal distribution equation and sample from that.

Don't worry, if this isn't all making sense now, we will continue to go over it again and again as these concepts are ubiquitous in reinforcement learning and machine learning more broadly.

---

But in many problems the environment is not static. Think of playing rock, paper, scissors. When playing against a player for the first time, there is no best action the optimal strategy and we would select each action with a 1/3 chance. If we were to this with a DQN algorithm and the values for each action were the same, we need to think carefully about what to do with the Q-values to split the tie. This is to say that there are often cases where there may be two or more actions with approximately equivalent value predictions, so we need a policy to determine how to break the tie, for example by flipping a coin and selecting one of the equivalently valued actions randomly. This tie-breaking mechanism adds complexity to the

algorithm. However, with a stochastic policy network, the need to take actions probabilistically in a stochastic environment is baked in to how the network works.

**Exploration.** Recall from the previous chapter that we needed our policy to include some randomness that would allow us to visit non-optimal states during training. For DQNs we followed the epsilon-greedy policy where there was a chance we would not follow the action that led to the greatest reward. If we always selected the action that led to the maximum predicted reward, then we'd never discover the even better actions and states available to us. For the stochastic policy gradient method, because our output is a probability distribution there should be a small chance that we explore all spaces, and only after sufficient exploration will the action distribution converge to producing the single best action. Or if the environment has an element of randomness, then the probability distribution will retain some probability mass to each action. When we initialize our model in the beginning, the probability of our agent picking each action should be approximately equal or uniform since the model has zero information about which action is better.

There is a variant of policy gradient called deterministic policy gradient (DPG) where there is a single output that the agent will always follow. In the case of Gridworld for example, it would produce a 4-dimensional binary vector with a 1 for the action to be taken and 0s for the other actions. This causes the agent to not explore properly if it always follows the output since there's no randomness in the action selection. Since the output of a deterministic policy function for a discrete action set would be discrete values, it is also difficult to get this working in the fully differentiable manner that we are accustomed to with deep learning, so we'll focus on stochastic policy gradients. Building a notion of uncertainty (e.g. using probability distributions) into the models is generally a good thing.

## 4.2 Reinforcing Good Actions: The Policy Gradient Algorithm

From the previous sections, we understand that there is a class of algorithms which attempts to create a function that outputs a probability distribution over actions and that this policy function  $\pi(s)$  can be implemented with a neural network.

**Defining An Objective.** Recall that neural networks need an objective function that is differentiable with respect to the network weights (parameters). In the last chapter we trained the Deep Q-network by a minimizing mean square error (MSE) loss function with respect to its predicted q-values and the target  $q$ -value. We had a nice formula for calculating the target  $q$ -value based on the observed reward since  $q$ -values are just averaged rewards (i.e. expectations), so this was not much different from a how we would normally train a supervised deep learning algorithm.

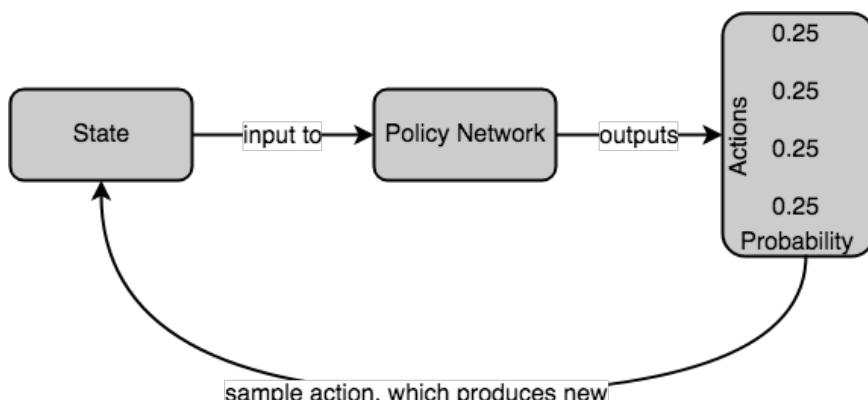
With a policy network that gives us a probability distribution over actions given a state,  $P(A | s)$ , how do we train this? There's no obvious way to map our observed rewards after taking an action to updating  $P(A | s)$ . Training the DQN was not much different than a supervised learning problem, because our Q-network generated a vector of predicted Q-

values, and using some formula we were able to generate the target q-value vector, and then we just minimized the error between the Q-network predictions and our target. With a policy network, we're predicting actions directly, and there is no way to come up with a target vector of what actions we should have taken instead, given the rewards. All we know is whether the action lead to positive or negative rewards. In fact, what the best action is secretly depends on a value function, but with a policy network we're specifically trying to avoid computing these action-values directly.

Let's go through an example to see how we might optimize our policy network. Let's start with some notation. Our policy network is denoted  $\pi$  and is parameterized by a vector  $\theta$ , this represents all of the weights of the neural network. As you know, neural networks have parameters in the form of multiple weight matrices, however, for the purposes of easy notation and discussion, it is standard to consider all the network parameters together as a single long vector, that we denote  $\theta$  (theta).

Whenever we run the policy network forward the parameter vector  $\theta$  is fixed, the variable is the data that gets fed into the policy network (i.e. the state). Hence, we denote the parameterized policy as  $\pi_\theta$ . Whenever we want to indicate that some input to a function is fixed, we will include it as a subscript rather than as an explicit input like  $\pi(x, \theta)$  where  $x$  is some input data (i.e. the state of the game). Notation like  $\pi(x, \theta)$  suggests  $\theta$  is a variable that changes along with  $x$ , whereas  $\pi_\theta$  indicates that  $\theta$  is a fixed parameter of the function.

Let's say we give our initially untrained policy network  $\pi_\theta$  some initial game state of Gridworld denoted  $s$ , and run it forward by computing  $\pi_\theta(s)$ . It returns a probability distribution over the 4 possible actions, for example  $[0.25, 0.25, 0.25, 0.25]$ . (It is a bit unrealistic that it would return a perfectly uniform distribution like that, but that won't affect our example). We sample from this distribution and since it's a uniform distribution, we essentially take a random action. We continue to take actions by sampling from the produced action distribution until we reach the end of the episode.



**Figure 4.1:** The general overview of policy gradients for an environment with 4 possible discrete actions. First we input the state to the policy network, which produces a probability distribution over the actions, then we sample

from this distribution to take an action, which produces a new state.

Remember, some games like Gridworld are episodic, meaning that there is a well-defined start and end point to a round of the game. In Gridworld, we start the game in some initial state, and play until we either hit the pit, land on the goal, or we take too many moves. So an episode is a sequence of states, actions, and rewards from an initial state to the terminal state where we win or lose the game. We denote this:

**Notation (episode):**

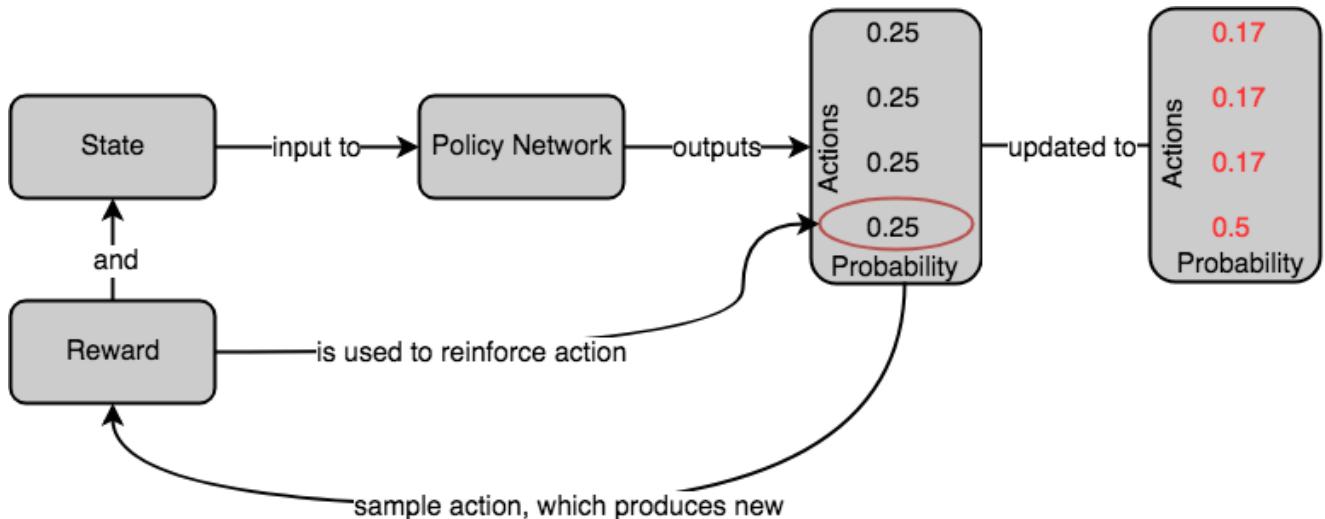
$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2), \dots, (S_{t-1}, A_{t-1}, R_t)$$

Each tuple is one time-step of the Gridworld game (or a Markov decision process more generally). After we've reached the end of the episode at time  $t$ , we've collected a bunch of historical data on what just happened. Let's say that by chance we hit the goal after just 3 moves determined by our policy network. Here's what our episode looks like:

$$\varepsilon = (S_0, 3, 0), (S_1, 1, 0), (S_2, 3, +1)$$

We've encoded the actions as integers from 0 to 3 (referring to array indices of the action vector) and left the states denoted symbolically since they're actually 64-length vectors. What is there to learn from in this episode? Well we won the game, indicated by the +1 reward in the last tuple, so our actions must have been "good" to some degree, so given the states we were in, we should encourage our policy network to make those actions more likely next time. We want to *reinforce* those actions that led to a nice positive reward. We will address what happens when our agent loses (receives a terminal reward of -1) later in this section, but in the meantime, we will focus on positive reinforcement.

**Positive Reinforcement.** We want to make small, smooth updates to our gradients to encourage the network to take assign more probability to these winning actions in the future. Let's focus on the last experience in the episode with state  $S_2$ . Remember, we're assuming our policy network produced the action probability distribution [0.25, 0.25, 0.25, 0.25] since it is untrained, and in the last time step we took action 3 (corresponding to element 4 in the action probability array) and it resulted in us winning the game with a +1 reward. We want to positively reinforce this action given state  $S_2$ , such that whenever the policy network encounters  $S_2$  or a very similar state, it will be more confident in predicting action 3 as the highest probability action to take.



**Figure 4.2:** One action is sampled from the policy network's probability distribution, it produces a new state and reward. The reward signal is used to reinforce the action that was taken, that is, increase the probability of that action given the state (if the reward is positive) or decrease if the reward was negative. Notice that we only received information about action 3 (element 4) but since probabilities must sum to 1, we have to lower the probabilities of the other actions.

A naïve approach might be to make a target action distribution  $[0, 0, 0, 1]$  so that our gradient descent will move the probabilities from  $[0.25, 0.25, 0.25, 0.25]$  close to  $[0, 0, 0, 1]$ , maybe ending up as  $[0.167, 0.167, 0.167, 0.5]$ . This is something we often do in the supervised learning realm when we are training a softmax-based image classifier. But in that case, there is a single correct classification to an image and there is no temporal association between each prediction. In our RL case, we want more control over how we make these updates. First, we want to make small, smooth updates because we want to maintain some stochasticity in our action sampling to adequately explore the environment. Secondly, we want to be able to weight how much we assign credit to each action for earlier actions. Let's review some more notation first before diving into these two problems.

Recall, our policy network is typically denoted  $\pi_\theta$  when we are running it forward (i.e. using it for to produce action probabilities), because we think of the network parameters  $\theta$  as being fixed and the input state is what varies. Hence calling  $\pi_\theta(s)$  for some state  $s$  will return a probability distribution over the possible actions given a fixed set of parameters. When we are training the policy network, we need to vary the parameters with respect to a fixed input to find a set of parameters that optimizes our objective (i.e. minimize a loss or maximizes a utility function), i.e.  $\pi_s(\theta)$ .

**NOTATION (CONDITIONAL PROBABILITY):** The probability of an action given the parameters of the policy network is denoted  $\pi_s(a | \theta)$ . This makes it clear that the probability of an action  $a$  explicitly depends on the parameterization of the policy network. In general, we denote a conditional probability as  $P(x | y)$  read “the probability distribution over  $x$  given  $y$ .” This means, we have some function that takes a parameter  $y$  and returns a probability distribution over some other parameter  $x$ .

In order to reinforce action 3, we want to modify our policy network parameters  $\theta$  such that we increase  $\pi_s(a_3 | \theta)$ . So our objective function merely needs to maximize  $\pi_s(a_3 | \theta)$  where  $a_3$  is action 3 in our example. Before training  $\pi_s(a_3 | \theta) = 0.25$  but we want to modify  $\theta$  such that  $\pi_s(a_3 | \theta) > 0.25$ . Because all of our probabilities must sum to 1, this means maximizing  $\pi_s(a_3 | \theta)$  will minimize the other action probabilities. And remember, we prefer to set things up so that we’re minimizing an objective function instead of maximizing, since it plays nice with PyTorch’s built-in optimizers, so we should instead tell PyTorch to minimize  $1 - \pi_s(a | \theta)$ . This loss function approaches 0 as  $\pi_s(a | \theta)$  nears 1 and so we are encouraging the gradients to maximize  $\pi_s(a | \theta)$  for the action we took. We will subsequently drop the subscript  $a_3$  as it should be clear from the context which action we’re referring to.

**Log Probability.** Mathematically this should make sense. But alas, due to computation imprecisions we need to make adjustment to this formula to stabilize the training. One problem is that probabilities are bounded between 0 and 1 by definition, so the range of values that the optimizer can operate over is limited and small. Sometimes probabilities may be extremely tiny or very close to one, and this runs into numerical issues when optimizing on a computer with limited numerical precision. If we instead use a surrogate objective, namely  $-\log \pi_s(a | \theta)$  (where  $\log$  is the natural logarithm), then we have an objective that has a larger “dynamic range” than raw probability space and this makes it easier to compute. There are a few other reasons for using log-probabilities rather than raw probabilities but suffice it to say that log-probabilities are in general just easier to work with. If we set our objective as  $-\log \pi_s(a | \theta)$  instead of  $1 - \pi_s(a | \theta)$  our loss still abides by the intuition that the loss function approaches 0 as  $\pi_s(a | \theta)$  approaches 1, while decreasing  $\pi_s(a | \theta)$  will increase objective. Our gradients will be tuned to try to increase  $\pi_s(a | \theta)$  to 1, where  $a =$  action 3, for our running example.

**Credit Assignment.** Okay, our objective function is  $-\log \pi_s(a | \theta)$ , but this assigns equal weight to every action in our episode. The weights in the network that produced the last action will be updated to the same degree as the first action. Why shouldn’t that be the case? Well it makes sense that the last action right before the reward “deserves more credit” for winning the game than does the first action in the episode. For all we know, the first action was actually sub-optimal, but then we later made a comeback and hit the goal. Or in other words, our confidence in how “good” each action diminishes the further we are from the point of reward. In a game of chess, we attribute more credit to last move made than the first one. We’re very confident that the move that directly led to us winning was a good move, but we

become less confident the further back we go. How much did the move from 5 time steps ago contribute to winning? We're not so sure.

We express this by multiplying the magnitude of the update by the discount factor, which we learned from previous Chapter 3 ranges from 0 to 1. The action right before the episode ends will have a discount factor of 1, meaning it will receive the full gradient update, while earlier moves will be discounted by a fraction such as 0.5 and therefore the gradient steps will be smaller.

Let's add those into our objective (loss) function. Our final objective function that we will tell PyTorch to minimize is  $-\gamma_t * G_t * \log \pi_s(a | \theta)$ . Remember  $\gamma_t$  is the discount factor and the subscript  $t$  tells us its value will depend on the time step  $t$ , since we want to discount more distant actions more than more recent ones. The parameter  $G_t$  is called the total return, or future return, at time step  $t$ . It is the return we expect to collect from time step  $t$  until the end of the episode and can be approximated by simply adding the rewards from some state in the episode until the end of the episode.

$$G_t = r_t + r_{t+1} + \dots + r_{T-1} + r_T$$

In Gridworld, all rewards are 0 except for the winning or losing reward, so the return is equal to the terminal reward. In other environments, however, positive or negative rewards may be given at each time step. So in our earlier example with three time steps that we ended up winning, we could calculate  $G_t$  as followed:

$$G_0 = 0 + 0 + 1 = 1$$

$$G_1 = 0 + 1 = 1$$

$$G_2 = 1$$

This is before we apply any discount though. Actions temporally more distant from the received reward should be weighted less than actions closer. From the start to terminal state, the sequence of discounted rewards in Gridworld if we win a game might look something like [0.970, 0.980, 0.99, 1.0]. The last action led to the winning state of +1, and is not discounted at all. The previous action is assigned a scaled reward by multiplying the terminal reward with the  $\gamma_{t-1}$  discount factor, which we've set to 0.99.

The discount is exponentially decayed from 1, i.e.  $\gamma_t = \gamma_0^{T-t}$  meaning that the discount at time  $t$  is calculated as the starting discount (here 0.99) exponentiated to the integer time distance from the reward. The length of the episode (total number of time steps) is denoted  $T$  and the local time step for a particular action is  $t$ . For  $T-t=0$ ,  $\gamma_{T-0}=0.99^0=1$ . For  $T-t=2$ , the  $\gamma_{T-2}=0.99^2=0.9801$  and so on. Each time step back, the discount factor is exponentiated to the distance from the terminal step, which results in an exponential decay of the discount factor the more distant (and irrelevant) the action was to the reward outcome.

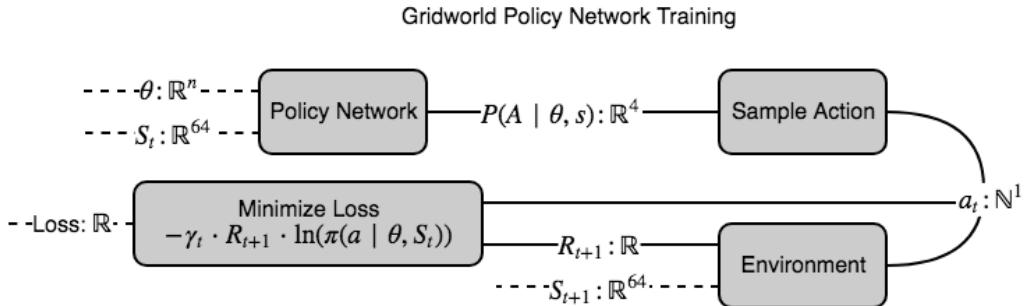
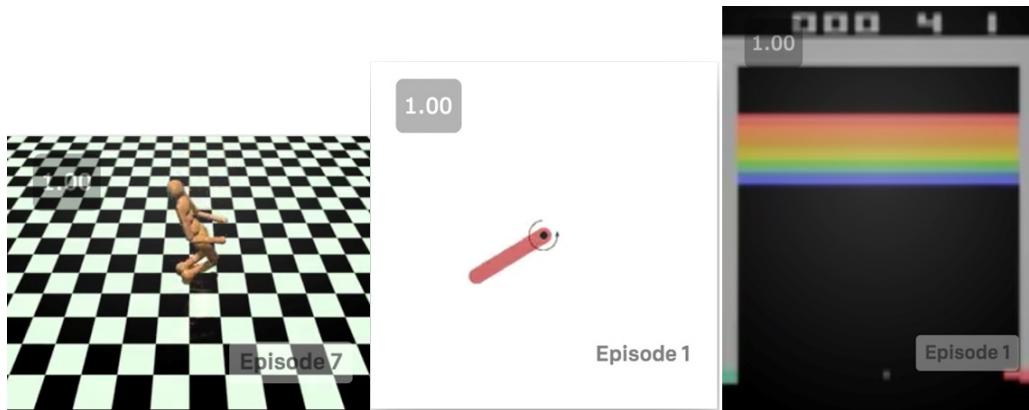


Figure 4.3: A string diagram for training a policy network for Gridworld. The policy network is a neural network parametrized by  $\theta$  (the weights) and accepts an input state that is a 64-dimensional vector. It produces a discrete 4-dimensional probability distribution over the actions. The sample action box samples an action from the distribution and produces an integer as the action, which is given to the environment (to produce a new state and reward) and to the loss function so we can reinforce that action. The reward signal is also fed into the loss function, which we attempt to minimize with respect to the policy network parameters.

**Negative Reinforcement.** And there we have it! We defined an objective function that will guide the weights in network to positively reinforce actions that led to rewards and the degree of reinforcement needs to be scaled in proportion to the amount of rewards and a discount factor (denoted  $y$ ). But what about episodes where we lose the game? This same objective function works even if we lose game (received a negative reward at the end) because our objective function will include a **negative** total return value instead of a positive one. Before including the total return value in our objective function, our objective function was just  $-\log \pi_s(a | \theta)$  which encourages the action taken to be taken again. But when include the return value back into the equation, we are multiplying the function by a negative number, which effectively discourages the action that was taken to be taken again in a similar state. Hopefully the mathematics of this all is starting to make sense, but the actual implementation in code may not be obvious yet. We're getting there.

### 4.3 Working with OpenAI Gym

To illustrate how policy gradients work we've been using Gridworld as an example since it is already familiar to you from last chapter. However, we thought we should use a different problem to actually implement the policy gradient algorithm for variety and also as a chance to introduce the OpenAI Gym. The OpenAI Gym is an open source suite of environments with a common API that is perfect for testing reinforcement learning algorithms. If you come up with some new Deep RL algorithm, testing it on a few of the environments in the Gym is a great place to get some idea of how well it is performing. The Gym contains very easy environments that can be "solved" by simple linear regression all the way to ones that all but require a sophisticated deep RL approach. There are games, robotic control and other types of environments. There's probably something in there you'll be interested in.



**Figure 4.4:** Three example environments provided by OpenAI's gym environment. The OpenAI gym provides hundreds of environments to test your reinforcement learning algorithms on.

OpenAI lists all of their currently supported environments on their website (<https://gym.openai.com/envs/>) and at the time of this writing they are broken down into 7 categories.

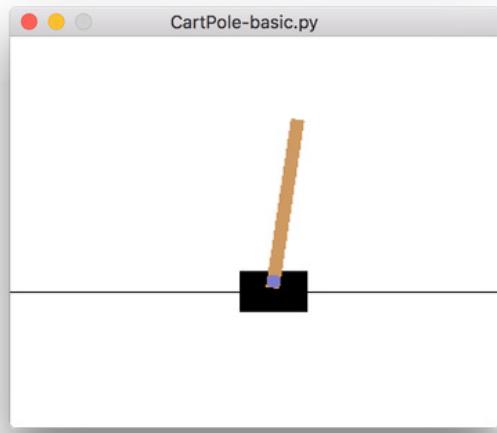
1. Algorithms
2. Atari
3. Box2D
4. Classic Control
5. MuJoCo
6. Robotics
7. Toy Text

They used to have board games, which included games like Go as a section, but removed it in a previous version (removed in v0.9.6). But don't worry, we have forked this library and provided a means for you to access the Go Game in a later chapter. In the meantime, you can just work with the latest version (v0.9.6). You can also view the entire list of environments from the OpenAI registry in your python shell with the following code snippet.

#### **Listing 4.1 OpenAI Gym**

```
from gym import envs
envs.registry.all()
```

There are hundreds of environments to choose from (797 in v0.9.6). Unfortunately, some of these environments require licenses (MuJoCo) or external dependencies (Box2D, Atari) and will therefore require a bit of setup time. As of now, we will be starting with a simple example, CartPole, to avoid any unnecessary complications and to get us coding right away.



**CartPole.** The CartPole falls under the Classic Control section above and has a very simple objective – don’t let the pole fall over. It’s the game equivalent of trying to balance a pencil on the tip of your finger. In order to balance the pole successfully, you have to apply just the right amount of small left and right movements to the cart. In this environment, there are only two actions that correspond to making a small push left or right. In the OpenAI Gym API, all actions are simply represented as integers from 0 to whatever the total number of actions are for the particular environment, so in Cartpole the possible actions are 0 and 1. The state is represented as an array of size four that indicates the cart position, cart velocity, pole angle and pole velocity. We receive a reward of +1 for every step the pole has not fallen over, which is when the pole angle is more than  $12^\circ$  from the center or the cart position is outside the window. Hence, the goal of Cartpole is to maximize the length of the episode since each time returns a positive +1 reward. More information can be found on the OpenAI gym github page (<https://github.com/openai/gym/wiki/CartPole-v0>). Note, not every problem has a nice specification page as CartPole but we will define the scope of the problem beforehand in all subsequent chapters.

**The OpenAI Gym API.** The OpenAI Gym has been built to be incredibly easy to use and there is less than half a dozen methods that you’ll routinely use. We have already seen one earlier in Listing 4.1 where we listed out all available environments. Another important method is creating our environment as shown below.

#### **Listing 4.2 OpenAI Gym**

```
import gym
env = gym.make('CartPole-v0')
```

From now on we will be interacting solely with this `env` variable. We need a way to observe the current state of the environment and then to interact with it. Thankfully, there are only two methods you need to do this.

#### **Listing 4.3 OpenAI Gym**

```
first_state = env.reset()
action = env.action_space.sample()
state, reward, done, info = env.step(action)
```

The `reset` method initializes our environment and returns the first state. For this example, we used the `sample` method of the `env.action_space` object to sample a random action. Soon enough, we'll sample actions from a trained policy network that will act as our reinforcement learning agent. Once we initialized our environment we are free to interact with it via the `step` method. The `step` method returns four important variables that our training loop needs access to in order to run. The first parameter `state` represents the next state after we take the action. The second parameter `reward` is the reward at that time step, which for our CartPole problem is 1 unless the pole has fallen down. The third parameter `done` is a Boolean that indicates whether or not a terminal state has been reached. For our CartPole problem this would initially always return false until the pole has fallen or the cart has moved outside the window. The last parameter `info` is a dictionary with diagnostic information that may be useful for debugging, however, we will not use it.

That's basically all you need to get most environments up and running in OpenAI Gym.

## **4.4 The REINFORCE Algorithm**

So now that we how to create an OpenAI gym environment and have hopefully developed an intuition for the policy gradient algorithm, let's dive in to getting a working implementation. Our discussion of policy gradients in the previous section was actually of a particular algorithm that has been around for decades (like most of deep learning and reinforcement learning) called REINFORCE (yes, it's always fully capitalized). We're going to consolidate what we learned previously, formalize it, and then turn it into glorious Python code.

Let's run through the steps of the algorithm with the Cartpole example.

**1. Create our policy network.** We build and initialize a neural network that serves as a policy network. The policy network will accept state vectors as inputs and produces a (discrete) probability distribution over the possible actions. You can think of "the agent" as a thin wrapper around the policy network that samples from the probability distribution to take an action. Remember, an agent in reinforcement learning is whatever function or algorithm takes a state and returns a concrete action that will be executed in the environment. Let's express this in code.

#### **Listing 4.4 Cartpole Policy Gradients**

```
import gym
```

```

import numpy as np
import torch

l1 = 4
l2 = 150
l3 = 2

model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.Softmax()
)

learning_rate = 0.0009
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

```

That should all look fairly familiar to you at this point. The model is only two layers with a leaky ReLU activation function for the first layer and the Softmax function for the last layer. We chose the leaky ReLU because it performed better empirically. We saw the Softmax function back in Chapter 2, it just takes an array of numbers and squishes them into a range of 0 to 1 and makes sure they all sum to 1, basically creating a discrete probability distribution out of any list of numbers (that are not probabilities to start with). For example,  $\text{softmax}([-1, 2, 3]) = [0.0132, 0.2654, 0.7214]$ . Unsurprisingly, the softmax will turn the bigger numbers into larger probabilities.

**2. Have the Agent Interact with the environment.** The agent consumes the state and takes an action  $a$  probabilistically. More specifically, the state is input to the policy network, which then produces the probability distribution over the actions  $P(A | \theta, s_t)$  given its current parameters and the state. Note, capital  $A$  refers to the set of all possible actions given the state whereas small  $a$  generally refers to a particular action. The policy network might return a discrete probability distribution in the form of a vector such as  $[0.25, 0.75]$  for our two possible actions in Cartpole. This means the policy network predicts action 0 is the best with 25% probability and action 1 is the best with 75% probability (or confidence). We call this array `pred`.

#### Listing 4.5 Cartpole Policy Gradients

```

pred = model(torch.from_numpy(state1).float()) #G
action = np.random.choice(np.array([0,1]), p=pred.data.numpy()) #H
state2, reward, done, info = env.step(action) #I

#G Call policy network model to produce predicted action probabilities
#H Sample an action from the probability distribution produced by the policy network
#I Take the action, receive new state and reward. The info variable is produced by the environment but is irrelevant

```

The environment responds to the action by producing a new state  $s_2$ , and a reward  $r_2$ . We will store those into two arrays (a *states* array and an *actions* array) for when we need to update our model after the episode ends. We then plug the new state into our model, get a

new state and reward, store those, and repeat until the episode ends (the pole falls over and the game is over).

**3. Training the model.** We train the policy network by updating the parameters to minimize the objective/loss function. This involves three steps: calculate the probability of the action actually taken at each time step, multiply it by the discounted return, and then use that to backpropagate and minimize the loss. Calculating the probabilities of the action taken is easy enough, we can use the stored past transitions to re-compute the probability distributions using the policy network, except this time, we extract just the predicted probability for the action that was actually taken. We'll denote this quantity  $P(a_t | \theta, s_t)$ ; this is a single probability value like 0.75.

To be concrete, let's say the current state is  $s_5$  (state at time step 5) and we input that into the policy network and it returns  $P_\theta(A | s_5) = [0.25, 0.75]$ . We sample from this distribution and take action  $a=1$  (the second element in the action array), and after this the pole falls over and the episode has ended. The total duration of the episode was  $T=5$ . For each of these 5 time steps, we took an action according to  $P_\theta(A | s_t)$  and we store the specific probabilities of the actions that were actually taken,  $P_\theta(a | s_t)$ , in an array, so it might look like  $[0.5, 0.3, 0.25, 0.5, 0.75]$ . We simply multiply these probabilities by the discounted rewards (explained below), take the sum, multiply it by -1 and call that our overall loss for this episode. Unlike Gridworld, in Cartpole, the last action is the one that loses the episode, so we discount it the most since it probably had the least affect in losing (since it takes several bad moves in a row to drop the pole). In Gridworld we would do the opposite and discount the first action in the episode the most since it would be the least responsible for winning or losing.

Minimizing this objective will tend to increase those probabilities  $P_\theta(a | s_t)$  weighted by the discounted rewards. So every episode we're tending to increase  $P_\theta(a | s_t)$  but for a particularly long episode (i.e. we're doing well in the game and get a large end-of-episode return), we will increase the  $P_\theta(a | s_t)$  to a greater degree, and hence, on average over many episodes we will reinforce the actions that are good while the bad actions will get left behind. Since probabilities must sum to 1, if we increase the probability of a good action, that automatically steals probability mass from the other presumably less good actions. Without the use of this redistributive nature of probabilities, this whole scheme wouldn't work (i.e., everything good and bad would just tend to increase).

**Calculating future rewards.** We will multiply  $P_\theta(a_t | s_t)$  by the total reward (aka return) we received after this state. As mentioned earlier in the section, we can get the total reward by just summing the rewards (which is just equal to the number of time steps the episode lasted in Cartpole) and create a return array that starts with the episode duration and decrements by 1 until 1. If the episode lasted 5 time steps, then the return array would be  $[5, 4, 3, 2, 1]$ . This makes sense because our first action should be rewarded the most, since it is the least responsible for the pole falling and losing the episode. In contrast, the action right before the pole fell is the worst action and should have the smallest reward. But this is a linear decrement, we want to discount the rewards exponentially.

So we compute the discounted rewards. We make an array of  $\gamma_t$  by taking our  $\gamma$  parameter which may be set to 0.99, for example, and exponentiating it according to the distance from the end of the episode. For example, we start with `gamma_t = [0.99, 0.99, 0.99, 0.99, 0.99]`, then have arrange another array of exponents `exp = [1,2,3,4,5]` and compute `torch.power(gamma_t, exp)`, which will give us: `[1.0, 0.99, 0.98, 0.97, 0.96]`.

**Loss Function.** As we discussed previously, we make our loss function the negative log-probability of the action given the state, scaled by the reward returns. In PyTorch this defined as: `-1 * torch.sum(r * torch.log(preds))`. We compute the loss with the data we've collected for the episode, and run the torch optimizer to minimize the loss.

Let's run through some actual code.

#### Listing 4.5 Cartpole Policy Gradients

```
def discount_rewards(rewards, gamma=0.99):
    lenr = len(rewards)
    d_rewards = torch.pow(gamma, torch.arange(lenr)) * rewards #A
    d_rewards = (d_rewards - d_rewards.mean()) / (d_rewards.std() + 1e-07) #B
    return d_rewards
```

#A Compute exponentially decaying rewards  
#B Normalize the rewards by subtracting the mean and dividing out the standard deviation

Here we define a special function to compute the discounted rewards given an array of rewards that looks like `[50,49,48,47,...]` if the episode lasted 50 time steps. It essentially turns this linear sequence of rewards into an exponentially decaying sequence of rewards, e.g. `[50.0000, 48.5100, 47.0448, 45.6041, ...]` and then it subtracts the mean to make the sequence to be zero-centered. Lastly it normalizes the sequence by dividing by the standard deviation, to end up with a sequence like: `[1.9257, 1.8240, 1.7240, 1.6256, ...]`. The first half of the sequence will be positive numbers while the second half will be negative numbers. The reason for this normalization step is just to improve the learning efficiency and stability since it keeps the return values within the same range no matter how big the raw return is. It will still work without normalization, just not as reliably.

**Backpropagate.** Now that we have all the variables in our objective function we can calculate the loss and then backpropagate to adjust the parameters. We define our loss function below.

#### Listing 4.5 Cartpole Policy Gradients

```
def loss_fn(preds, r): #A
    return -1 * torch.sum(r * torch.log(preds)) #B
```

#A The loss function expects an array of action probabilities for the actions that were taken and the discounted rewards.  
#B It computes the log of the probabilities, multiplies by the discounted rewards, sums them all and flips the sign.

**Full training loop.** Initialize, collect experiences, calculate the loss from those experiences, backpropagate and repeat. Below defines the full training loop of our REINFORCE agent.

#### Listing 4.6 Cartpole Policy Gradients

```

env = gym.make('CartPole-v0')
MAX_DUR = 200
MAX_EPISODES = 500 #A
gamma_ = 0.99 #B
for episode in range(MAX_EPISODES):
    state1 = env.reset()
    done = False #C
    t = 0
    obs = [] #D
    actions = [] #E
    while not done: #F
        pred = model(torch.from_numpy(state1).float()) #G
        action = np.random.choice(np.array([0,1]), p=pred.data.numpy()) #H
        state2, reward, done, info = env.step(action) #I
        obs.append(state1) #J
        actions.append(action) #K
        state1 = state2
        t += 1
        if t > MAX_DUR: #L
            break;
    ep_len = len(obs) # M
    rewards = torch.arange(ep_len, 0, -1) #N
    preds = torch.zeros(ep_len) #O
    d_rewards = discount_rewards(rewards, gamma_) #P
    for j in range(ep_len): #Q
        state = obs[j]
        action = int(actions[j])
        pred = model(torch.from_numpy(state).float())
        preds[j] = pred[action] #R

    loss = loss_fn(preds, d_rewards)
    losses.append(loss)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

```

#A The number of training episodes

#B The discount factor

#C The done variable is set by the environment and indicates if the game is still in play

#D An array to store the state observations during an episode

#E An array to store the actions taken during an episode

#F Continue taking actions until episode is done

#G Call policy network model to produce predicted action probabilities

#H Sample an action from the probability distribution produced by the policy network

#I Take the action, receive new state and reward. The info variable is produced by the environment but is irrelevant

#J Add the state to our storage array of states

#K Add the action taken to the storage array of actions taken

#L If the episode is exceeding some defined max number of time steps, quit the episode.

#M Now the episode has ended, entering training steps. ep\_len is the duration of the episode

#N Create an array of rewards associated to each action taken

```
#O Setup an array to store the predicted probabilities of the actions actually taken
#P Compute the discounted rewards
#Q Revisit each state in the storage array of states
#R Fill in the array to store the probability of the actions actually taken, then run the optimizer
```

The code does exactly what we outlined earlier. We start an episode, use the policy network to take actions, record the states and actions we observe, and then once we break out of an episode, we have to re-compute the predicted probabilities to use in our loss function. If you run this code, you should be able to plot the episode duration against the episode number and will hopefully see a nicely increasing trend.

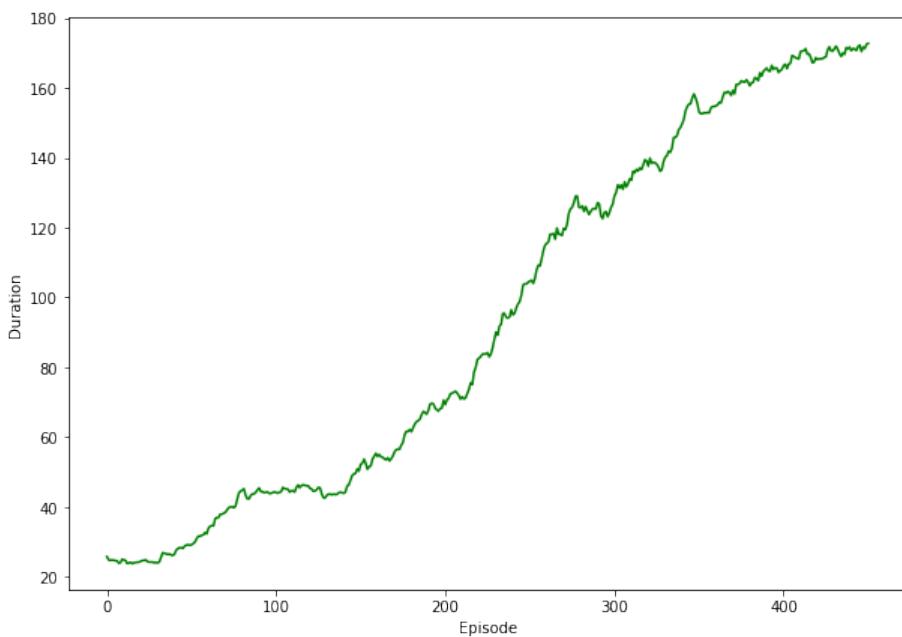


Figure 4.7: After training the policy network to 500 epochs, we get a plot that demonstrates the agent really is learning how to play cartpole. Note this is a moving average plot with a window of 50 to smooth the plot.

The agent learns how to play Cartpole! The nice thing about this example is that it should be able to train in under 1 minute on your laptop with just the CPU. The state of cartpole is just a 4-dimensional vector and our policy network is only 2 small layers, so it's much faster to train than the DQN we made to play Gridworld. OpenAI's documentation says that the game is considered "solved" if the agent can play an episode beyond 200 time-steps. Although the plot looks like it tops off at around 180, that's because it's a moving average plot, so there are episodes that it reaches 200. Also, we capped the episode duration at 200, so if you increase the cap it will be able to play even longer.

## 4.5 Summary and what's next

In this chapter we covered the REINFORCE algorithm, one of the earliest versions of the policy gradient algorithms.

REINFORCE is an effective and very simple way of training a policy function. But it's a little too simple. For Cartpole, it works very well since the state space is simple and there are only two actions. If we're dealing with an environment with many more possible actions, then reinforcing all of them each episode and hoping that on average it will only reinforce the good actions becomes less and less reliable. In the next two chapters we explore more sophisticated ways of training the agent.

# 5

## *Tackling more complex problems with Actor-Critic methods*

### This chapter covers:

- The limitations of the REINFORCE algorithm described in the previous chapter
- Introducing a critic to improve sample efficiency and decrease variance
- Using the advantage function to speed up convergence
- Speeding up the model by parallelizing training

In the previous chapter we introduced a vanilla version of a policy gradient method called REINFORCE. The REINFORCE algorithm is generally implemented as an episodic algorithm, meaning that we only apply to update to our model parameters after the agent has completed an entire episode (and collecting rewards along the way). Recall the policy is a function  $\pi:S \rightarrow P(a)$ , that is, a function that takes a state and returns a probability distribution over actions. Then we sample from this distribution to get an action, such that the most probable action (the “best” action) is most likely to be sampled. At the end of the episode, we compute the *return* of the episode, which is essentially the sum of the discounted rewards in the episode. If action 1 was taken in state A and resulted in a return of +10, then the probability of action 1 given state A will be increased a little, whereas if action 2 was taken in state A and resulted in a return of -20 then the probability of action 2 given state A will decrease. Essentially, we minimized this loss function:

$$\text{loss} = -\log(P(a | S)) * R$$

This says “minimize the (logarithm) of the probability of the action a given state S times the return R.” So if the reward is a big positive number and  $P(a_1 | S_A) = 0.5$ , for example, then

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

minimizing this loss will involve increasing this probability. So with REINFORCE we just keep sampling episodes (or trajectories more generally) from the agent and environment and periodically update the policy parameters by minimizing this loss.

### WHY LOG-PROBABILITIES?

Remember, we only apply the logarithm to the probability because a probability is bounded between 0 and 1 whereas the log-probability is bounded from  $-\infty$  (negative infinity) to 0, so given that numbers are represented by a finite number of bits, we can represent very small (close to 0) or very large (close to 1) probabilities without underflowing or overflowing the computer's numerical precision. Logarithms also have nicer mathematical properties that we don't need to cover, but that is why you'll almost always seen log-probabilities used in algorithms and machine learning papers even though we are conceptually interested in just the raw probabilities themselves.

By sampling a full episode, we get a pretty good idea of the "true value" of an action because we can see its downstream effects rather than just its immediate effect (which may be misleading due to randomness in the environment) and this full episode sampling is under the umbrella of Monte Carlo approaches. But not all environments are episodic, and sometimes we want to be able to make updates in an incremental or *online* fashion, i.e. making updates at regular intervals irrespective of what is going on in the environment. Our Deep Q-network did well in the non-episodic setting and could be considered an online-learning algorithm but it required an experience replay buffer in order to effectively learn.

The replay buffer was necessary because true online learning where parameter updates are made after each action is taken would be unstable due to the inherent variance (randomness) in the environment. An action taken once may by chance result in a big negative reward, but in expectation (average long-term rewards) it is a good action, so updating after a single action may result in erroneous parameter updates that will ultimately prevent adequate learning.

In this chapter, we introduce a new kind of policy gradient method called distributed advantage actor-critic (DA2C) that will have the online-learning advantages of DQN without a replay buffer and the advantages of policy methods where we can directly sample actions from the probability distribution over actions, thereby eliminating the need for choosing a policy (such as the epsilon-greedy policy) that we needed with DQN.

## 5.1 Combining the value and policy function

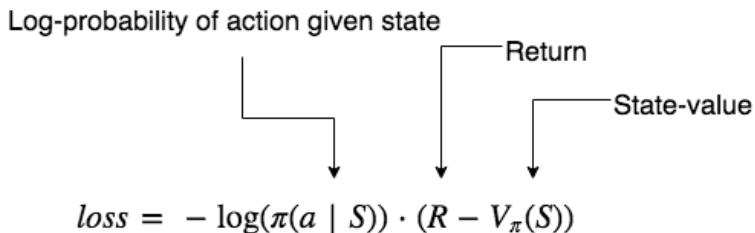
Q-learning (e.g. DQN) uses an action-value function to directly model the value (i.e. expected reward) of an action given a state. This is a very intuitive way of solving a Markov decision process (MDP) since we only observe states and rewards so it makes sense to predict the rewards and then just take actions that have high predicted rewards. On the other hand, we saw the advantage that direct policy learning (e.g. policy gradients) has, namely we get a true conditional probability distribution over actions,  $P(a|S)$ , that we can directly sample from to

take an action. Naturally someone decided it might be a good idea to combine these two approaches to get the advantages of both.

In building such a combined value-policy learning algorithm, we will start with the policy learner as the foundation. There are two challenges we want to overcome to increase the robustness of the policy learner:

1. We want to improve our sample efficiency by updating more frequently and
2. We want to decrease the variance of the reward we used to update our model.

These problems are related since the reward variance depends on how many samples we collect (i.e. more samples yields less variance). The idea behind a combined value-policy algorithm is to use the value learner to reduce the variance in the rewards used to train the policy. That is, instead of minimizing the REINFORCE loss that included direct reference to the observed return  $R$  from an episode, we instead add a baseline value such that the loss is now:



Where  $V(S)$  is the value of state  $S$ , which is the state-value function (only a function of the state) rather than an action-value function (a function of both state and action), although an action-value function could be used as well. This quantity  $V(S)-R$  is termed the **advantage**. Intuitively, the advantage quantity tells you how much better an action is relative to what you expected.

## **VALUES DEPEND ON THE POLICY**

Remember that the value function (state or action-value) implicitly depends on the choice of policy, hence we ought to write  $V_\pi(S)$ . To make it explicit, however we drop the  $\pi$  subscript for notational simplicity. The policy's influence on the value is crucial, since a policy of taking random actions all of the time would result in all states being more or less equally low value.

Imagine that we're training a policy on a simple game with discrete actions and a small discrete state-space such that we can use a vector where each position in the vector represents a distinct state and the element is the average rewards observed after that state is visited. This look-up table would be the  $V(S)$ . So we might sample action 1 from the policy and observe reward +10, but then we use our value look-up table and see that on average we get +4 after visiting this state, so the advantage of action 1 given this state is  $10 - 4 = +6$ . This means that when we took action 1, we got a reward that was significantly better than what we

expected based on past rewards from that state, which suggests that was a good action. Compare this to the case where we take action 1 and receive reward +10 but our value look-up table says we expected to see +15, so the advantage is  $10 - 15 = -5$ , suggesting this was a relatively bad action despite the fact we still received a reasonably large positive reward.

Rather than using a look-up table of course, we will use some sort of parameterized model such as a neural network that can be trained to predict expected rewards for a given state. So we want to simultaneously train a policy neural network and a state or action-value neural network.

Algorithms of this sort are called **actor-critic** methods, where “actor” refers to the policy since that’s where the actions are generated from and “critic” refers to the value function, since that’s what (in part) tells the actor how good its actions are. Since we’re using  $R-V(S)$  to train the policy rather than just  $V(S)$ , this is called **advantage actor-critic**.

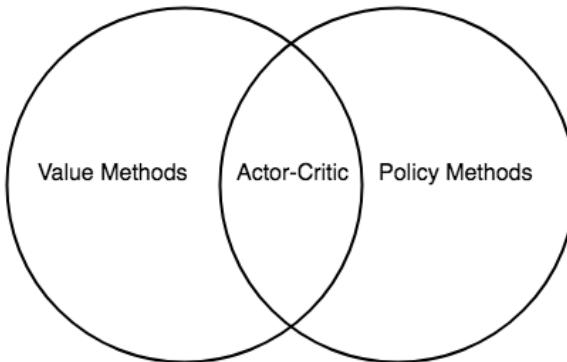


Figure 5.1

### VALUE BASELINE VERSUS ACTOR-CRITIC

Technically what we have described so far would not considered a true actor-critic method by some since we’re only using the value function as a baseline and not using it to “bootstrap” by making a prediction about a future state based on the current state. We will see how bootstrapping comes into play soon.

The policy network has the issue of a sensitive loss function that depends on the rewards collected at the end of the episode. If we naively tried to make online updates then, depending on the type of environment, we may never learn anything because the rewards may be too sparse.

In Gridworld, for example, the reward is 0 on every move except for the end of the episode. So the vanilla policy gradient method wouldn’t know what action to reinforce, since the vast majority of the time all the actions end up resulting in the same reward of 0. In contrast, a Q-network can learn decent  $q$ -values even when the rewards are sparse because it

**bootstraps.** When we say an algorithm bootstraps we mean it can make a prediction from a prediction.

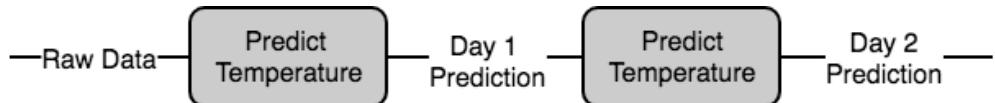


Figure 5.2

If I ask you what the weather is going to be like in 2 days from now, you might first predict what the weather will be tomorrow, and then base your 2-day prediction off that. You're bootstrapping. Of course, if your first prediction was bad, your second may be even worse, so bootstrapping introduces a source of **bias**. Bias is a systematic deviation from the true value of something, in this case, the true q-values. On the other hand, making predictions from predictions introduces a kind of "self-consistency" that results in lower **variance**. Variance is exactly what it sounds like, a lack of precision in the predictions, i.e. predictions that vary a lot. With the temperature example, if I make my day 2 temperature prediction based on my day 1 prediction, it will likely not be too far from my day 1 prediction.

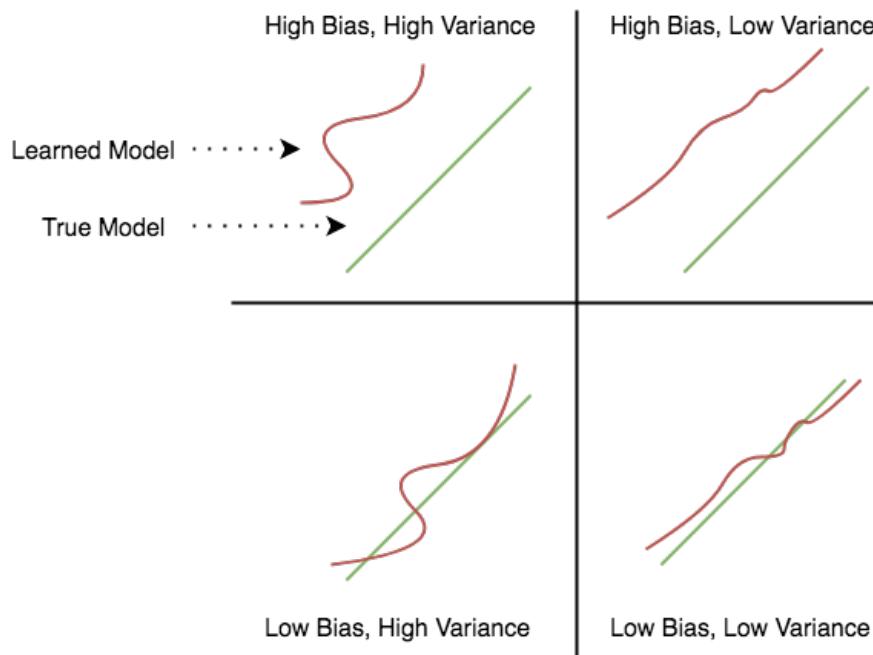
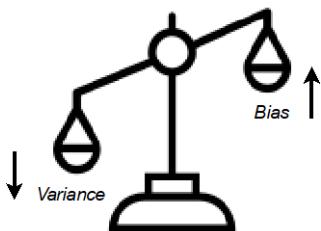


Figure 5.3: The bias-variance tradeoff is a fundamental machine learning concept that says any machine learning model will have some degree of systematic deviation from the true data distribution and some degree

of variance. You can try to reduce the variance of your model but it will always come at the cost of increased bias.

Bias and variance are key concepts relevant to basically all of machine learning, not just deep learning or deep reinforcement learning. Generally, if you reduce bias you will increase variance and vice versa. For example, if I ask you to predict the temperature for tomorrow and the next day, you could tell me a specific temperature: "The 2-day temperature forecast is 20.1 C and 20.5 C." This is a high-precision prediction, you've given me a temperature prediction to a tenth of a degree! But you don't have a crystal ball, so your prediction is almost surely going to be systematically off, hence biased toward whatever your prediction procedure involved. Or you could have told me: "The 2-day temperature forecast is 15-25 C and 18-27 C." In this case, your prediction has a lot of spread or variance since we're giving fairly wide ranges, but low bias, i.e. you have a good chance of the real temperatures falling in your intervals. This spread might be because your prediction algorithm didn't give undue weight to any of the variables used for prediction, hence not particularly biased in any direction. Indeed, machine learning models are often **regularized** by imposing a penalty on the size of the parameters during training. Regularization essentially means to modify your machine learning procedure in a way to mitigate overfitting.



**Figure 5.4: The bias-variance tradeoff.** Increasing model complexity can reduce bias but will increase variance. Reducing variance will increase bias.

So we want to combine the potentially high-bias, low-variance value prediction with the potentially low-bias, high-variance policy prediction to get something with moderate bias and variance, and therefore something that will work well in the online setting. The role of the critic is hopefully starting to become clear. The actor (policy network) will take a move, but the critic (state-value network) will tell the actor how good or bad the action was, rather than only using the potentially sparse raw reward signals from the environment. Thus the critic will be a term in the actor's loss function. The critic, just like with Q-learning, will learn directly from the reward signals coming from the environment, however the sequence of rewards will depend on the actions taken by the actor, so the actor affects the critic too, albeit more indirectly.

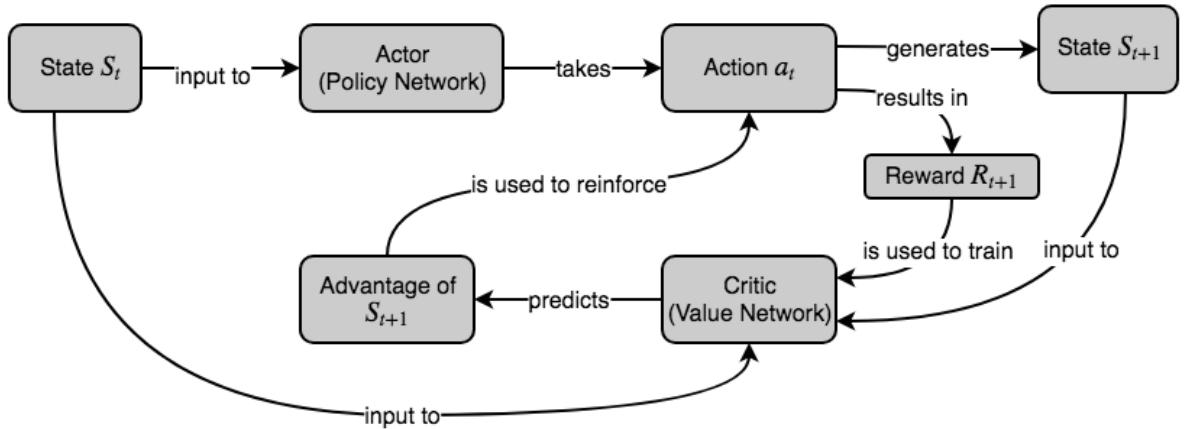


Figure 5.5: The general overview of actor-critic models. First, the actor predicts the best action and chooses the action to take, which generates a new state. The critic network computes the value of the old state and the new state. The relative value of  $S_{t+1}$  is called its advantage, and this is the signal used to reinforce the action that was taken by the actor.

The actor is trained in part by using signals coming from the critic, but how exactly do we train a state-value function as opposed to the action-value ( $Q$ )-functions we're more accustomed to? With action-values we computed the expected return (i.e. sum of future discounted rewards) for a given state-action pair. Hence we could predict whether a state-action pair would result in a nice positive reward, a bad negative reward, or something in between. But recall with our DQN, our  $Q$ -network returned separate action-values for each possible discrete action, so if we employ a reasonable policy like epsilon-greedy, then the state-value will essentially be the highest action-value. Thus the state-value function just computes this highest action-value rather than separately computing action-values for each action.

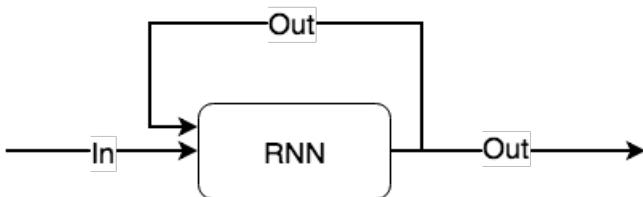
## 5.2 Distributed Training

We presented the advantage actor-critic part of the distributed advantage actor-critic (DA2C), now this section is about the distributed part. For virtually all deep learning models we do batch training where a random subset of our training data is batched together and we compute the loss for this entire batch before we backpropagate and do gradient descent. This is necessary because the gradients if we trained with a single datum at a time would have too much variance and the parameters would never converge on their optimal values. We need to average out the noise in a batch of data to get the real signal before updating the model parameters.

For example, if you're training an image classifier to recognize hand-drawn digits and you only train it one image at a time, the algorithm will think that the background pixels are just

as important as the digits in the foreground because it can only see the signal when averaged together with other images. The same concept applies in reinforcement learning, which is why we had to use an experience replay buffer with DQN.

Having a sufficiently large replay buffer is requires a lot of memory, and in some cases, a replay buffer is impractical. A replay buffer is possible when your reinforcement learning environment and agent algorithm follow the strict criteria of a Markov decision process, in particular, the Markov property. Recall the Markov property says that the optimal action for a state  $S_t$  can be computed without reference to any prior states  $S_{t+1}$ , i.e. there is no need to keep a history of previously visited states. For simple games this is the case, but for more complex environments, it may be necessary to remember the past in order to select the best option now.



**Figure 5.6:** A generic recurrent neural network (RNN) layer processes a sequence of data by incorporating its previous output with the new input. Thus an RNN will not work properly with single experiences in an experience replay buffer.

Indeed, in many complex games it is common to use recurrent neural networks (RNNs) like a long short-term memory (LSTM) network or a Gated Recurrent Unit (GRU). These RNNs can keep an internal state that can store traces of the past. They are particularly useful for natural language processing (NLP) tasks where keeping track of preceding words or characters is critical to being able to encode or decode a sentence. Experience replay doesn't work with an RNN unless your replay buffer stores entire trajectories or full episodes since the RNN is designed to process sequential data.

One solution is to run multiple copies of the agent in parallel each with separate instantiations of the environment. By distributing multiple independent agents across different CPU processes, we can collect a varied set of experiences and therefore get a sample of gradients that we can average together to get a lower variance mean gradient. This eliminates the need for experience replay and allows us to train an algorithm in a completely online fashion, visiting each state only once as it appears in the environment.

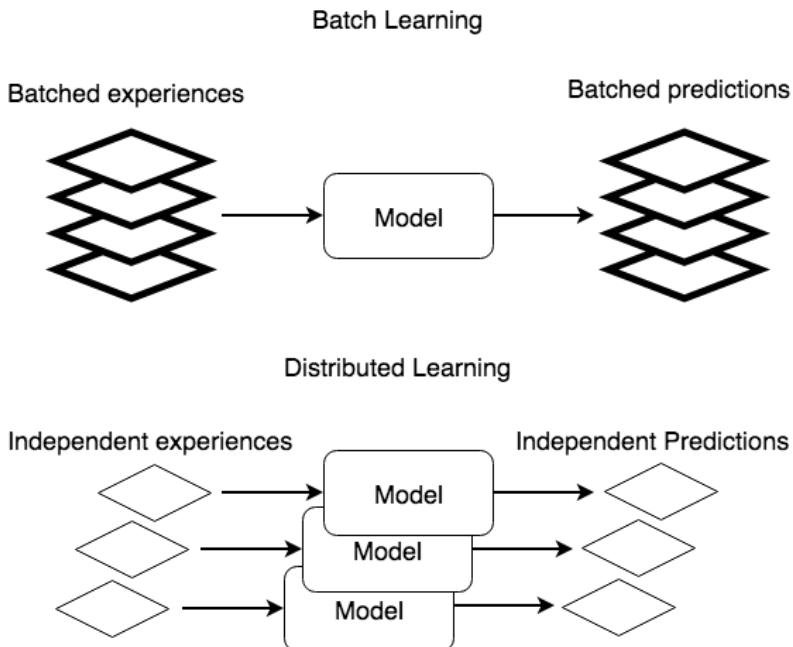


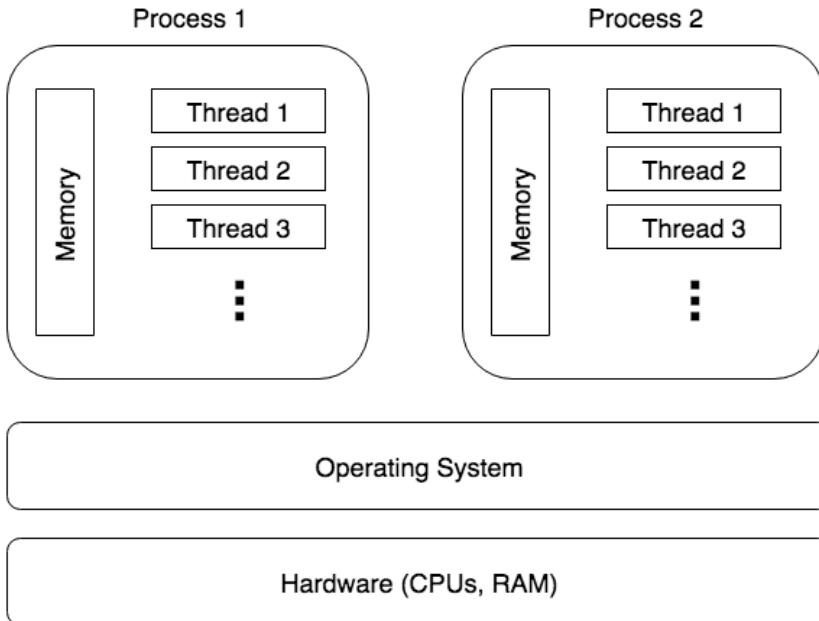
Figure 5.7: The most common form of training a deep learning model is to feed a batch of data together into the model to return a batch of predictions. Then we compute the loss for each prediction and average or sum all the losses before backpropagating and updating the model parameters. This averages out the noise present in each individual datum. Alternatively, we can run multiple models each taking a single piece of data and making a single prediction, backpropagate through each model to get the gradients and then sum or average the gradients before making any parameter updates.

### MULTIPROCESSING VERSUS MULTITHREADING

Modern desktop and laptop computers have central process units (CPUs) with multiple cores, which are independent processing units capable of running computations simultaneously. Therefore, if you can split a computation into pieces that can be computed separately and combined later, you can get dramatic speedups. The operating system software abstracts the physical CPU processors into virtual processes and threads. A process contains its own memory space and threads run within a single process.

Multithreading is like when people multi-task, i.e. they can work on only one thing at a time but switch between different tasks while one task is idle; therefore tasks are not truly simultaneously performed with multithreading. Multithreading is really effective when your task requires a lot of input/output operations such as reading and writing data to the hard disk. When data is being read into RAM from the hard disk, computation on the CPU is idle waiting for the required data, so during that time the operating system can use the idle CPU time to work on a different task and then switch back when the I/O operation is done.

Running machine learning models generally does not require I/O operations; machine learning is limited by computation speed, therefore ML benefits from true simultaneous computation with multiprocessing.



**Figure 5.8:** Processes are an abstraction of the underlying CPU hardware created by the operating system. If you have two CPUs, then you can run two simultaneous processes. However, the operating system will let you spawn more than 2 virtual processes and it will figure out how to multi-task between them. Each process has its own memory address space and can have multiple threads. A thread is a task that can be run and while one task is waiting for an external process to finish (such as an input/output operation), the OS can let another thread run. This maximizes the usage of whatever CPUs you have.

Large machine learning models all but require graphics processing units (GPUs) to perform efficiently, however, distributed models on multiple CPUs can be competitive in some cases. Python provides a library called `multiprocessing` that makes multiprocessing very easy. Additionally, PyTorch wraps this library and has a method for allowing a model's parameters to be shared across multiple processes. Let's see a simple example of multiprocessing.

A contrived simple example is that we have an array with the numbers 0, 1, 2, 3 ... 64 and we want to square each number. Since squaring a number does not depend on any other numbers in the array, we can easily parallelize this across multiple processors.

#### **Listing 5.1: Introduction to multiprocessing**

```
import multiprocessing as mp
import numpy as np
def square(x): #A
    return np.square(x)
x = np.arange(64) #B
>>> print(x)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
```

```

    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
    34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
    51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63])
>>> mp.cpu_count()
8
pool = mp.Pool(8) #C
squared = pool.map(square, [x[8*i:8*i+8] for i in range(8)]) #D
>>> squared
[array([ 0,  1,  4,  9, 16, 25, 36, 49]),
 array([ 64,  81, 100, 121, 144, 169, 196, 225]),
 array([256, 289, 324, 361, 400, 441, 484, 529]),
 array([576, 625, 676, 729, 784, 841, 900, 961]),
 array([1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521]),
 array([1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209]),
 array([2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025]),
 array([3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969])]

```

#A This function just takes an array and squares each element

#B Setup an array with a sequence of numbers

#C Setup a multiprocessing processor pool with 8 processes

#D Use the pool's map function to apply the square function to each array in the list and return the results in a list

Here we simply define a function square that takes an array and squares it. This is the function that will get distributed across multiple processes. We create some toy data that is simply the list of numbers 0 to 63 and rather than sequentially squaring them in a single process, we chop up the array into 8 pieces and compute the squares for each piece independently on a different processor.

You can see how many hardware processors your computer has using the `mp.cpu_count()` function, you can see we have 8. Many modern computers may have say 4 independent hardware processors but will have twice as many “virtual” processors via something called hyperthreading. Hyperthreading is a performance trick by some processors that can allow two processes to run essentially simultaneously on one physical processor. It is important not to create more processes than there are CPUs on your machine as the additional processes will essentially function as threads where the CPU will have to rapidly switch between processes.

We setup a processor pool of 8 processes with `mp.Pool(8)` and then use `pool.map` to distributed the square function across the 8 pieces of data. You can see we get a list of 8 arrays with all their elements squared just as we wanted. Processes will return as soon as their complete so the order of the elements in the returned list will be unpredictable and may not always been in the order they were mapped.

We’re going to need a bit more control over our processes than a processor Pool allows, so we will create and start a bunch of processes manually.

### **Listing 5.2: Manually starting individual processes**

```

def square(i, x, queue):
    print("In process {}".format(i))
    queue.put(np.square(x))
processes = [] #A
queue = mp.Queue() #B

```

```

x = np.arange(64) #C
for i in range(8): #D
    start_index = 8*i
    proc = mp.Process(target=square,args=(i,x[start_index:start_index+8], queue))
    proc.start()
    processes.append(proc)

for proc in processes: #E
    proc.join()

for proc in processes: #F
    proc.terminate()
results = []
while not queue.empty(): #G
    results.append(queue.get())
>>> results
[array([ 0,  1,  4,  9, 16, 25, 36, 49]),
 array([256, 289, 324, 361, 400, 441, 484, 529]),
 array([ 64,   81, 100, 121, 144, 169, 196, 225]),
 array([1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209]),
 array([576, 625, 676, 729, 784, 841, 900, 961]),
 array([1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521]),
 array([2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025]),
 array([3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969])]

#A Setup a list to store a reference to each process
#B Setup a multiprocessing queue, a data structure than can be shared across processes
#C Setup some toy data, a sequence of numbers
#D Start 8 processes with the square function as the target and an individual piece of data to process
#E Wait for each process to finish before returning to the main thread
#F Terminate each process
#G Convert the multiprocessing queue into a list

```

This is more code but functionally the same as what we did before with the `Pool`, but now its easy to share data between processes using special shareable data structures in the multiprocessing library and to have more control over the processes. We modify our `square` function a little to accept an integer representing the process ID, the array to square, and a shared global data structure called a `queue` that we can put data into and extract from using the `.get()` method.

Next we setup a list to hold the instances of our processes, the shared queue object and create our toy data as before. We then enter a loop to create (in our case) 8 processes and start them using the `.start()` method, then we add them to our `processes` list so we can access them later. Next we run through the `processes` list and call each `processes .join()` method, this makes it so we wait to return anything until all the processes have finished. Then we call each `processes .terminate()` method to ensure it is killed. Lastly, we collect all the elements of the queue into a list and print it out. The results look the same as with the process Pool except their in a random order. That's really all there is to distributing a function across multiple CPU processors.

## 5.3 Advantage Actor-Critic

Now that we know how to distribute computation across processes, we can get back to the real reinforcement learning. In this section we put together the pieces of the full distributed advantage actor-critic model. To allow fast training and to compare to the previous chapter, we will also use the CartPole game as our test environment, however you can easily adapt the algorithm to a more difficult game such as Pong in OpenAI Gym and you can find such an implementation at this chapter's GitHub page:

< <https://github.com/DeepReinforcementLearning/DeepReinforcementLearningInAction> >

While we presented the actor and critics as two totally separate functions, we can actually combine them into a single neural network with two output "heads." That is, instead of a normal neural network that returns a single vector, ours will return two different vectors, one for the policy and one for the value. This allows for some parameter sharing between the policy and value that can make things more efficient since some of the information needed to compute values is also probably useful to predicting the best action for the policy. But if a two-headed neural network seems too exotic right now, go ahead and write two separate neural networks, it will work just fine.

Let's see some pseudocode for what the algorithm looks like and then we'll translate to Python.

**Listing 5.3: Pseudocode for Online Advantage Actor-Critic**

```
gamma = 0.9
for each epoch do {
    state = environment.get_state()
    value = critic(state)
    policy = actor(state)
    action = policy.sample()
    next_state, reward = environment.take_action(action)
    value_next = critic(next_state)
    advantage = reward + (gamma * value_next - value)
    loss = -1 * policy.logprob(action) * advantage
    minimize(loss)
```

This is very simplified pseudocode but it gets the main idea across. The important part to point out is the advantage calculation. Consider the case where we take action 1 receive reward +10, the value prediction is +5, and the value prediction for the next state is +7. We discount the value of the next state (since future predictions are always less valuable than currently observed reward) by the gamma discount factor, so our  $advantage = 10 + 0.9*7 - 5 = 10 + (6.3 - 5) = 10 + 1.3 = +11.3$ . Since the difference between the next state value and the current state value is positive, it increases the overall value of the action we just took so we will reinforce it more. Notice that the advantage function *bootsraps* because it computes a value for the current state and action based on predictions for a future state.

In this chapter we're going to use our DA2C model on Cartpole again, which is episodic, so if we do a full Monte Carlo update (i.e. update after the full episode is complete) then

`value_next` will always be 0 for the last move (since there is no next state if the episode is over), thus the advantage term actually reduces to `advantage = reward - value`, which is the value baseline we discussed at the beginning of the chapter. The full advantage expression  $A = r_{t+1} + \gamma * v(s_{t+1}) - v(s_t)$  is used when we do online or **N-step learning**.

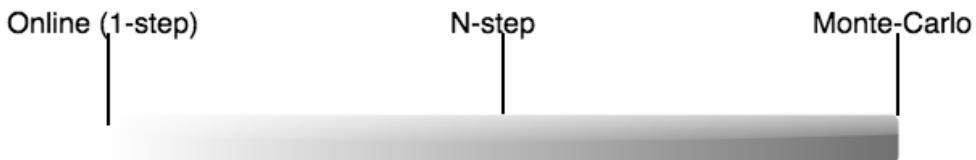


Figure 5.9

N-step learning is what's in between fully online learning and waiting for a full episode (i.e. Monte Carlo) before updating. As the name suggests, we accumulate rewards over N-steps and then compute our loss and backpropagate. The number of steps can be anywhere from 1 (which reduces to fully online learning) to the maximum number of steps in the episode (Monte Carlo), but usually we pick something in between to get the advantages of both. We will first show the episodic actor-critic algorithm and then we will adapt it to N-step with N set to 10.

Alright, we're ready to get to the coding of an actor-critic model to play Cartpole. As usual, let's sketch out the sequence of steps.

1. Setup our actor-critic model, a two-headed model (alternatively you can setup two independent actor and critic networks). The model accepts a Cartpole state as input, which is a vector of 4 real numbers. The actor head is just like the policy network (actor) from the previous chapter, so it outputs a 2-dimensional vector representing a discrete probability distribution over the 2 possible actions. The critic outputs a single number representing the state-value. The critic is denoted  $v(s)$  and the actor is denoted  $\pi(s)$ . Remember that  $\pi(s)$  returns the log-probabilities for each possible action, in our case 2 actions.
2. While in current episode
  - 2.1 Define hyper parameter:  $\gamma$  (gamma, discount factor)
  - 2.2 Start a new episode, in initial state  $s_t$
  - 2.3 Compute value  $v(s_t)$  and store in list.
  - 2.4 Compute  $\pi(s_t)$ , store in list, sample and take action  $a_t$
  - Receive new state  $s_{t+1}$  and reward  $r_{t+1}$ . Store reward in list.
3. Training.
  - 3.1 Initialize  $R=0$ . Loop through rewards in reverse order to generate returns

$$R = r_t + \gamma * R$$

3.2 Minimize the actor loss:

$$-\lambda * \gamma_t * (R - v(s_t)) * \pi(a | s)$$

3.3 Minimize critic loss:

$$(R - v)^2$$

4. Repeat for new episode.

#### **Listing 5.4 Cartpole Actor-Critic Model**

```
import torch
from torch import nn
from torch import optim
import numpy as np
from torch.nn import functional as F
import gym
import torch.multiprocessing as mp #A
class ActorCritic(nn.Module): #B
    def __init__(self):
        super(ActorCritic, self).__init__()
        self.l1 = nn.Linear(4,25)
        self.l2 = nn.Linear(25,50)
        self.actor_lin1 = nn.Linear(50,2)
        self.l3 = nn.Linear(50,25)
        self.critic_lin1 = nn.Linear(25,1)
    def forward(self,x):
        x = F.normalize(x,dim=0)
        y = F.relu(self.l1(x))
        y = F.relu(self.l2(y))
        actor = F.log_softmax(self.actor_lin1(y),dim=0) #C
        c = F.relu(self.l3(y.detach()))
        critic = torch.tanh(self.critic_lin1(c)) #D
        return actor, critic #E
```

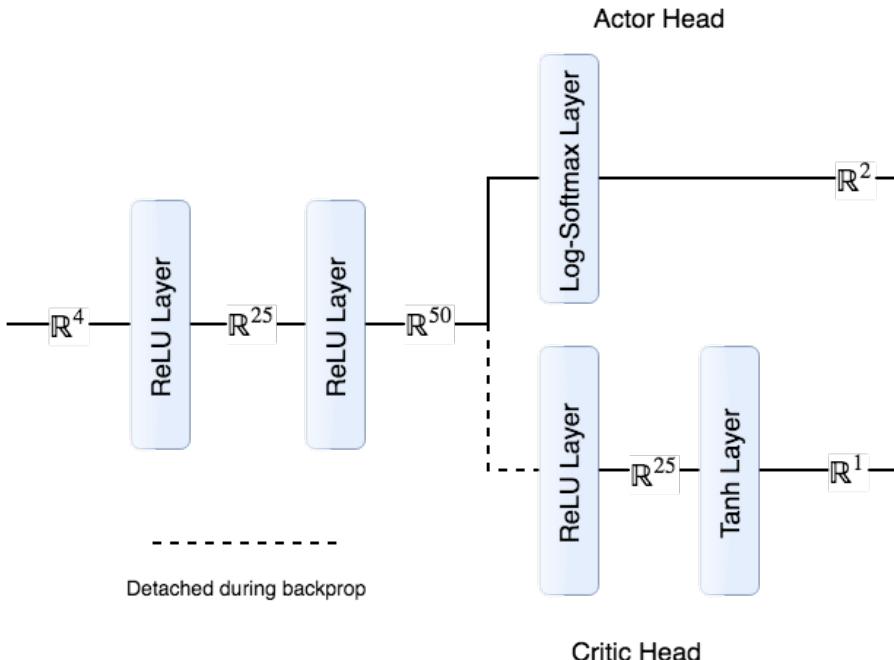
#A PyTorch wraps Python's built-in multiprocessing library and the API is the same

#B Define a single combined model for the actor and critic

#C The actor head returns the log-probabilities over the 2 actions

#D The critic returns a single number bounded between -1 and +1

#E We simply return the actor and critic results as a tuple



**Figure 5.10:** This is an overview of the architecture for our two-headed actor-critic model. It's overall very simple with two shared linear layers and then a branching point where the output of the first two layers is sent to a log-softmax layer of the actor head and also sent to another ReLU layer of the critic head before finally passing through a tanh layer. This model returns a 2-tuple of tensors rather than a single tensor.

For CartPole, we have a fairly simple neural network other than having two output heads. First we normalize the input so that the state values are all within the same range, then the first two layers are ordinary linear layers with the ReLU activation function. Then we fork the model into two paths. The first path is the actor head that takes the output of layer 2 and applies another linear layer and then the `log_softmax(...)` function. The `log_softmax` is logically equivalent to doing `log(softmax(...))` but the combined function is more numerically stable since if you compute the functions separately you might end up with overflowed or underflowed probabilities after the `softmax`.

The second path is the critic head, which applies a linear layer and ReLU to the output of layer 2, but notice we call `y.detach()`, which detaches the `y` node from the graph and thus the critic loss won't backpropagate and modify the weights in layer 1 and 2, only the actor will cause the weights to be modified. This is not a crucial feature but it improves the training time a bit by reducing the conflict between what the actor and critic want. With two-headed models, it often makes sense to make one head the dominant head and control most of the parameters by detaching the other head from the first several layers. Lastly, the critic applies

another linear layer with the tanh activation function that bounds the output to the interval  $(-1,1)$ , which is perfect for Cartpole since the rewards are  $+1$  and  $-1$ .

### **Listing 5.5: Distributing the training**

```

MasterNode = ActorCritic() #A
MasterNode.share_memory() #B
processes = [] #C
params = {
    'epochs':1000,
    'n_workers':7,
}
counter = mp.Value('i',0) #D
for i in range(params['n_workers']):
    p = mp.Process(target=worker, args=(i,MasterNode,counter,params)) #E
    p.start()
    processes.append(p)
for p in processes: #F
    p.join()
for p in processes: #G
    p.terminate()

print(counter.value,processes[1].exitcode) #H

#A Create a global, shared actor-critic model
#B The share_memory() method will allow the parameters of the model to be shared across processes rather than
copied
#C Setup a list to store our instantiated processes
#D Shared global counter using multiprocessing's built-in shared object. The 'i' parameter indicates the type as integer.
#E Start a new process that runs the worker function
#F "Join" each process to wait for it to finish before returning to main process
#G Make sure each process is terminated
#H Print the global counter value and the first process' exit code (should be 0)

```

This is exactly the same setup we had when demonstrating how to split up an array across multiple processes, except this time we're going to be running a function called `worker` that will run our CartPole reinforcement learning algorithm.

### **Listing 5.6: The main training loop**

```

def worker(t, worker_model, counter, params):
    worker_env = gym.make("CartPole-v1")
    worker_env.reset()
    worker_opt = optim.Adam(lr=1e-4, params=worker_model.parameters()) #A
    worker_opt.zero_grad()
    for i in range(params['epochs']):
        worker_opt.zero_grad()
        values, logprobs, rewards = run_episode(worker_env,worker_model) #B
        actor_loss,critic_loss,eplen =
        update_params(worker_opt,values,logprobs,rewards) #C
        counter.value = counter.value + 1 #D

```

#A Each process runs its own isolated environment and optimizer but share the model

#B The `run_episode` function plays an episode of the game, collecting data along the way

```
#C We use the collected data from run_episode to run one parameter update step
#D counter is a globally shared counter between all the running processes
```

The `worker` function is the function that each individual process will run separately. Each worker (i.e. process) will create its own CartPole environment and its own optimizer but will share the actor-critic model, which is passed in as an argument to the function. Since the model is shared, whenever a worker updates the model parameters it is also updated for all the other workers.

Since each worker is spawned in a new process that has its own memory, all the data the worker should be passed in as an argument to the function explicitly. This also prevents bugs.

### **Listing 5.7: Running an episode**

```
def run_episode(worker_env, worker_model):
    state = torch.from_numpy(worker_env.env.state).float() #A
    values, logprobs, rewards = [],[],[] #B
    done = False
    j=0
    while (done == False): #C
        j+=1
        policy, value = worker_model(state) #D
        values.append(value)
        logits = policy.view(-1)
        action_dist = torch.distributions.Categorical(logits=logits)
        action = action_dist.sample() #E
        logprob_ = policy.view(-1)[action]
        logprobs.append(logprob_)
        state_, _, done, info = worker_env.step(action.detach().numpy())
        state = torch.from_numpy(state_).float()
        if done: #F
            reward = -10
            worker_env.reset()
        else:
            reward = 1.0
        rewards.append(reward)
    return values, logprobs, rewards
```

```
#A Convert the environment state from a numpy array to a PyTorch Tensor
#B Create lists to store the computed state-values (critic), log-probabilities (actor), and rewards
#C Keep playing the game until the episode ends
#D Compute the state-value and log-probabilities over actions
#E Using the actor's log-probabilities over actions, create and sample from a categorical distribution to get an action
#F If the last action caused the episode to end, set the reward to -10 and reset the environment
```

The `run_episode` function just runs through a single episode of CartPole and collects the computed state-values from the critic, log-probabilities over actions from the actor, and rewards from the environment. We store these in lists and use them to compute our loss function later. Since this is an actor-critic method and not Q-learning, we take actions by directly sampling from the policy rather than arbitrarily choosing a policy like epsilon-greedy in

Q-learning. There's nothing too out of the ordinary in this function, so let's move on to the updating function.

### **Listing 5.8: Computing and minimizing the loss**

```
def update_params(worker_opt, values, logprobs, rewards, c1c=0.1, gamma=0.95):
    rewards = torch.Tensor(rewards).flip(dims=(0,)).view(-1) #A
    logprobs = torch.stack(logprobs).flip(dims=(0,)).view(-1)
    values = torch.stack(values).flip(dims=(0,)).view(-1)
    Returns = []
    ret_ = torch.Tensor([0])
    for r in range(rewards.shape[0]): #B
        ret_ = rewards[r] + gamma * ret_
        Returns.append(ret_)
    Returns = torch.stack>Returns().view(-1)
    Returns = F.normalize(Returns, dim=0)
    actor_loss = -1*logprobs * (Returns - values.detach()) #C
    critic_loss = torch.pow(values - Returns, 2) #D
    loss = actor_loss.sum() + c1c*critic_loss.sum() #E
    loss.backward()
    worker_opt.step()
    return actor_loss, critic_loss, len(rewards)
```

#A We reverse the order of the rewards, logprobs and values\_arrays and call .view(-1) to make sure they're flat

#B For each reward (in reverse order), we compute the return value and append it to a returns array

#C We need to detach the values\_tensor from the graph to prevent backpropagating through the critic head

#D The critic just attempts to learn to predict the return

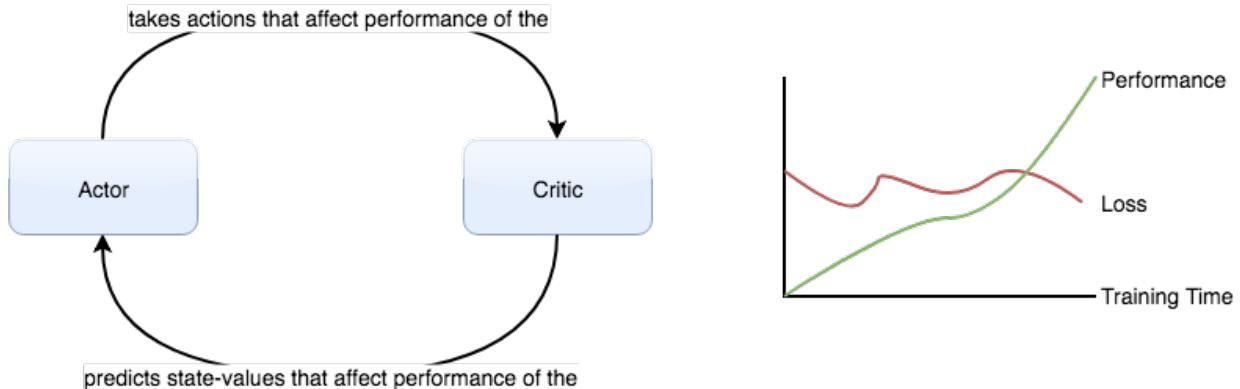
#E We sum the actor and critic losses to get an overall loss. We scale down the critic loss by the `c1c` factor.

The `update_params` function is where all the action is and what sets distributed advantage actor-critic apart from the other algorithms we've learned so far. First we take the lists of rewards, log-probabilities, and state-values and convert them to PyTorch tensors. We then reverse their order because we want to consider the most recent action first, and then we make their they are flattened 1-D arrays by calling the `.view(-1)` method.

The `actor_loss` is computed just we described with math using the advantage (technically baseline since there's no bootstrapping) rather than raw reward. Crucially, we must detach the `values` tensor from the graph when we use in the `actor_loss`, otherwise we will backpropagate through the actor and critic heads, and we only want to update the actor head. The critic loss is a simple squared error between the state-values and the returns and we make sure *not* to detach here since we want to update the critic head. Then we sum the actor and critic losses to get the overall loss. We scale down the critic loss by multiplying by 0.1 because we want the agent to learn faster than the critic. We return the individual losses and the length of the rewards tensor (which indicates how long the episode lasted) to monitor their progress during training.

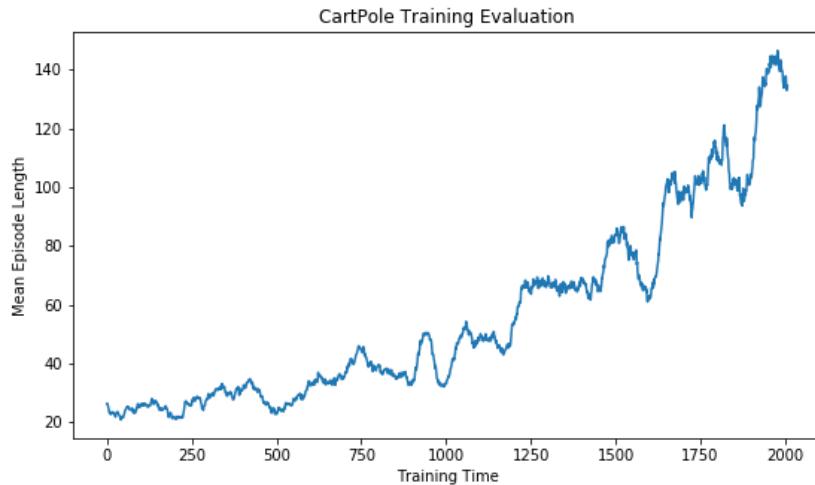
Put it all together and run it and you'll get a trained CartPole agent within 1 minute on a modern computer running just on a few CPU cores. If you plot the loss over time for this it probably won't be a nice down-trending line like you hope because the actor and critic are in competition with one another. The critic is incentivized to model the returns as best as it can

(and the returns depend on what the actor does), but the actor is incentivized to beat the expectations of the critic. If the actor improves faster than the critic, then the critic's loss will be high and vice versa, so there is a somewhat adversarial relationship between the two.



**Figure 5.11:** The actor and critic have a bit of an adversarial relationship since the actions that the agent take affect the loss of the critic, and the critic makes predictions of state-values that get incorporated into the return that affects the training loss of the actor. Hence the overall loss plot may look chaotic despite the fact that the agent is indeed increasing in performance.

Adversarial training like this is a very powerful technique in many areas of machine learning, not just reinforcement learning. For example, generative adversarial networks (GANs), are an unsupervised method to generate realistic-looking synthetic samples of data from a training data set using a pair of models that function similarly to an actor and critic. In fact, we will build an even more sophisticated adversarial model in Chapter 8 so you will learn more then. But the take home here is that if you're using an adversarial model, the loss will be largely uninformative (unless it goes to 0 or explodes toward infinity, then something is probably wrong). You have to rely on actually evaluating the objective you care about, in our case how well the agent is performing in the game. Here is the plot of average episode length during the first 45 seconds of training:



**Figure 5.12** The mean episode length over training time for our Monte Carlo distributed advantage actor-critic model. This model is not considered a true critic since the critic is not bootstrapping during training. As a result the training performance has high variance.

## 5.4 N-Step Actor-Critic

In the last section we implemented distributed advantage actor-critic, except that we trained in “Monte Carlo mode,” i.e. we ran a full episode before updating the model parameters. While that makes total sense for a simple game like CartPole, most of the time we want to be able to make more frequent updates. We briefly touched on N-step learning before, but to re-iterate, it means we simply calculate our loss and update the parameters after N-steps where N is whatever we choose it to be. If N is 1 then it is fully online learning or we could set N to be very large and then it will be Monte Carlo again. The sweet spot is somewhere in-between.

With Monte Carlo full-episode learning, we don’t take advantage of bootstrapping since there’s nothing to bootstrap. We do bootstrap in online learning like we did with DQN, however, with 1-step learning the bootstrap may introduce a lot of bias. This bias may be harmless if it still pushes our parameters in the right direction, but in some cases the bias can be so off that we never move in the right direction.

This is why N-step learning is usually better than 1-step (online) learning, the target value for the critic is more accurate so the critic’s training will be more stable and will be able to produce less biased state-values. With bootstrapping, we’re making a prediction from a prediction, so the predictions will be better if you’re able to collect more data before making them. And we like bootstrapping because it improves sample efficiency since you don’t need to see as much data (e.g. frames in a game) before updating the parameters in the right direction.

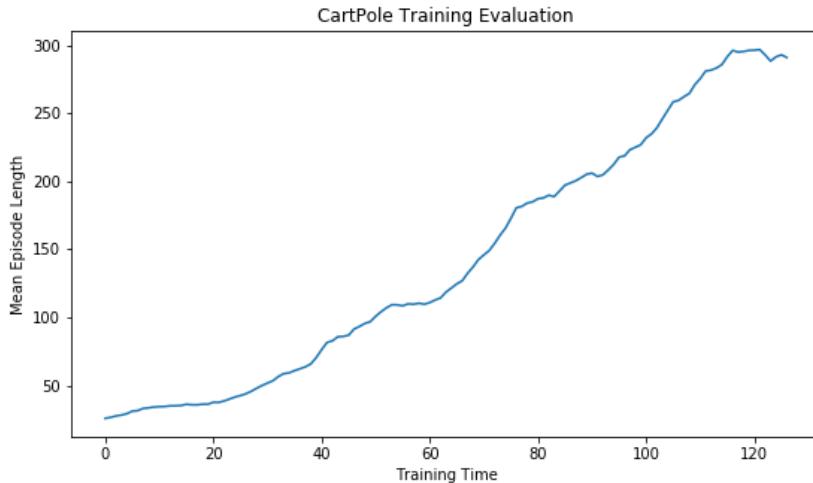
So let's see how to modify our code to do N-step learning. The only function we need to modify is the `run_episode` function. We need to change it run for only N-steps rather than wait for the episode to finish. If the episode finishes before N-steps, then the last return value will be set to 0 (since there is no next state when the game is over) as it was in the Monte Carlo case. However, if the episode hasn't finished by N-steps, then we use the last state-value as our prediction for what the return would have been had we kept playing, that's where the bootstrapping happens. Without bootstrapping, the critic is just trying to predict the future returns from a state and it gets the actual returns as training data. With bootstrapping, it is still trying to predict future returns but it is doing so in part by using its own prediction about future returns (since the training data will include its own prediction).

### **Listing 5.9: N-step training with CartPole**

```
def run_episode(worker_env, worker_model, N_steps=10):
    raw_state = np.array(worker_env.env.state)
    state = torch.from_numpy(raw_state).float()
    values, logprobs, rewards = [],[],[]
    done = False
    j=0
    G=torch.Tensor([0]) #A
    while (j < N_steps and done == False): #B
        j+=1
        policy, value = worker_model(state)
        values.append(value)
        logits = policy.view(-1)
        action_dist = torch.distributions.Categorical(logits=logits)
        action = action_dist.sample()
        logprob_ = policy.view(-1)[action]
        logprobs.append(logprob_)
        state_, _, done, info = worker_env.step(action.detach().numpy())
        state = torch.from_numpy(state_).float()
        if done:
            reward = -10
            worker_env.reset()
        else: #C
            reward = 1.0
            G = value.detach()
        rewards.append(reward)
    return values, logprobs, rewards, G
```

#A The variable G refers to the return. We initialize to 0  
#B Play game until N-steps or when episode is over  
#C If episode is not done, set return to the last state-value

The only things we've changed are the conditions for the while loop (exit by N-steps) and we've set the return to be the state-value of the last step if the episode is not over, thereby enabling bootstrapping. You might be surprised at how much more efficient N-step learning is, here's the plot of episode length over the first 45 seconds of training for this model:



**Figure 5.13:** Performance plot for distributed advantage actor-critic with true N-step bootstrapping. Compared to our previous Monte Carlo algorithm, the performance is much smoother due to the more stable critic.

Notice that the N-step model starts getting better right away and reaches an episode length of 300 after just 45 seconds, compared to only about 140 for the Monte Carlo version. Also notice that this plot is much smoother than the Monte Carlo one. Bootstrapping reduces the variance in the critic and allows it to learn much more rapidly than Monte Carlo.

As a concrete example, imagine the case where you get 3-step rewards of [1,1,-1] for episode 1 and then [1,1,1] for episode 2. The overall return for episode 1 is 0.01 (with  $\gamma=0.9$ ) and 1.99 for episode 2; that's two orders of magnitude difference in return just based on the random outcome of the episode early in training. That's a lot of variance. Compare that to the same case except with (simulated) bootstrapping so that the return for each of those episodes also includes the bootstrapped predicted return. With a bootstrapped return prediction of 1.0 for both (hence assuming the states are similar), the calculated returns are 0.99 and 2.97, which are much closer than without bootstrapping. You can reproduce this toy example with the following code:

#### **Listing 5.10: Returns with and without bootstrapping**

```
#Simulated rewards for 3 steps
r1 = [1,1,-1]
r2 = [1,1,1]
R1,R2 = 0.0,0.0
#No bootstrapping
for i in range(len(r1)-1,0,-1):
    R1 = r1[i] + 0.99*R1
for i in range(len(r2)-1,0,-1):
    R2 = r2[i] + 0.99*R2
print("No bootstrapping")
```

```

print(R1,R2)
#With bootstrapping
R1,R2 = 1.0,1.0
for i in range(len(r1)-1,0,-1):
    R1 = r1[i] + 0.99*R1
for i in range(len(r2)-1,0,-1):
    R2 = r2[i] + 0.99*R2
print("With bootstrapping")
print(R1,R2)
>>> No bootstrapping
0.01000000000000009 1.99
With bootstrapping
0.9901 2.9701

```

## 5.5 Summary and what's next

In this chapter we covered the advantage actor-critic algorithm and made it run on multiple processes, hence we called it distributed advantage actor-critic (DA2C). In the plain policy gradient method of the previous chapter, we only trained a policy function that would output a probability distribution over all the actions such that the predicted best action would be assigned the highest probability. Unlike Q-learning where a target value is learned, the policy function is directly reinforced to increase or decrease the probability of the action taken depending on the reward. Many times the same action may produce opposite results in terms of reward, causing high variance in the training.

To mitigate this, we introduce a critic model (or in this chapter we used a single, two-headed model) that reduces the variance of the policy function updates by directly modeling the state-value. This way, if the actor (policy) takes an action and gets an unusually big or small reward, the critic can moderate this big swing and prevent an unusually large (and possibly destructive) parameter update to the policy). This also leads to the notion of advantage where instead of training the policy based on raw return (average accumulated rewards), we train based on how much better (or worse) the action was compared to what we predicted it would be from the critic. This is helpful since if two actions both lead to the same positive reward, we will naively assume their equivalent actions, but if we compare to what we expected to happen and one reward performed much better than anticipated, that action should be reinforced more.

As with the rest of deep learning methods, we generally must use batches of data in order to effectively train. Training a single example, a time introduces too much noise and the training will likely never converge. To introduce batch training with Q-learning we used an experience replay buffer that we could randomly select batches of previous experiences. We could have used experience replay with actor-critic, but it is more common to use distributed training with actor-critic (and to be clear, Q-learning can also be distributed). One reason is that in many cases we want to use a recurrent neural network (RNN) layer as part of our reinforcement learning model in cases where keeping track of prior states is necessary or helpful in achieving the goal. But RNNs need a sequence of temporally related examples, and experience replay relies on a batch of independent experiences. We could store entire

trajectories (sequences of experiences) in a replay buffer, but that just adds additional complexity. Instead, with distributed training with each process running online with its own environment, the models can easily incorporate RNNs.

While not covered here, there's another way to train an online actor-critic algorithm besides distributed training and that's by simply utilizing multiple copies of your environment and then batching together the states from each independent environment, feeding it into a single actor-critic model that will then produce independent predictions for each environment. This is a viable alternative to distributed training when the environments are not expensive to run. If your environment is a complicated, high-memory and computer intensive simulator, then it's probably going to be very slow to run multiple copies of it in a single process, so in that case use a distributed approach.

After this chapter, we have covered what we consider to be the most foundational parts of reinforcement learning today. You should be comfortable with the basic mathematical framework of reinforcement learning as a Markov decision process (MDP) and you should be able to implement Q-learning, plain policy gradient, and actor-critic models. If you've followed so far, you should have a good base to be able to tackle many other reinforcement learning domains. In the rest of the book, we cover more advanced reinforcement learning methods with the aim of teaching you

# 6

## *Alternative Optimization Methods: Evolutionary Strategies*

### **This chapter covers:**

- What are evolutionary strategies
- Pros and cons of evolutionary approaches vs previous algorithms
- Implementing the CartPole game without backpropagation
- Why Evolutionary Strategies scale better than other algorithms

### **6.1 A Different Approach to Reinforcement Learning**

Our next algorithm is a bit different from the previous approaches we took. For DQN and Policy Gradient, we created one agent whose policy depended on a neural network to approximate the Q-function or policy function. The agent would interact with the environment, collect experiences, and then use backpropagation to improve the accuracy of its neural network and hence, it's policy. We needed to carefully tune several hyperparameters ranging from selecting the right optimizer function, picking an initial learning rate along with a learning rate schedule such that the agent would learn within a reasonable amount of time, minibatch size (and replay buffer if needed) to name a few. Since training DQN and policy gradient algorithms both rely on stochastic gradient descent, which as the name suggests, relies on noisy gradients, there is no guarantee that these models will successfully learn (i.e. converge on a local or global optimum).

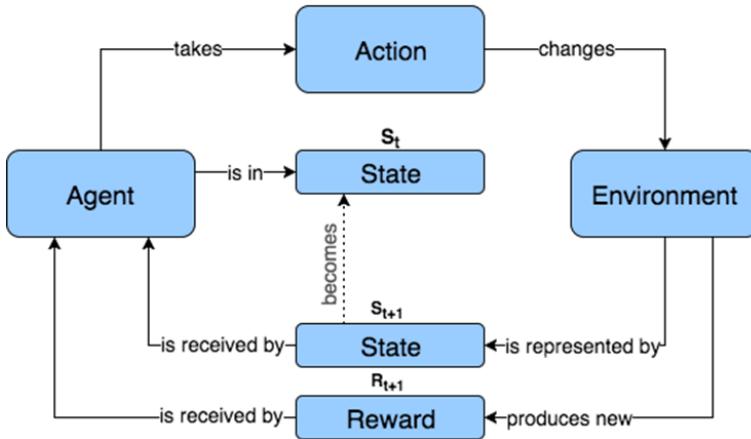


Figure 6.1: For the past algorithms that we covered, our agent interacted with environment, collected experiences, and then learned from those experiences. We repeated the same process over and over for each epoch until the agent stopped learning.

Depending on the environment and complexity of the network, creating an agent with the right hyperparameters may be incredibly difficult. Moreover, in order to use gradient descent and backpropagation we need a model that is differentiable. There are certainly models you could construct that might be interesting useful models but are impossible to train with gradient descent due to the lack of differentiability. But instead of creating one agent and improving it, what if we instead took a page from Charles Darwin and relied on evolution by (un)natural selection. We could spawn multiple different agents with different parameters (weights), observe which ones did the best, and breed the best agents such that the descendants could inherit their parents desirable traits – just like in natural selection. We could emulate biological emulation using algorithms. We would not need to struggle to tune hyperparameters and wait the multiple epochs to see if an agent is learning “correctly” – we would just pick the agents that are already performing better.

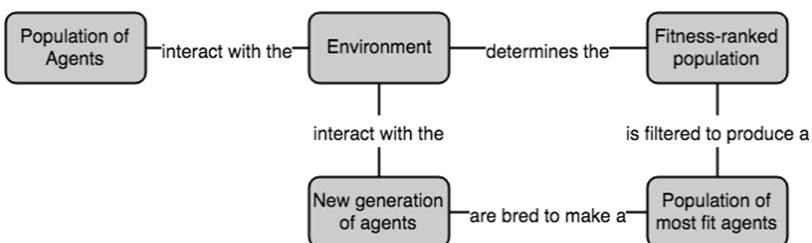


Figure 6.2: Evolutionary algorithms are different from gradient descent-based optimization techniques. With Evolutionary Strategies, we generate agents and pass the most favorable weights down to the subsequent agents.

This class of algorithms does not require an individual agent to learn - it does not rely on gradient descent - and is aptly called a **gradient-free algorithm**. But just because individual agents are not being nudged toward some objective directly, does not mean that we are relying on pure chance. The renowned evolutionary biologist Richard Dawkins once said, "Natural selection is anything but random." Similarly, in our quest to build, or more accurately discover, the best agent, we will not be relying on pure chance, but instead selecting for the fittest amongst a population with variance in traits.

## 6.2 Reinforcement Learning with Evolution Strategies

In this section, we'll talk about how fitness plays into evolution strategies, and we'll briefly cover the task of selecting for the fittest agents. Next, we'll work on how to recombine those agents into new agents and show what happens when we introduce mutations. This evolution is a multiple-generation process, so we'll discuss that and recap the full training loop.

### 6.2.1 Fitness

If you remember from your high school biology class, natural selection selects for the "most fit" individuals from each generation. In biology this represents the individuals that had the greatest reproductive success, and hence passed on their genetic information to subsequent generations. Birds with beak shapes more adept at procuring seeds from trees would have more food and thus be more likely to survive to pass that beak-shape gene to its children and grandchildren. But remember, "most fit" is relative to an environment. A polar bear is well adapted to the polar ice caps but would be very unfit in the Amazonian rainforests. You can think of the environment as determining an objective or fitness function that assigns individuals a fitness score based on their performance within that environment, and their performance is determined solely by their genetic information.

In evolutionary reinforcement learning we are selecting for traits that give our agents the highest reward in a given environment, and by traits, we mean model parameters (e.g. the weights of a neural network) or entire model structures. An RL agent's fitness can be determined by the expected reward it would receive if it were to perform in the environment. Let's say Agent A played the Atari game Breakout is able to achieve on average a score of 500 while Agent B is only able to obtain 300 points. We would say that Agent A is more fit than Agent B and that we want our optimal agent to be more similar to Agent A than B. And remember, the only reason why Agent A would be more fit than Agent B is because its model parameters are slightly more optimized to the environment.

The objective in evolutionary reinforcement learning is exactly the same as what we're used to. The only difference is we use this evolutionary process, which is often referred to as a genetic algorithm, to optimize the parameters of a model such as a neural network. The process is quite simple, but let's run through the steps of a genetic algorithm in more detail. We'll keep the description general but later we'll apply these specifically to a reinforcement learning problem.

Let's say we have some function  $f(x, \text{params})$ , which may be a neural network model or some other machine learning model, which takes some input  $x$  and a vector of parameters ( $\text{params}$ ) and we want to find the set of parameters where the output of the function  $f$  given  $x$  takes on some maximum or minimum value. If  $f$  were the cost function of a neural network, then we'd want to find the set of parameters that minimizes the cost given the input. With gradient descent, we'd compute the gradient of the function with respect the parameters and use that gradient information to take small, stochastic steps toward a minimum of the function (stochastic gradient descent). Let's see how this same problem is solved with a genetic algorithm.

- We generate an initial population of random potential solutions (i.e. a population of parameter vectors). We refer to each parameter vector in the population as an individual. Let's say this initial population has 100 individuals.
- Then we iterate through this population and assess the fitness of each individual by running the model with that parameter set and recording the model output. Each individual is assigned a fitness score based on its performance on the training data.
- We randomly sample a pair of individuals ("parents") from the population weighted according to their relative fitness score (individuals with higher fitness have a higher probability of being selected) to create a "breeding population".

**NOTE:** There are many different methods of selecting "parents" for the next generation. One way is to simply map a probability of selection onto each individual based on their relative fitness score, and then sample from this distribution. In this way, the most fit will be selected most often, but still leaving a small chance of poor performers to mate. This may help maintain population diversity. Another way is to simply rank/sort all the individuals and take the top N individuals and use those to mate to fill the next generation. Just about any method that preferentially selects the top performers to mate will work.

- The individuals in this breeding population will then "mate" to produce "offspring" that will form a new, full population of 100 individuals. For example, if the individuals are simply vectors of real numbers (e.g. parameter vectors), then mating vector 1 with vector 2 involves taking a subset from vector 1 and combining it with a complementary subset of vector 2 to make a new offspring vector of the same dimensions. For example, Vector 1: [1 2 3], Vector 2: [4 5 6]. Vector 1 mates with Vector 2 to produce [1 5 6] and [4 2 3]. We simply randomly pair up individuals from the breeding populations and recombine them to produce 2 new offspring until we fill up a new population.
- So now we have a new population with the top solutions from the last generation along with new offspring solutions, at this point, we will iterate over our solutions and randomly mutate some of them to make sure to introduce new "genetic diversity" into every generation to prevent premature convergence on a local optimum. Mutation simply means adding a little random noise to the parameter vectors. If these were binary vectors, mutation would be randomly flipping bits. Importantly, the mutation

- rate needs to be fairly low otherwise we risk ruining the already present good solutions.
- Repeat this process for  $N$  number of generations or until we reach convergence (i.e. the average population fitness has stopped improving significantly).

Before we dive into the reinforcement learning application, we'll run a super simple genetic algorithm on a toy problem for illustrative purposes. We will create a population of a random strings and try to evolve them toward a target string of our choosing, such as "Hello World!" So we'll start out with random strings like "gMIgSkybXZyP" and "adIBOM XIrBH," then we use a function that can tell us how similar these strings are to the target string "Hello World!" to give us the fitness scores. So strings that are more similar to "Hello World!" are more likely to be selected to be in the next generation.

We score all the initial random strings in the population and then we sample pairs of parents from the population weighted according to their relative fitness scores such that individuals with higher fitness scores are more likely to be chosen to become parents. We then "mate" these parents (also called crossing or recombining) to produce two offspring strings and add them to the next generation. We also mutate the offspring by randomly flipping a few characters in the string. We iterate this problem and eventually we expect that the population will become enriched with strings very close to our target, and probably at least one will hit our target exactly (at which point we stop the algorithm). Admittedly, we're unable to think of how this particular example might be useful in the real world, but it's one of the simplest examples of a genetic algorithm and the concepts will directly apply to our reinforcement learning tasks.

### **Listing 6.1 Evolving Strings**

```
import random
from matplotlib import pyplot as plt

alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ,.!" #A
target = "Hello World!" #B

class Individual: #C
    def __init__(self, string, fitness=0):
        self.string = string
        self.fitness = fitness

from difflib import SequenceMatcher

def similar(a, b): #D
    return SequenceMatcher(None, a, b).ratio()

def spawn_population(length=26, size=100): #E
    pop = []
    for i in range(size):
        string = ''.join(random.choices(alphabet, k=length))
        individual = Individual(string)
        pop.append(individual)
    return pop
```

```
#A This is the list of characters we sample from to produce random strings
#B This is the string we're trying to evolve from a random population
#C We set up a simple class to store information about each member of the population
#D This method will compute a similarity metric between two strings, giving us a fitness score
#E This method will produce an initial random population of strings
```

So far this code will create an initial population of individuals which are class objects composed of a string field and a fitness score field. We create the random strings by sampling from a list of alphabetic characters. After we have a population, we need to evaluate the fitness of each individual. As we'll see later, this is going to be a function of our loss function when we're using a machine learning model, but for strings, we can compute a similarity metric using a built-in Python function.

### **Listing 6.1 (continued)**

```
def recombine(p1_, p2_): #A
    p1 = p1_.string
    p2 = p2_.string
    child1 = []
    child2 = []
    cross_pt = random.randint(0, len(p1))
    child1.extend(p1[0:cross_pt])
    child1.extend(p2[cross_pt:])
    child2.extend(p2[0:cross_pt])
    child2.extend(p1[cross_pt:])
    c1 = Individual(''.join(child1))
    c2 = Individual(''.join(child2))
    return c1, c2

def mutate(x, mut_rate=0.01): #B
    new_x_ = []
    for char in x.string:
        if random.random() < mut_rate:
            new_x_.extend(random.choices(alphabet, k=1))
        else:
            new_x_.append(char)
    new_x = Individual(''.join(new_x_))
    return new_x
```

```
#A This function recombines two parent strings into two new offspring
#B This function will mutate a string by randomly flipping characters
```

The recombination function simply takes two parent strings like “hello there” and “fog world” and randomly recombines them by generating a random integer up to the length of the strings and taking the first piece of parent 1 and the second piece of parent 2 to create an offspring, such as “fog there” and “hello world” if the split happened in the middle. This means if we have evolved a string that contains part of what we want like “hello” and another string that contains another part of what we want like “world” then the recombination process might give us all of what we want.

The mutation process takes a string like “hellb” and with some small probability (the mutation rate) will replace a character in the string with a random one. For example, if the mutation rate was 20% (0.2) then it is probably that at least one of the 5 characters in “hellb” will be mutated to a random character, hopefully it will be mutated into “hello” if that was our target. The purpose of mutation is to introduce new information (variance) into the population. If all we did was recombine then it is likely all the individuals in the population would become too similar too fast and we wouldn’t find the solution we want because information gets lost each generation if there is no mutation. Note that the mutation rate is critical. If it’s too high then the fittest individuals will lose their fitness by mutation and if it’s too low we won’t have enough variance to find the optimal individual. Unfortunately, you just have to find the right mutation rate empirically.

#### **Listing 6.1 (continued)**

```
def evaluate_population(pop, target): #A
    avg_fit = 0
    for i in range(len(pop)):
        fit = similar(pop[i].string, target)
        pop[i].fitness = fit
        avg_fit += fit
    avg_fit /= len(pop)
    return pop, avg_fit

def next_generation(pop, size=100, length=26, mut_rate=0.01): #B
    new_pop = []
    while len(new_pop) < size:
        parents = random.choices(pop, k=2, weights=[x.fitness for x in pop])
        offspring_ = recombine(parents[0], parents[1])
        child1 = mutate(offspring_[0], mut_rate=mut_rate)
        child2 = mutate(offspring_[1], mut_rate=mut_rate),
        offspring = [child1, child2]
        new_pop.extend(offspring)
    return new_pop

#A This function assigns a fitness score to each individual in the population
#B This function generates a new generation by recombination and mutation
```

These are the last two functions we need to complete the evolutionary process. We have a function that evaluates each individual in the population and assigns a fitness score, which is just how similar the individual’s string is to the target string. The fitness score will vary depending on what the objective is for a given problem. Lastly, we have a function to generate a new population by sampling the most fit individuals in the current population, recombining them to produce offspring and mutating them.

#### **Listing 6.1 (continued)**

```
num_generations = 100
population_size = 3000
str_len = len(target)
mutation_rate = 0.001 #A
```

```

pop_fit = []
pop = spawn_population(size=population_size, length=str_len) #initial population
for gen in range(num_generations):
    pop, avg_fit = evaluate_population(pop, target)
    pop_fit.append(avg_fit) #record population average fitness
    new_pop = next_generation(pop, \
        size=population_size, length=str_len, mut_rate=mutation_rate)
    pop = new_pop

```

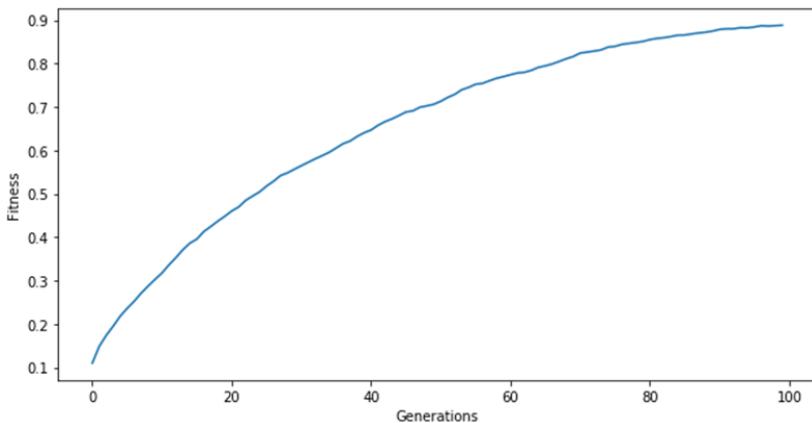


Figure 6.3: This is a plot of the average population fitness over the generations. The average population fitness increases fairly monotonically and then plateaus, which looks promising. If the plot was very jagged, then the mutation rate might be too high or the population size too low. If the plot converged too quickly, then the mutation rate might be too low.

If you run the algorithm, it should take a few minutes on a modern CPU and you can find the highest rank individual in the population using:

```

>>> pop.sort(key=lambda x: x.fitness, reverse=True) #sort in place, highest fitness first
>>> pop[0].string
"Hello World!"

```

It worked! This is actually a more difficult problem to optimize using an evolutionary strategy because the space of strings is not continuous, so it is hard to take small, incremental steps in the right direction since the smallest step is flipping a character. Hence, if you try making a longer target string it will take much more time and resources to evolve. When we're optimizing real-valued parameters in a model, even a small increase in value might improve the fitness and we can exploit that.

Let's see how this evolution strategy works in a simple reinforcement learning example. We're going to use an evolutionary process to optimize an agent to play Cartpole. We can represent an agent as a neural network that approximates the policy function – it accepts a

state and outputs an action. Below we have an example of a three-layer network initiated with random weights.

### **Listing 6.2 Defining an Agent**

```
import numpy as np
import torch

agent_weights = [
    torch.rand(self.state_space, 10), # fc1 weights
    torch.rand(10), # fc1 bias
    torch.rand(10, 10), # fc2 weights
    torch.rand(10), # fc2 bias
    torch.rand(10, self.action_space), # fc3 weights
    torch.rand(self.action_space), # fc3 bias
]

def get_action_from_agent_weights(agent_weight, state):
    x = F.relu(torch.add(torch.mm(state, weights[0]), weights[1]))
    x = F.relu(torch.add(torch.mm(x, weights[2]), weights[3]))
    act_prob = F.softmax(torch.add(torch.mm(x, weights[4]), weights[5]))
    action = np.random.choice(range(len(act_prob)), p=act_prob)
    return action
```

Note that we are not using PyTorch's nn.Sequential module like we did in the previous section. Although PyTorch's module provides useful abstractions, it does not easily allow us to set the default weights of our model, which we will need to do later in the chapter. All we need to do is initiate some weights, run our input through the model, and then select an action from the output.

To determine the fitness of the agent, we just need to run its policy through the environment and calculate the total reward that it would accrue. We should run it for multiple trials and average the runs to reduce the variance and get more accurate measurements. Just keep in mind that the more trials that we run, the longer it will take to generate determine the fitness.

### **Listing 6.3 Determining Fitness**

```
def get_fitness(self, env, agent_weights, max_episode_length, trial=5):
    total_reward = 0
    for _ in range(trials):
        observation = env.reset()
        for i in range(max_eps_length):
            action = agent(weights, observation)
            observation, reward, done, info = env.step(action)
            total_reward += reward
            if done: break
    return total_reward / trials
```

## 6.2.2 Selecting for the Fittest Agents

Okay, now we know how to determine when one agent is more fit than another. We will first create a set of agents (we'll call them the first generation) and evaluate the fitness of each of the agents.

### **Listing 6.4 Creating our first generation of agents**

```
first_generation = [init_random_agent_weights() for _ in range(generation_size)]
```

Once we have our first generation we can then select for the top two agents with the highest fitness but having each agent run through the environment. We can call these our parents.

### **Listing 6.5 Selecting the fittest agents**

```
top_agents_weights = sorted(agents_weights, reverse=True, key=lambda a:  
    get_fitness(env, a))[:2]
```

## 6.2.3 Recombining Agents to Produce New Agents

Now that we have determined the most agents, we need to recombine them to pass down their favorable weights to the next generation. We touched on recombination previously, but let's go over it again. Note that recombination is also called "crossing" or "mating" and we will use these terms interchangeably. In biological terms, each agent being recombined, or parent, has a set of genes made of DNA that they will pass down to their offspring. Each parent cannot pass down their entire genes or else their offspring will have twice as many genes as the parents did. Humans for example have 23 pairs of chromosomes (bundles of DNA) for a total of 46. If each parent passed all 46 of their chromosomes to their children then they would have  $46+46=92$  chromosomes. And if they have children their children will have more than that. This obviously is not the case (when too many chromosomes are passed on, it is generally not compatible with life). What instead happens is that each parent provides half of their genes, one chromosome per set. The child then will have half of its genes from one parent and the other.

Our artificial agents will follow a similar pattern. The genes in this case will be the weights of our network and each layer can be considered a chromosome. And we do not need to pass exactly half of each parents' genes to their children – we could pass down 1/3 from Parent A and 2/3 from Parent B, 4/5 from Parent A and 1/5 from Parent B, and so on. The problem is that we do not know which genes from each parent to pass down to their children, and so therefore we have to essentially randomly select them. Each parent has a chance of giving a set of weights to its offspring.



**Figure 6.4:** We will be passing the weights of each parent of each parent down to each child. Each child will receive a random amount of weights from each parent. In the above case, the child received the first two “genes” (FC1 and B1) from parent 1 and the remaining four from parent two, but this could be random permutation.

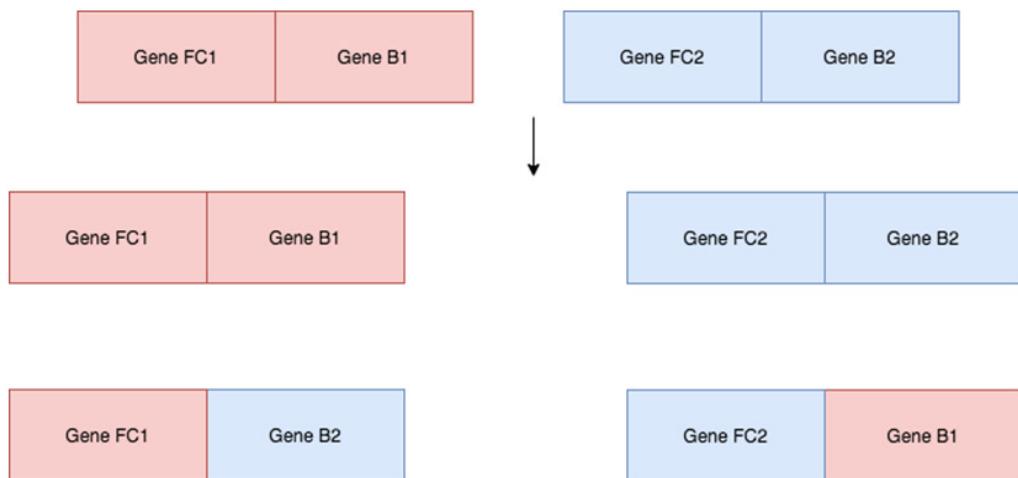
Below is the code implementation of mating (crossing) two agents. We randomly generate a random number between 0 and the total number of weights called a `crossover_idx` and the child will inherit all genes before the `crossover_idx` from Parent A and all genes after the `crossover_idx` from Parent B.

#### **Listing 6.6 Crossing**

```
def cross(agent1_weights, agent2_weights, agent_config):
    num_params = len(agent1_weights)
    crossover_idx = np.random.randint(0, num_params)
    new_weights = agent1.weights[:crossover_idx] + agent2.weights[crossover_idx:]
    num_params_to_update = np.random.randint(0, num_params) # num of params to
    change
    return new_weights
```

#### **6.2.4 Introducing Mutations**

Mutations are critical in evolution. In biology, mutations are random changes in an organism’s DNA. This sometimes leads to new traits that have not been exhibited before in any previous generations. Without mutation we would just be shuffling around the same genes of our ancestors. For example, let’s say there are two parents where parent 1 held genes FC1 and B1 and parent 2 held genes FC2 and B2. When they bred, and mutations are not possible, not matter how many children they have the children could have only four distinct sets of genes. Mutations add variance to a population, allowing novel information to be created.



**Figure 6.5: If two parents had only two genes, there would be only four different types of descendants that could form.**

There would be no point in breeding for more generations as well, as the same genes present in all the children were present in the parent. We could increase the number of genes to say 6 like we showed in Figure 6.4 and there would be 64 ( $2^6$ ) different children that could be produced. However, unless we start with a huge population size, it will be very difficult to produce enough variance in each generation such that some are significantly more adept than others.

Mutations allow us to generate a lot more variety in our populations. So when two parents cross to produce a child, they will not always produce an agent that can be predictably determined after the mutation process.

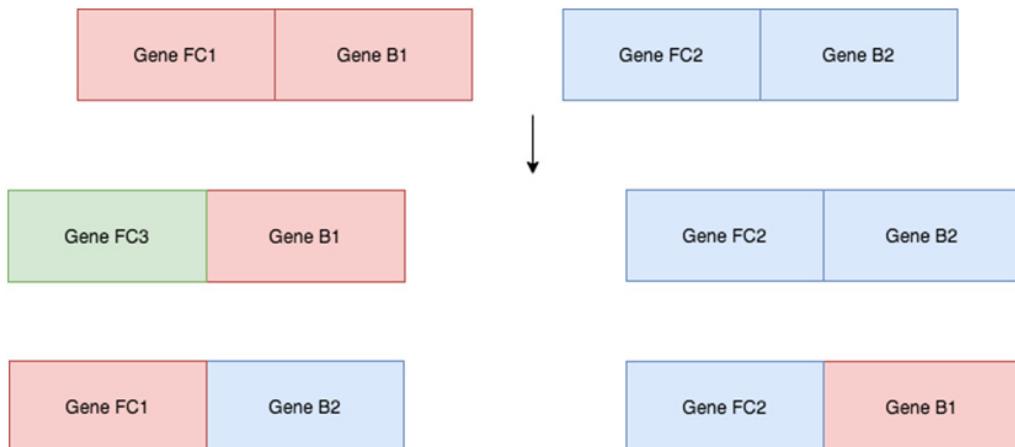


Figure 6.6: Mutations introduce randomly into each generation. Without mutations the combinations of children can easily be determined.

As shown in Figure 6.6, when mutations are introduced, there is a chance that a new variation of a gene may be formed, one that we may not have anticipated. This can then be passed on to future generations. And remember, each future generation has a chance to introduce even more mutations! Importantly, however, mutations happen at a low rate and only introduce small changes. A single mutation is not going to take a random set of parameters and transform them into perfectly optimized ones. But a few small mutations may add up and push the parameters slightly closer to the optimum.

Let's see how this would look in code. The way will do this is by taking an agent's weights, and randomly generate the number of weights we want to modify. For each weight that we want to modify, we will add some random noise to the weights.

### **Listing 6.7 Mutation**

```
def mutate(new_weights):
    num_params_to_update = np.random.randint(0, num_params) # num of params to
    change
    for i in range(num_params_to_update):
        n = np.random.randint(0, num_params)
        new_weights[n] = new_weights[n] + torch.rand(new_weights[n].size())
    return new_weights
```

### **6.2.5 Evolution takes multiple generations**

In biology, each mutation very subtlety changes the characteristics of the organism such that it may be difficult to discern one generation from another. However, it is the process of crossing multiple generations, allowing these mutations and variations to accumulate that allow for perceptible changes. Let's use the bird beak example we talked about briefly before.

At first all birds would have roughly the same beak shape. But as generations occurred, random mutations were introduced into the population. Again, most of these mutations probably did not impact the birds at all or even had a deleterious effect. But with a large enough population and enough generations, random mutations would occur that would affect beak shape favorably. Those birds would have an advantage getting food over the other birds and therefore have a higher likelihood of passing down their genes. Therefore, the next generation would have an increased frequency of the favorably shaped beak gene.

Enough about birds, now back to our reinforcement learning agents. Mutations increase the variance in performance of a population of agents. Because the mutations are random and the environment requires methodical navigation, most of these changes will not impact the fitness of the agent much or even cause the agent to perform worse than its parents.

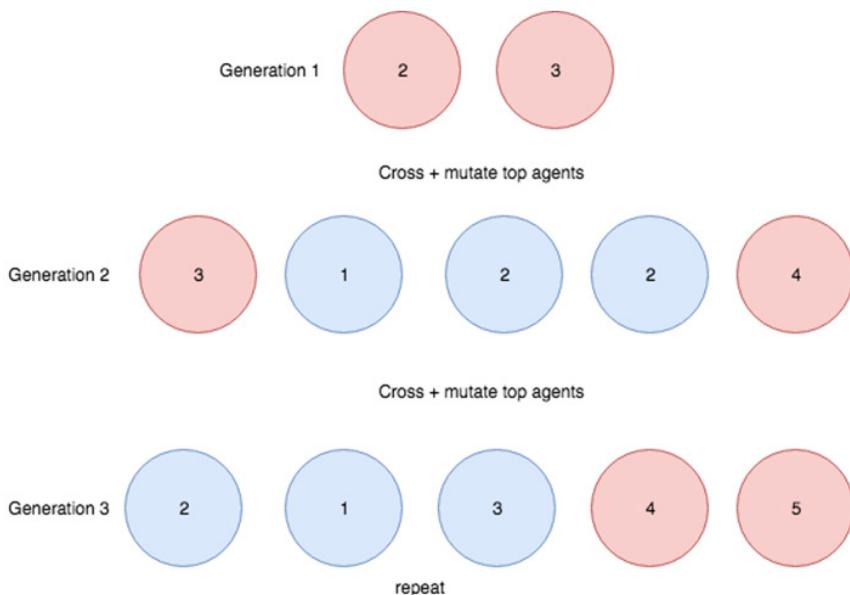


Figure 6.7: A diagram showing the principles of evolution by natural selection. The number represents the fitness of the agent and red agents are the parents that are selected to produce the subsequent generation. Each generation produces descendants with recombined characteristics derived from their parents. Since the crossing and mutations are random, not all descendants are more fit than their parents. It is the process of more fit individuals being more likely to cross that increases the overall population fitness from generation to generation.

However, with large enough population size and generation, there are bound to be descendants that perform slightly better than their parents.

### 6.2.6 Full training loop

Let's review all the steps. We first need to initiate our first generation of agents. We then select the top two agents, cross them while introducing mutations to produce the next generation. From that next generation we select the top two parents and repeat this until the top agents are produced.

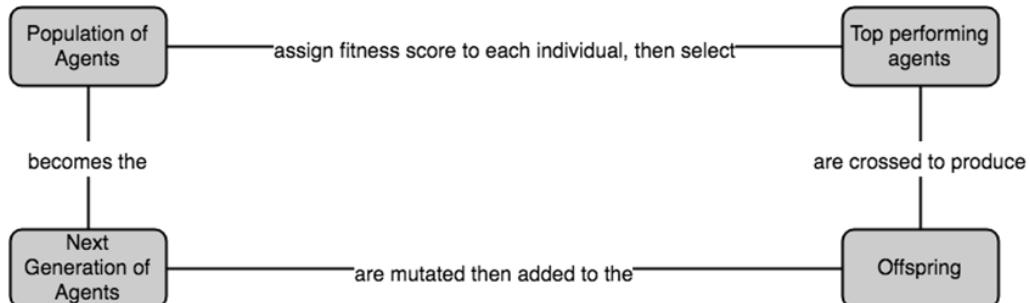


Figure 6.8: Full training loop for evolutionary optimization of reinforcement learning agents.

Below is the entire training loop. We will repeat this loop until an agent of desired fitness is obtained.

#### **Listing 6.8 Full Training Loop**

```

n_generations = 100
generation_size = 20
generation_fitness = []

env = gym.make('CartPole-v1')
max_fitness = 0

agents = [init_random_agent_weights(), init_random_agent_weights()]
for i in range(n_generations):
    next_generation = reproduce(env, agents, generation_size)
    ranked_generation = sorted([get_fitness(env, a) for a in next_generation],
                               reverse=True)
    avg_fitness = (ranked_generation[0] + ranked_generation[1]) / 2
    generation_fitness.append(avg_fitness)
    agents = next_generation
  
```

Let's give the above code a run and see how it turns out.

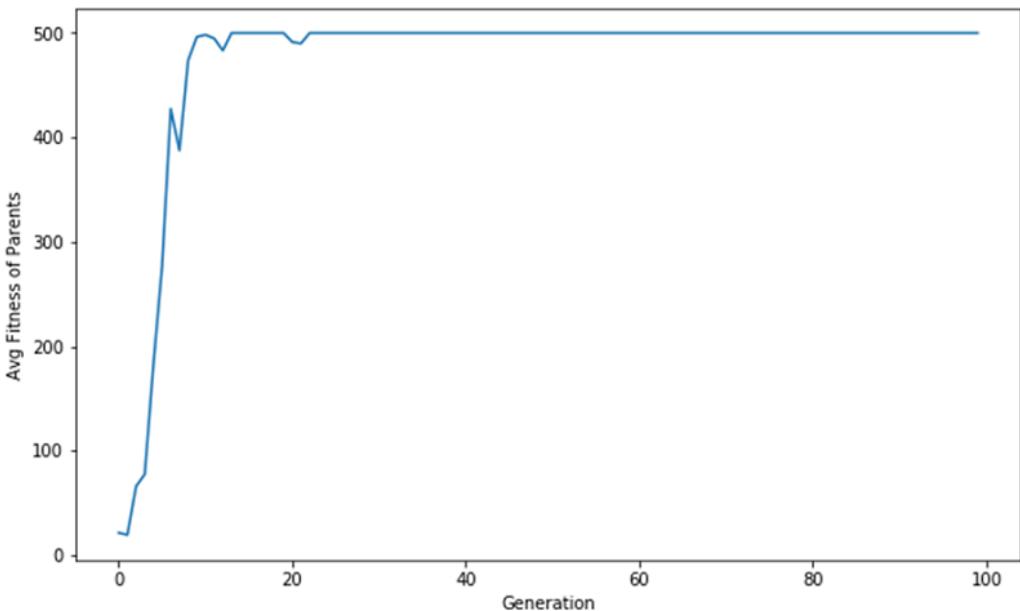


Figure 6.9. Figure caption.

Sweet, we just created an algorithm that mimics natural selection. As you can see from above, after creating a few generations of agents, we produced one that is able to maximize its reward in the environment. One of the drawbacks of evolutionary strategies is that we have to run our agent through the environment a lot more times than if we just used a gradient-based optimization algorithm. We will discuss a few pros and cons of this approach in the next section.

## 6.3 Pros and Cons of Evolutionary Algorithms

As you probably have seen, the algorithm we implemented above is a bit different the previous approaches. There are circumstances where an ES approach works better and others that make it impractical.

### 6.3.1 Evolutionary Algorithms Explore More

One advantage of gradient-free approaches is that they tend explore more than their gradient-based counter parts. Both DQN and PG followed a similar strategy – collect experiences and nudge our agent to take actions that led to more reward. As we discussed before, this tends to cause our agents to abandon exploring new states if it knows it prefers an action already. We addressed this with DQN by incorporating an epsilon-greedy strategy, meaning there's a small chance we will take a random action even if we have a preferred action. With stochastic policy

gradient we relied on drawing a variety of actions from the action probability vector output by our model.

The agents in the genetic algorithm on the other hand are not nudged toward any direction. We are producing a lot of agents in each generation, and with so much random variation between them, most of them will have different policies between each other.

### 6.3.2 Evolutionary algorithms are incredibly sample intensive

As you could probably see from the above code, we need each agent to run through the environment 5 times to determine its fitness. We also have 10 agents per generation. That means that we need to run through the environment 50 times to produce an incrementally improved agent. Compared with gradient-based methods, evolutionary algorithms tend to be more sample hungry since we aren't strategically adjusting the weights of our agents, we are just creating lots of agents and hoping that the random mutations / crossing we introduce are beneficial. We will say that evolutionary algorithms are less **data-efficient** than DQN or PG methods.

We could lower the number of episodes we need to determine fitness from 5 to a smaller number. However, there will be a lot more variance in the fitness value that is calculated from a lower number of runs. Remember, we want our agents to generalize well to being in different states in its environment and having it run through the environment multiple times is a more accurate way of doing so. For example, if we were only requiring the agent to run through the environment one time, we are depending on that single trial to provide us its expected fitness. It could be a bad agent that generalizes poorly but was initiated in a starting position that it was very familiar with leading to a high fitness. We would then falsely believe that the agent was more fit in general, select it as a parent, and pass its poor and overfitted weights to the next generation. This may mean that we need more generations in order to produce a better agent, requiring us to run through the environment more time in the end.

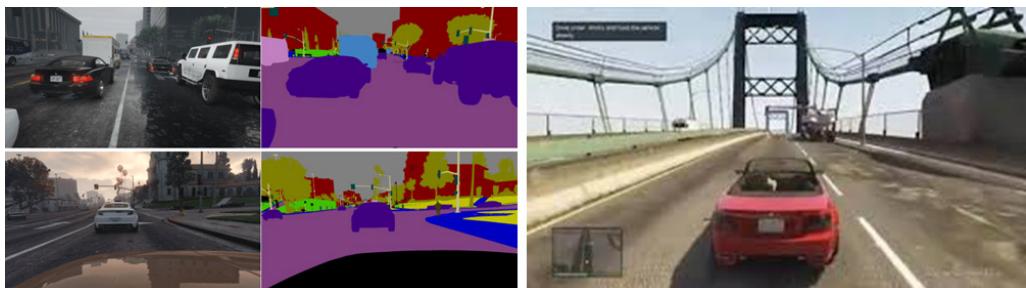
Let's say that want to decrease the size of the population so our parents will need to produce less agents. If you decrease the population size, there are less agents to select from when we are picking our two parents. This will make it likely that less fit individuals will make it into the next generation. We rely on a large number of agents being produced in hopes of finding a combination that leads to a better fitness. Additionally, as in biology, mutations most of the time have a negative impact and will lead to worse fitness. Having a larger population size increases the probability that a mutation we introduce will have a beneficial effect.

Being data-inefficient is a problem if collecting data is expensive, such as in robotics or with autonomous vehicles. Having a robot collect one episode of data usually takes a couple of minutes and we know from our past algorithms that training a simple agent takes hundreds if not thousands of episodes. Imagine how many episodes an autonomous vehicle would need to sufficiently explore its state space (the world). In addition to taking considerably more time, training with physical agents is much more expensive since you need to purchase the robot

and account for any maintenance. Ideally, we could train such agents without having to give them physical bodies.

### 6.3.3 Simulators

Simulators address the concerns listed above. Instead of using an expensive robot or building a car with the necessary sensors, we could instead use a computer software that would emulate the experiences the environment would provide. For example, when training agents to drive autonomous cars, instead of equipping cars with the necessary sensors and deploying their model on physical cars, we can just train the agents inside software environments such as the driving game Grand Theft Auto (or ideally a more realistic driving simulator). The agent would receive as input the images of its surroundings and would be trained to output driving actions that would get their vehicle to the programmed destination as safely as possible.



**Figure 6.10:** The video game Grand Theft Auto was initially used by autonomous driving engineers to test their models. The models would receive images of its surroundings and would output driving actions.

In addition to being significantly cheaper to train agents with, agents are able to train much more quickly since they can interact with environment much faster than in real life. If you need to watch and understand a 2-hour movie, it will require...2 hours of your time. If you focus more intensely, you could probably increase the playback speed by 2 or 3, dropping the amount of time needed to 1 hour or a bit less. A computer on the other hand can be finished before you finish the first act. Depending on the size of your computer, an 8 GPU computer (p3.16x large on Amazon AWS) running ResNet-50 can process over 700 images per second. In a 2-hour movie running at 24 frames per second (standard in Hollywood), there are 172,800 frames that need to be processed. This will require 4 minutes to finish. We could also effectively increase the playback speed for our deep learning model as well by dropping every few frames, which will drop our processing time to under 2 minutes. We could also throw more computers at the problem to increase processing power. For a more recent reinforcement learning example, the OpenAI Five bots were able to play 180 years of Dota 2 games each day. You get the picture...computers can process faster than we can, and that's why simulators are valuable.

### 6.3.4 Gradient-Free Algorithms could faster to train

So back to Evolutionary Strategies. If a simulator is present, the time and financial cost of collecting samples with ES is less of an issue. In fact, producing a viable agent with ES can sometimes be faster than gradient-based approaches because we do not have to compute the gradients via backpropagation. Depending on the complexity of the network, this will cut down the computation time by roughly 2-3 times. But there is another advantage of ES that can allow them to train faster than their gradient counter-parts – they can scale incredibly well. We will discuss this in detail in the next section.

## 6.4 Evolutionary Strategies are Parallelizable

**Here we'll dive into Evolutionary Strategies as a scalable alternative, discussing parallel v serial processing, scaling efficiency, communication between nodes, and a variety of scaling approaches.**

### 6.4.1 ES as a scalable alternative

Another reason that training an ES agent could be faster is that the algorithm can be scaled very efficiently. OpenAI in 2017 released a paper called *Evolutionary Strategies as a Scalable Alternative to Reinforcement Learning*, where they could train agents incredibly quickly efficiently by adding more machines. On a single machine with 18 CPU cores, they were able to make a 3D Humanoid learn to walk in 11 hours. But with 80 machines (1440 CPU cores) they were able to produce an agent in under 10 minutes!

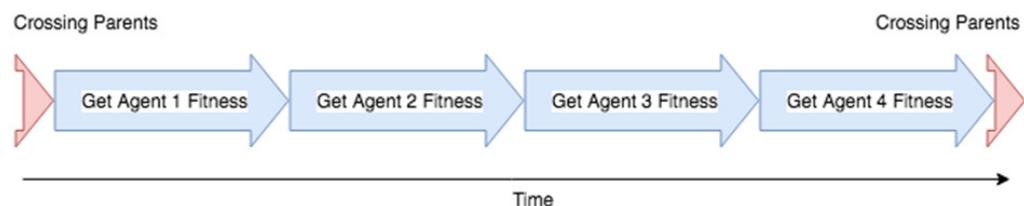


Figure 6.11: The 3D Humanoid environment provided by the gym library. This is high dimensional problem where the objective is for the humanoid to walk forward. With just one machine an ES was able to “solve” this problem in 11 hours, but by using 80 computers they were able to produce an agent in 10 minutes.

You may be thinking...well that's obvious, they just threw more machines and money at the problem. But this is actually trickier than it sounds, and other gradient-based approaches struggle to scale to that many machines. Before we dive the differences for how these methods scale, let's take a moment to demonstrate in detail what it means for a machine learning algorithm to scale.

#### 6.4.2 Parallel vs Serial Processing

For our above example, when determining the fittest agent in each generation, we had to wait for each agent to run through the environment completely before starting the next run. If the agent takes 30 seconds to run through the environment, and we have 10 agents we are determining the fitness for, this will take 5 minutes.



**Figure 6.12: Our current computational complexity.** Determining the fitness of our agent is often the slowest step in our training loop and requires that we run our agent through the environment (possibly many times). If we are doing this on single computer, we will be doing this in serial – that is we have to wait for one to finish running through the environment before we can start determining the agent of the second agent.

This is known as running a program in **serial**. Determining each agent's fitness will generally be the longest running task in an ES strategy, but each agent can evaluate its own fitness independent of each other. We could instead run each agent in the generation on multiple computers at the same time. Each of the 10 agents would go on 10 machines and we can determine their fitness simultaneously. This means that completing one generation will take ~30 seconds as opposed to 5 minutes when running on one machine, a 10x speed up. This will be known as running our process in **parallel**.

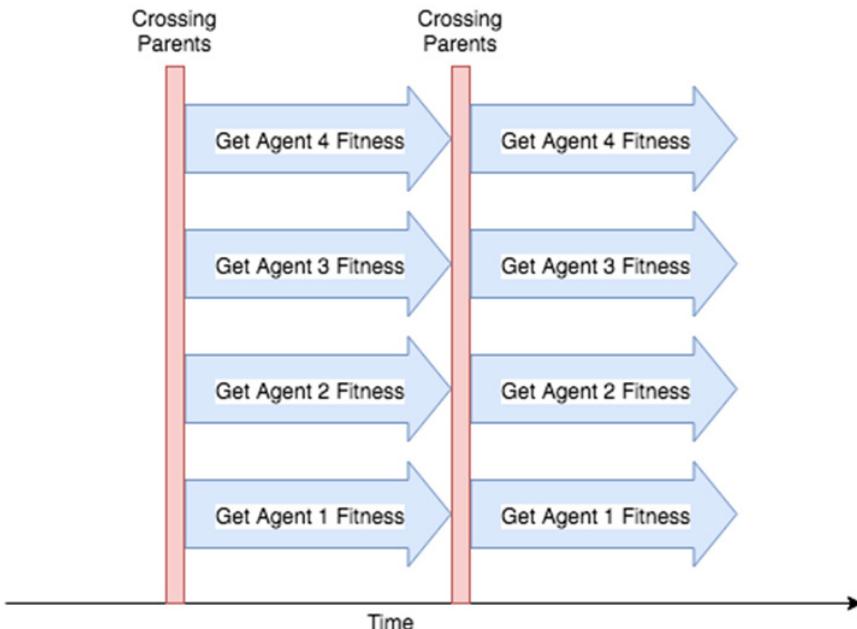


Figure 6.13: If we have multiple machines at our disposal, we can determine the fitness of each agent on its own machine in parallel with each other. We would not have to wait for one agent to finish running through the environment before starting the next one. This will provide us huge speed up if we are training agent with a long episode length.

### 6.4.3 Scaling Efficiency

So we can just throw more machines and money at the problem and we don't have to wait nearly as long. In the hypothetical example above where we added 10 machines and got a 10x speedup up. This is known as having a scaling efficient of 1.0. **Scaling efficiency** is a term used to describe how a particular approach improves as more resources are thrown at it and can be calculated as below.

$$\text{Scaling Efficiency} = \frac{\text{Multiple of Performance Speed up after adding resources}}{\text{Multiple of Resources Added}}$$

However, in the real world, processes never have a scaling efficiency of 1. There is always some additional cost to adding more machines that decreases its efficiency. More realistically adding 10 more machines will only give us a 9x speed up. Using the scaling efficiency equation above we can calculate the scaling efficiency as 0.9 (which is actually pretty good in the real world). Now back to our evolutionary example. We could implement it such that one machine is responsible for selecting the best agents and the other machines determine the fitness as shown in Figure 6.13.

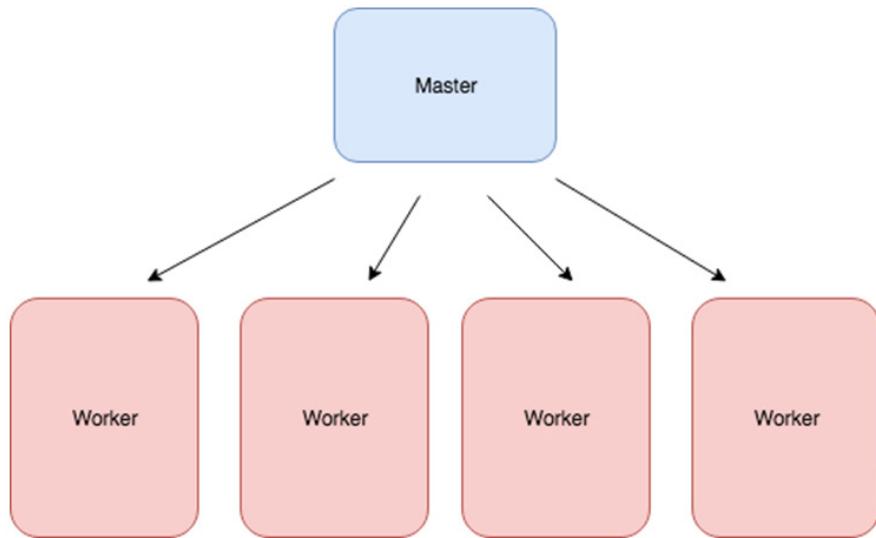
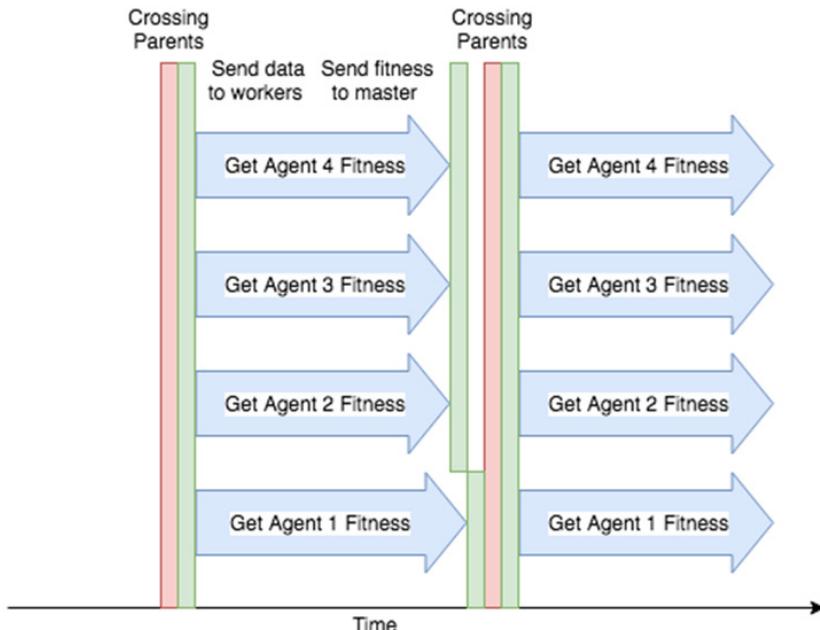


Figure 6.14: The topology for how we can distribute our ES algorithm onto many machines. We have one machine that does the selection process (we will call it the “master” node) and n-machines that determine the fitness of our agent.

Every step does take a little bit of network time to communicate between machines, which is something that we would not encounter if we were running everything on a single machine. Additionally, in order for the “crossing parents” step to begin in the master node, we need all our workers to complete calculating the fitness for each of their agents. If there is just one machine that is slower than the others, the other workers will need to wait.



**Figure 6.15:** A revised view of Figure 6.12 that accounts that introduces real world inefficiencies. The green bars represent network time it takes in order for the master and workers machines to communicate with one another. Additionally, if any of the workers are slower than the others, such as Agent 1, the other workers have to wait for it.

#### 6.4.4 Communicating Between Nodes

So far we have not talked about what data we are sending from the master node to the worker node, but you can imagine that the more data that you are sending between machines the longer it takes. The researchers at OpenAI developed this nifty strategy where each node sends only one number to each other node. We have modified their variation, and have simplified it such that the master to sends two numbers to each worker and each worker only needs to send one number back to the master.

Let's first take a naïve approach. So the master node is in charge of aggregating the fitness scores of the population and determining which agents are going to be the two new parents of the next generation. The workers on the other hand are responsible for determining the fitness of one agent and then sending that fitness to the master. We could have the master node produce the next generation and then send each worker the one agent's weights. However, this is incredibly expensive. Without our very small network with 3 layers of 10 hidden nodes each, there are over 1000 parameters each represented as a float32 (4 bytes of data). That means we would need to send 4 kB per worker. We can definitely do better than that.

### 6.4.5 Sending only two numbers

Okay, so only the master node knows which agents are the parents and that information is necessary in order to produce the next generation. What if each worker had all the same agents on them and the master node was only responsible for communicating which agents were the parents.

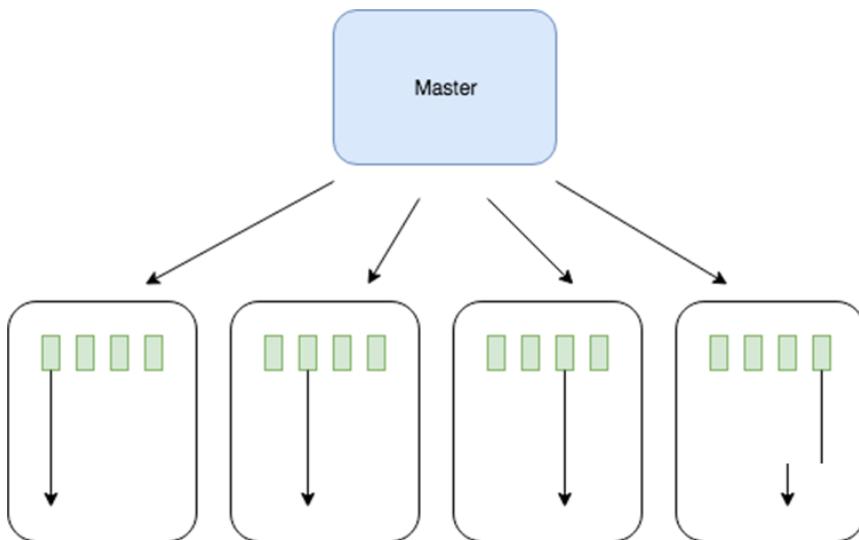


Figure 6.16: The architecture derived from OpenAI's distributed ES paper. Each worker contains all the weights of all the agents but is responsible for evaluating the fitness of only a subset of the population. The fitness scores of the agents can then be sent to the master and the master can send the positions of the most fit agents (the parents) back to the workers. The workers can then produce the next generation and repeat this process.

We would only need to send two numbers to each worker, the identification of parent 1 and parent 2. The workers, now knowing the identity of the parents, would be responsible of producing the next generation, which is a relatively computationally cheap task. There's a catch though. Each worker would need to produce the exact same generation as each other. Even though they have both parents' weights, as we mentioned throughout this chapter, there is a lot of randomness involved. For each descendant in the next generation we randomly generate a `crossover_idx`, the number of genes we want to mutate, and then add random noise to mutate the genes. Luckily, there is solution to this already.

### 6.4.6 Seeding

Seeding allows us to consistently generate the same random numbers every time, even on different machines. If you run the following code you will get the same output as we've shown below above even though these numbers should be generated "randomly."

#### **Listing 6.9 Seeding**

```
import numpy as np
np.random.seed(10)
np.random.rand(4)
>>> array([0.77132064, 0.02075195, 0.63364823, 0.74880388])

np.random.seed(10)
np.random.rand(4)
>> array([0.77132064, 0.02075195, 0.63364823, 0.74880388])
```

Seeding is important so that experiments involving random numbers could be replicated again for other researchers to reproduce. If you do not supply an explicit seed, then usually the system time or some other sort of variable number is used. If we came up with a novel new RL algorithm, we would want others to be able to verify our work on their own machines. We would want the agent that another lab generated to be identical to eliminate any source of error (and therefore doubt) and that's why it's important we provide as much detail about our algorithm as possible – the architecture, the hyperparameters used, and sometimes the random seed we used. However, we hope we've developed an algorithm that is robust to the particular set of random numbers generated.

Now back to scaling our ES problem. If each worker is initially seeded with the same number, the crossing and mutating steps will be exactly the same, and each generation produced on each worker will be identical to each other, which was exactly what we were looking to accomplish. All the workers need to do next is to figure out which subset of agents they will be evaluating the fitness for and we can send those numbers back to the master server and repeat this process over again. We can just have each worker communicate to each other which agents in the generation they are going to evaluate.

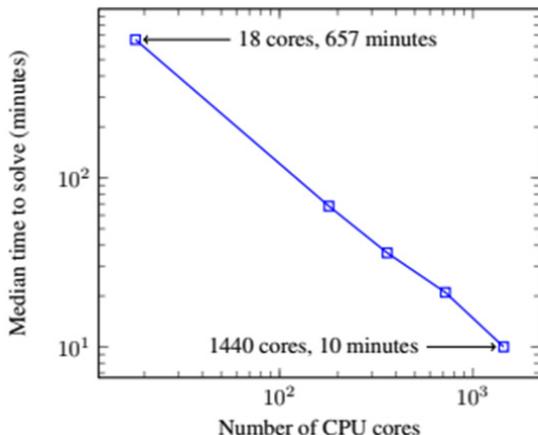
But does that even need to be communicated amongst the workers? We can structure it such that each node always evaluates the fitness of the same subsets of agents. In Figure 6.15 we show that each worker node houses a generation of agents of size 4. The first node could always be responsible for evaluating the fitness of the agent at index 0, the second node responsible for the agent at index 1 and so on.

### 6.4.7 Scaling Linearly

This approach we showed above is a simplified version of the real architecture that the researchers at OpenAI did.

As you may have noticed, the master node does not seem like it has that large of a responsibility and the OpenAI engineers actually removed it and just had the workers

communicate directly with each other. The workers would calculate the fitness of its specified agents, then send over the fitness scalar to every other worker. At the end of the episode, each worker would have all the finesse of the generation, can determine which should be bred, and then cross its subset and repeat this process. By reducing the volume of data sent between the nodes, adding an additional node did not affect the network significantly, and they were able to scale to over a thousand workers linearly.



**Figure 6.17.** Figure taken from the *OpenAI Evolutionary Strategies as a Scalable Alternative to Reinforcement Learning* paper. The figure demonstrates that as more compute resources were added, the time improvement remained constant.

**Scaling linearly** means that for every machine added, we receive roughly the same performance boost as we did with adding the previous machine. This is denoted by a straight line on a performance over resources graph as seen in Figure 6.16.

#### 6.4.8 Scaling Gradient Based Approaches

Gradient-based approaches can be trained on multiple machines as well. However, they do not scale nearly as well as ES. Currently, most approaches for distributed training of gradient-based approaches involve training the agent on each worker and then passing the gradients back to a central machine to be aggregated. All the gradients must be passed for each epoch or update cycle which requires a lot of network bandwidth and strain on the single central machine. Eventually the network is saturated and adding more workers does not improve training speed as well.

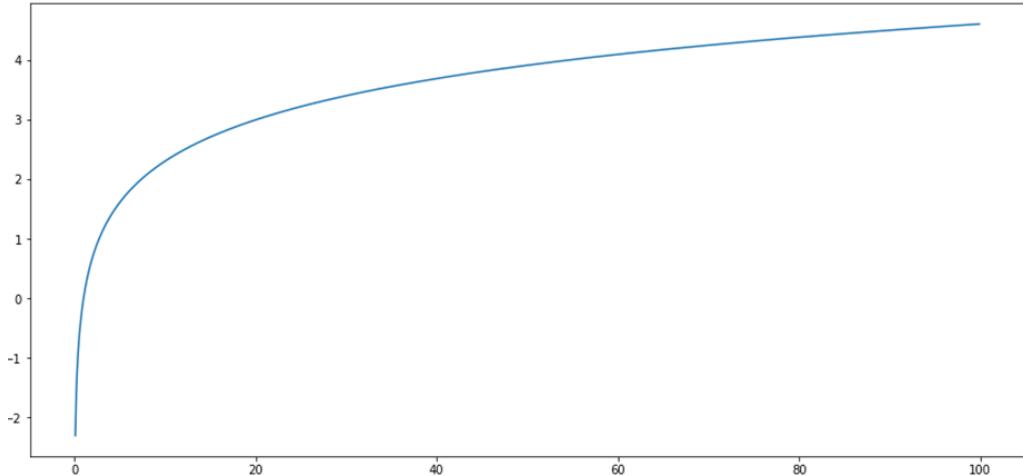


Figure 6.18. The performance of current gradient-based approaches resembles graph like this. In the beginning there is seemingly linear trend because the network has not been saturated. But eventually as more resources are added, we get less and less of a performance boost.

Evolutionary approaches on the other hand, since it does not require backpropagation, does not need to send gradient updates to a central server. And with smart techniques like the ones that OpenAI developed, it may only need to send a single number.

## 6.5 Summary

- Evolutionary Strategies provide us another powerful toolkit in our arsenal. Adapting from how we humans have evolved:
  - 1.1 we produce individuals
  - 1.2 select the best from the current generation
  - 1.3 shuffle their genes around
  - 1.4 mutate them to introduce some variation
  - 1.5 and mate them to create new generations for the next population
- We were able to code an evolutionary algorithm to fit our reinforcement learning task. We calculated how to determine the fitness of an agent, how to shuffle parameters around to produce new agents, and how to introduce mutations to produce new variation.
- ES algorithms tend to be more data hungry and less data-efficient than gradient-based approaches and that in some circumstances this may be fine, notably if you have a simulator. For some problems though, we may not have access to a simulator and we need to be frugal with our data, such as when we are trying to train a robot.

- In the next section, we will discuss a class of algorithms notable for being more data-efficient than both ES and gradient-base algorithms.

# 8

## *Curiosity-Driven Exploration*

### In this chapter

- Understand the sparse reward problem
- Understand how curiosity can serve as an intrinsic reward
- Play Super Mario Bros. from OpenAI Gym
- Implement an intrinsic curiosity module in PyTorch
- Train a deep Q-network agent to play Super Mario Bros. using just curiosity

Code for this chapter is at this book's GitHub repository under Chapter 8:

<https://github.com/DeepReinforcementLearning/DeepReinforcementLearningInAction/>

---

The fundamental reinforcement learning algorithms we have studied so far such as Deep Q-learning and policy gradient methods are very powerful techniques in a lot of situations, but fail dramatically in other environments. Google DeepMind pioneered the field of deep reinforcement learning back in 2013 when they used Deep Q-learning to train an agent to play multiple Atari games at superhuman performance levels. But the performance of the agent was highly variable across different types of games. At one extreme their DQN agent played the Atari game Breakout vastly better than a human, but at the other extreme, the DQN was much worse than a human at playing Montezuma's Revenge where it could not even pass the first level.

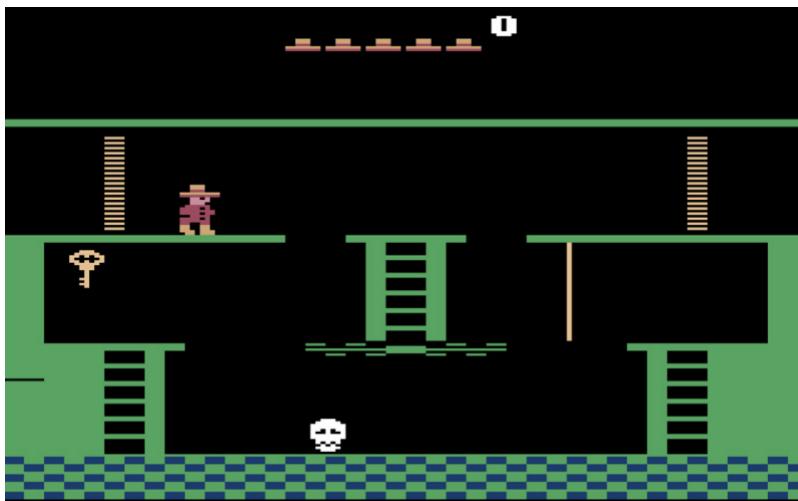
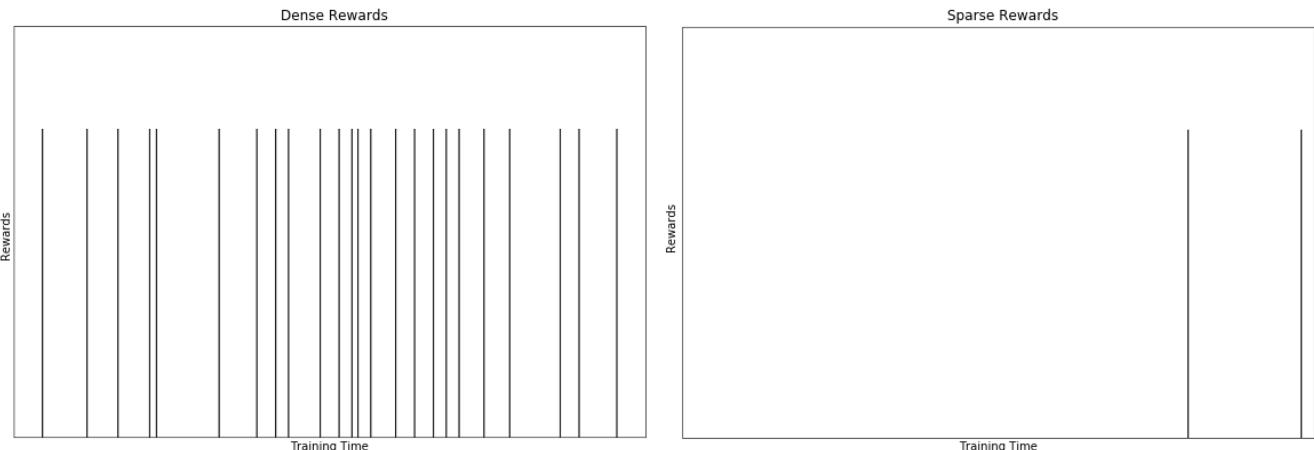


Figure 8.1: Screenshot from the Montezuma's Revenge Atari game. The player must navigate through obstacles to get a key before any rewards are received.

### **WANT TO LEARN MORE ?**

The paper that brought great attention to the field of deep reinforcement learning was “Human-level control through deep reinforcement learning” by Mnih and collaborators at Google DeepMind in 2015. The paper is fairly readable and contains the details you need to replicate their results.

What’s the difference between the environments that explains these disparities in performance? The games that DQN was successful at all gave relatively frequent rewards during game play and did not require significant long-term planning. Montezuma’s Revenge on the other hand only gives a reward after finding a key in the room, which also contains numerous obstacles and enemies. With a vanilla DQN the agent starts exploring essentially at random. It will take random actions and wait to observe rewards, and then the rewards reinforce which actions are best to take given the environment. But in the case of Montezuma’s Revenge, it is extremely unlikely that the agent will find the key and get a reward with this random exploration policy, so it will never observe a reward and will never learn.



**Figure 8.2:** In environments with dense rewards, rewards are received fairly frequently during training time making it easy to reinforce actions. In sparse reward environments, rewards may only be received after many sub-goals are completed, making it difficult or impossible for an agent to learn based on reward signals alone.

This problem is called the sparse reward problem since the rewards in the environment are sparsely distributed. If the agent doesn't observe enough reward signals to reinforce its actions, it can't learn. Animal and human learning offer us the only natural examples of intelligent systems and we can turn to them for inspiration. Indeed, researchers trying to tackle this sparse reward problem noticed that humans not only maximize extrinsic (i.e. from the environment) rewards like food and sex, but also demonstrate an intrinsic curiosity, a motivation to explore just for the sake of understanding how things work.

In this chapter we will learn about methods for successfully training reinforcement learning agents in sparse reward environments by using principles from human intelligence, specifically our innate curiosity. We will see how curiosity can drive the development of basic skills that the agent can use to accomplish sub-goals and find the sparse rewards.

## 8.1 Tackling Sparse Rewards with Predictive Coding

In the world of neuroscience and particularly computational neuroscience, there is a framework for understanding neural systems at a high level called the predictive coding model. In this predictive coding model, the theory says essentially that all neural systems from individual neurons up to large scale neural networks are running an algorithm that attempts to predict inputs, and hence tries to minimize the prediction error. So at a high level, as you're going about your day your brain is taking in a bunch of sensory information from the environment and it's training to predict how the sensory information will evolve, it's trying to stay one step ahead of the actual raw data coming in.

If something surprising (i.e. unexpected) happens, your brain experiences a large prediction error and then presumably does some parameter updating to prevent that from happening again. For example, you might be talking to someone you just met and your brain is constantly trying to predict the next word that person will say before they say it. Since this is someone you don't know, your brain will probably have a relatively high average prediction error but if you become best friends, you'll probably be quite good at finishing their sentences. Similarly, curiosity can be thought of as a kind of desire to reduce the uncertainty in your environment (and hence reduce prediction errors). If you were a software engineer and saw some online posts about this interesting field called machine learning, your curiosity to read a book like this would be to reduce your uncertainty about machine learning.

Using a prediction error mechanism was one of the first attempts to imbue reinforcement learning agents with a sense of curiosity. The idea was that in addition to trying to maximize extrinsic (i.e. environment-provided) rewards, the agent would also just try to predict the next state of the environment given its action and would try to reduce its prediction error. In very familiar areas of an environment, the agent would learn how it works and would have a low prediction error. By using this prediction error as another kind of reward signal, the agent would be incentivized to visit areas of the environment that were novel and unknown. That is, the higher the prediction error is, the more surprising a state is and therefore the agent should be incentivized to visit these high prediction error states. Figure 8.1 shows the basic framework for this approach.

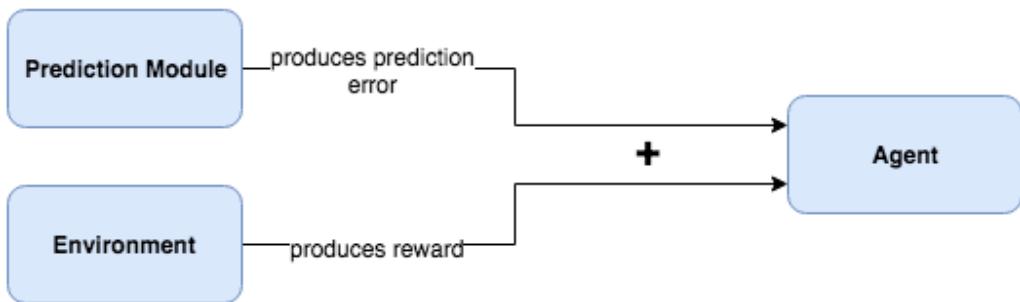
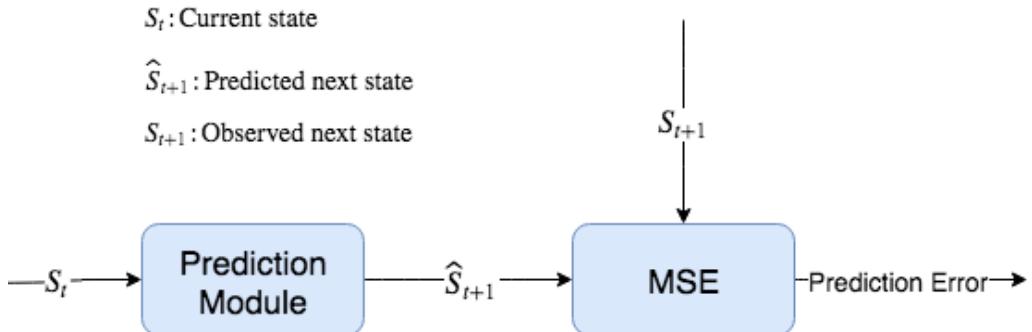


Figure 8.3: Prediction error is summed with the extrinsic environment reward for use by the agent.

The idea is to just sum the prediction error (which we will call the intrinsic reward) with the extrinsic reward and use that as the new reward signal for the environment. Now the agent is incentivized to not only figure out how to maximize the environment reward but also to be curious about the environment. The prediction error is calculated according to Figure 8.2.



**Figure 8.4:** The prediction module takes in a state  $S_t$  and action  $a_t$  and produces a prediction for the subsequent state  $S_{t+1}$ . This prediction along with the true next state are passed to a mean-squared error (or some other error) function which produces the prediction error.

So this intrinsic reward is based on the prediction error of states in the environment. It turns out this works fairly well at first pass, however, people eventually realized it runs into another problem often called the “noisy TV problem.” It turns out, if you train these agents in an environment that has a constant source of randomness, such as a TV screen playing random noise, the agent will have a constantly high prediction error and will be unable to reduce it, so it just stares at the noisy TV indefinitely. This is more than just an academic problem since many real-world environments have these kinds of sources of randomness (e.g. a tree’s leaves blowing in the wind).

At this point, it seems like prediction error has a lot of potential, but the noisy TV problem is a big flaw. Perhaps we shouldn’t pay attention to the absolute prediction error but instead to the rate of change of prediction error. When the agent transitions to an unpredictable state, it will experience a transient surge of prediction error, but then it goes away. Similarly, if the agent encounters a noisy TV, at first it is highly unpredictable and therefore high prediction error, but the high prediction error is maintained and thus the rate of change is zero. This formulation is better, but still has some potential issues. Imagine an agent is outside and sees a tree with its leaves blowing in the wind. The leaves are blowing around randomly and hence this is a high prediction error. The wind stops blowing, and now the prediction error goes down since the leaves are not moving anymore. Then the wind starts blowing again and prediction error goes up. In this case, even if we’re using prediction error rate, the rate will be fluctuating along with the wind. We need something more robust.



**Figure 8.5:** The noisy TV problem is a theoretical and practical problem where a reinforcement learning agent with a naïve sense of curiosity will become entranced by a noisy TV, forever staring at it. This is because it is intrinsically rewarded by unpredictability and white noise is very unpredictable.

We want to use this prediction error idea but don't want it to be vulnerable to randomness or unpredictability in the environment that doesn't matter. How do we add in the "doesn't matter" constraint to the prediction error module? Well when we say something doesn't matter, we mean that it does not affect us or is perhaps uncontrollable. If leaves are randomly blowing in the wind, the agent's actions don't affect the leaves and the leaves don't affect the actions of the agent. It turns out we can implement this idea as a separate module in addition to the state prediction module, and that is the subject of this chapter. This chapter is based on elucidating and implementing the idea from the paper "Curiosity-driven Exploration by Self-supervised Prediction" Pathak et al 2017 which successfully resolves the issues we've been discussing so far.

We will follow this paper pretty closely because it was one of the biggest contributions to solving the sparse reward problem that led to a flurry of related research. It also turns out to be one of the easiest algorithms to implement among the many others in this area. In addition, one of the goals of this book is to not only teach you the foundational knowledge and skills of reinforcement learning, but to give you a solid enough mathematics background to be able to read and understand reinforcement learning papers and implement them on your own. Of course, some papers require advanced mathematics that are outside the scope of this book, many of the biggest papers in the field require only some basic calculus, algebra, and linear algebra understanding, things that you probably know if you have made it this far. So the only barrier is really getting past the mathematical notation, something we hope to make easier here. We want to teach you how to fish rather than just giving you fish, as the saying goes.

## 8.2 Inverse Dynamics Prediction

Let's start to peer into the inside of the prediction module. The prediction error module from last section is implemented as a function  $f:(S_t, a_t) \rightarrow \hat{S}_{t+1}$  that takes a state and the action

taken and returns the predicted next state. It is predicting the dynamics of the environment, and so we call it the **forward prediction model**, since it predicts the future (forward) state.

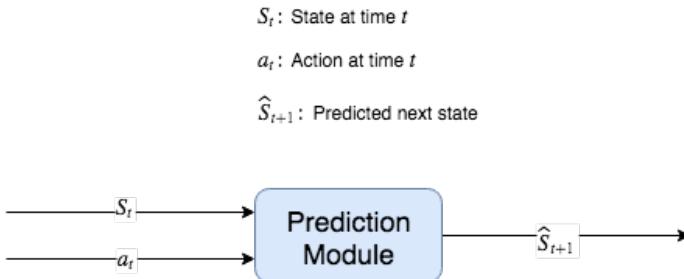


Figure 8.6: Diagram of the forward prediction module function  $f:(S_t, a_t) \rightarrow \hat{S}_{t+1}$ , that maps a current state and action to a predicted next state.

Remember we want to only predict aspects of the state that actually matter, not parts that are trivial or noise. The way we build in the “doesn’t matter” constraint to the prediction model is to add another model called an **inverse model**  $g:(S_t, S_{t+1}) \rightarrow \hat{a}_t$ . That is, it is a function  $g$  that takes a state and the next state and then returns a prediction for which action was taken that led to that transition  $S_t$  to  $S_{t+1}$ .

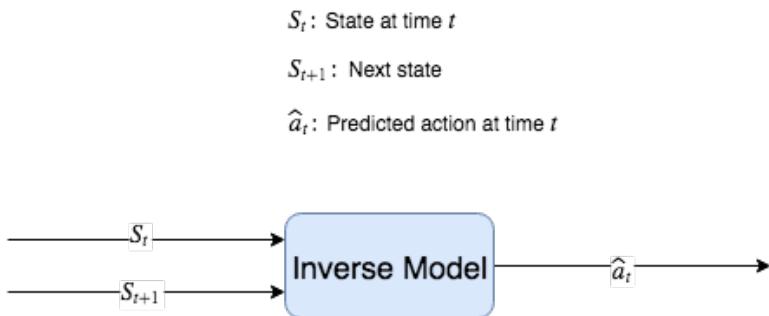


Figure 8.7: The inverse model takes two consecutive states and tries to predict which action was taken.

On its own, this inverse model is not really useful, there’s an additional model that is tightly coupled to the inverse model called the encoder model, denoted  $\phi$ . The encoder is a function,  $\phi:S_t \rightarrow \hat{S}_t$ , i.e. it takes a state and returns an encoded state  $\hat{S}_t$  such that the dimensionality of  $\hat{S}_t$  is significantly lower than the raw state  $S_t$ . The raw state might be an RGB video frame with height, width and channel dimensions and  $\phi$  will encode that state into a low dimensional vector. For example, a frame might be 100 pixels by 100 pixels by 3 color channels for a total

of 30,000 elements. Many of those pixels will be redundant and not useful, so we want our encoder to encode this state into say a 200-element vector with high level features.

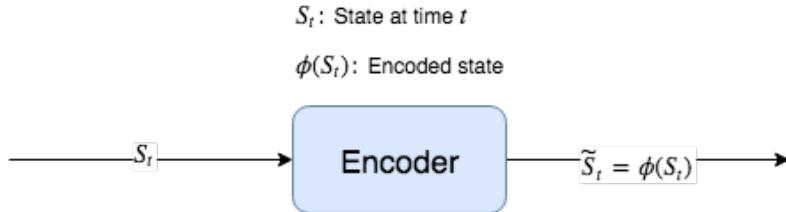


Figure 8.8: The encoder model takes a high-dimensional state representation such as an RGB array and encodes it as a low-dimensional vector.

## NOTATION

A variable with the tilde symbol over it such as  $\tilde{S}_t$  denotes some sort of transformed version of the underlying variable which may have different dimensionality. A variable with the hat symbol over it such as  $\hat{S}_t$  denotes an approximation (or prediction) of the underlying state and is the same dimensionality.

The encoder model is trained via the inverse model because we actually use the encoded states as inputs to the forward and inverse models  $f$  and  $g$  rather than the raw states. That is, the forward model becomes a function  $f: \phi(S_t) \times a_t \rightarrow \hat{\phi}(S_{t+1})$  where  $\hat{\phi}(S_{t+1})$  refers to a prediction of the encoded state, and the inverse model becomes a function  $g: \phi(S_t) \times \phi(S_{t+1}) \rightarrow (a_t)$ .

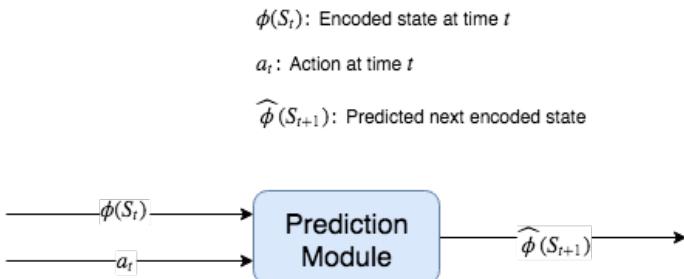
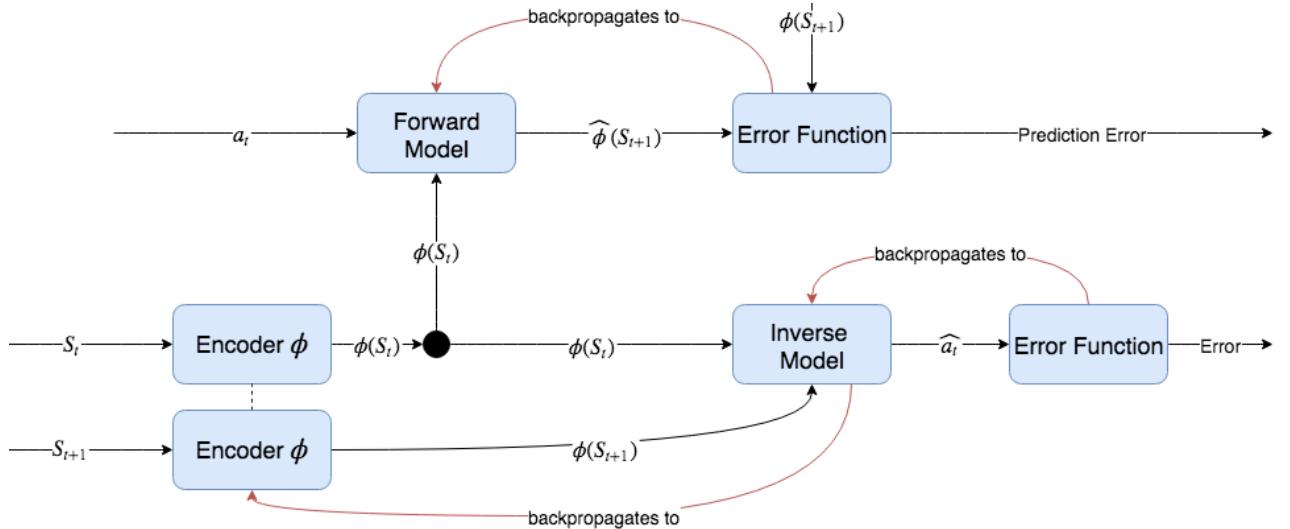


Figure 8.9: The forward prediction module actually uses encoded states, not the raw states. Encoded states are denoted  $\phi(St)$  or  $\tilde{S}_t$

The encoder model isn't trained directly, it is *not* an autoencoder, it is only trained through the inverse model. The inverse model is trying to predict the action that was taken to transition from one state to the next using the encoded states as inputs, and in order to minimize its own prediction error, its error will backpropagate through to the encoder model as well as itself. The encoder model will then learn to encode states in a way that is useful for the task of

the inverse model. Importantly, although the forward model also uses the encoded states as inputs, we do **not** backpropagate from the forward model to the encoder model. If we did, then the forward model will coerce the encoder model into mapping all states into a single fixed output since that will be the easiest to predict. Figure 8.3 shows the overall graph structure of these components.



**Figure 8.10: The curiosity module.** First the encoder will encode states  $S_t$  and  $S_{t+1}$  into low dimensional vectors,  $\phi(S_t)$  and  $\phi(S_{t+1})$ , respectively. These encoded states are passed to the forward and inverse models. Notice that the inverse model backpropagates to the encoded model thereby training it through its own error. The forward model is trained by backpropagating from its own error function but it does **not** backpropagate through to the encoded like the inverse model does. This ensures that the encoder learns to produce state representations that are only useful for predicting which action was taken. The black circle just indicates a copy operation that copies the output from the encoder and passes the copies to the forward and inverse models.

Figure 8.3 shows the forward pass of all these components and also the backward (backpropagation) pass to update the model parameters. It is worth repeating that the inverse model backpropagates back through to the encoder model and the encoder model is only trained together with the inverse model. We must use PyTorch’s `detach()` method to detach the forward model from the encoder so that it won’t backpropagate into the encoder. The purpose of the encoder is not to give us a low-dimensional input for improved performance, the purpose is to learn to encode the state using a representation that only contains information relevant for predicting actions. This means that aspects of the state that are randomly fluctuating and have no impact on the agent’s actions will be stripped from this encoded representation. This, in theory, should avoid the noisy TV problem.

Notice that for both the forward and inverse models we need access to the data for a full transition, i.e. we need  $(S_t, a_t, S_{t+1})$ , but this is not an issue when we use an experience replay

memory as we did in the Deep Q-learning chapter since the memory will store a bunch of these kinds of tuples.

### 8.3 Setting up Super Mario Bros.

Together, the forward, inverse and encoder models form the Internal Curiosity Module (ICM), which we will cover in detail later in this chapter. The components of the ICM function together for the sole purpose of generating an intrinsic reward that drives curiosity in the agent. The ICM only generates a new intrinsic reward signal based on information from the environment, therefore it is independent of how the agent model is implemented. We could use whatever agent model implementation we want, such as A3C (the actor-critic policy gradient method we covered in chapter 5) model, however, we will use a Q-learning model here to keep things simple and focus on implementing the ICM. The ICM can be used for any type of environment, but it will be most useful in sparse reward environments. We will use Super Mario Bros as our testbed.

Super Mario Bros. does not really suffer from the sparse reward problem. The particular environment implementation we will use provides a reward in part based on forward progress through the game, so positive rewards are almost continuously provided. However, Super Mario Bros. is still a great choice to test the ICM because we can choose to “turn off” the extrinsic (environment provided) reward signal and see how well the agent explores the environment just based off curiosity. The implementation of Super Mario Bros. has 12 discrete actions that can be taken at each time step including a NO-OP (no-operation, i.e. do nothing) action. The complete action list is below:

**Table 8.1: Actions in Super Mario Bros.**

Index	Action
0	Do nothing
1	Right
2	Right + A
3	Right + B
4	Right + A + B
5	A
6	Left
7	Left + A
8	Left + B
9	Left + A + B

10	Down
11	Up

A = Jump, B = run

You can install Super Mario Bros. with pip:

```
>>> pip install gym-super-mario-bros
```

After it is installed, you can test the environment (e.g. try running this code in a Jupyter Notebook) by playing a random agent (taking random actions). All of the code for this chapter is at this book's GitHub repository under Chapter 8:

<https://github.com/DeepReinforcementLearning/DeepReinforcementLearningInAction/>

To review how to use the OpenAI Gym, please refer back to chapter 4.

### **Listing 8.1: Setting up the Super Mario Bros. Environment**

```
import gym
from nes_py.wrappers import BinarySpaceToDiscreteSpaceEnv #A
import gym_super_mario_bros
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT, COMPLEX_MOVEMENT #B
env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = BinarySpaceToDiscreteSpaceEnv(env, COMPLEX_MOVEMENT) #C

done = True
for step in range(2500): #D
    if done:
        state = env.reset()
    state, reward, done, info = env.step(env.action_space.sample())
    env.render()
env.close()
```

#A This a wrapper module that will makes the action-space smaller by combining actions together.

#B There are two sets of action-spaces we can import, one with 5 actions (simple) and one with 12 (complex).

#C Wrap the environment's action space to be 12 discrete actions.

#D Test the environment by taking random actions

If everything went well, a little window should pop-up with Super Mario Bros. but it will be doing random actions and not making any forward progress through the level. By the end of this chapter you will have trained an agent that makes consistent forward progress, has learned to avoid or jump on enemies and jump over obstacles, and only using the intrinsic curiosity-based reward.

Recall that the environment is instantiated as a class object called `env` and that the main method you need to use is its `step(...)` method. The `step` method takes an integer representing the action to be taken. As with all OpenAI Gym environments, this one returns `state`, `reward`, `done`, and `info` data after each action is taken. The `state` is a numpy array with

dimensions (240, 256, 3) representing an RGB video frame. The reward is bounded between -15 and 15 and is based on the amount of forward progress. The `done` variable is a Boolean that indicates whether or not the game is over (e.g. Mario dies). The `info` variable is a Python dictionary with the following metadata:

**Table 8.2: Description of the meta-data returned after each action in the `info` variable**

Key	Type	Description
<code>coins</code>	int	The number of collected coins
<code>flag_get</code>	bool	True if Mario reached a flag or ax
<code>life</code>	int	The number of lives left, i.e., {3, 2, 1}
<code>score</code>	int	The cumulative in-game score
<code>stage</code>	int	The current stage, i.e., {1, ..., 4}
<code>status</code>	str	Mario's status, i.e., {'small', 'tall', 'fireball'}
<code>time</code>	int	The time left on the clock
<code>world</code>	int	The current world, i.e., {1, ..., 8}
<code>x_pos</code>	int	Mario's x position in the stage

Source: <https://github.com/Kautenja/gym-super-mario-bros>

We will only need to use the `x_pos` key/value. In addition to getting the state after calling the `step` method, you can also retrieve the state at any point by calling `env.render("rgb_array")`. That's basically all you need to know about the environment in order to train an agent to play it.

## 8.4 Preprocessing and the Q-network

The raw state is an RGB video frame with dimensions (240, 256, 3), which is unnecessarily high-dimensional and would be computationally costly at no advantage. We will convert these RGB states into grayscale and resize to be 42 x 42.

**Listing 8.2: Downsample State and Convert to Grayscale**

```
import matplotlib.pyplot as plt
from skimage.transform import resize #A
import numpy as np

def downscale_obs(obs, new_size=(42,42), to_gray=True):
    if to_gray:
        return resize(obs, new_size, anti_aliasing=True).max(axis=2) #B
    else:
        return resize(obs, new_size, anti_aliasing=True)
```

#A The scikit-image library has an image resizing function built-in

#B To convert to grayscale we simply take the maximum values across the channel dimension for good contrast

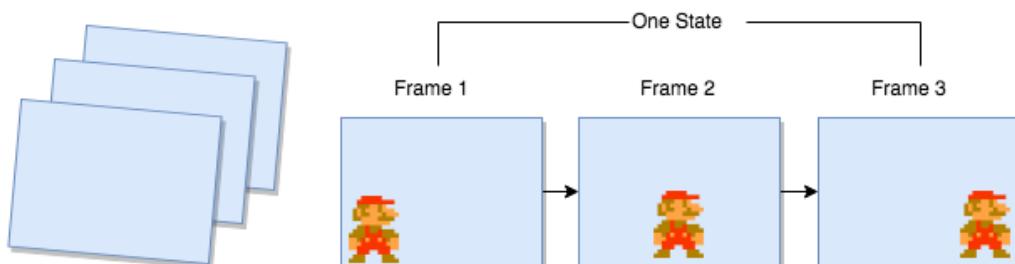
This function accepts the state array (obs), a tuple indicating the new size in height and width and a Boolean for whether to convert to grayscale or not. We set it True by default since that is what we want. We use the scikit-image library's resize function, so you may need to install it if you don't have it already by following the instructions at <https://scikit-image.org/download>. It's a very useful library for working with image data in the form of multi-dimensional arrays.

You can use matplotlib to visualize a frame of the state.

```
>>> plt.imshow(env.render("rgb_array"))
>>> plt.imshow(downsamp_obs(env.render("rgb_array")))
```

The down-sampled image will look pretty blurry, but it still contains enough visual information to play the game.

We have a few other data processing functions to build to transform these raw states into a useful form. We will not just pass a single 42x42 frame to our models, we will instead pass the last 3 frames of the game (in essence adding a channel dimension) so the states will be 3x42x42. Using the last 3 frames gives our models access to velocity information (i.e. how fast and which direction things are moving) rather than just positional information.



State Representation: 3 Grayscale Frames

**Figure 8.11:** Each state given to the agent is a concatenation of the 3 most recent (grayscale) frames in the game. This is necessary so that the model can have access to not just the position of objects but also their direction of movement.

When the game first starts, we only have access to the first frame, so we prepare the initial state by just concatenating the same state 3 times to get that  $3 \times 42 \times 42$  initial state. After this initial copied state, we can just replace the last frame in the state with the most recent frame from the environment, replace the second frame with the old last one, and replace the first frame with the old second. Basically we have a fixed length first-in-first-out data structure where we append to the right and the left automatically pops off. Python has a built in data

structure called the `deque` from the `collections` library that can implement this behavior when the `maxlen` attribute is set to 3.

These are the three functions we will use to prepare the raw states into a form that our agent and encoder models will use. The `prepare_state` function just resizes the image, converts to grayscale, converts from numpy to PyTorch Tensor and adds a batch dimension using the `.unsqueeze(dim=0)` method. The `prepare_multi_state` takes a tensor of dimensions `Batch x Channel x Height x Width` and updates the channel dimension with new frames. This function will only be used during testing of the trained model, during training we will use a `deque` data structure to continuously append and pop frames.

#### **Listing 8.4: Preparing the states**

```
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F

def prepare_state(state): #A
    return torch.from_numpy(downscaled_obs(state,
        to_gray=True)).float().unsqueeze(dim=0)

def prepare_multi_state(state1, state2): #B
    state1 = state1.clone()
    tmp = torch.from_numpy(downscaled_obs(state2, to_gray=True)).float()
    state1[0][0] = state1[0][1]
    state1[0][1] = state1[0][2]
    state1[0][2] = tmp
    return state1

def prepare_initial_state(state, N=3): #C
    state_ = torch.from_numpy(downscaled_obs(state, to_gray=True)).float()
    tmp = state_.repeat((N, 1, 1))
    return tmp.unsqueeze(dim=0)
```

#A Downscale state and convert to grayscale, then convert to a PyTorch Tensor, then finally add a batch dimension

#B Given an existing 3-frame state1 and a new single frame 2, add the latest frame to the queue

#C Create a state with 3 copies of the same frame and add a batch dimension

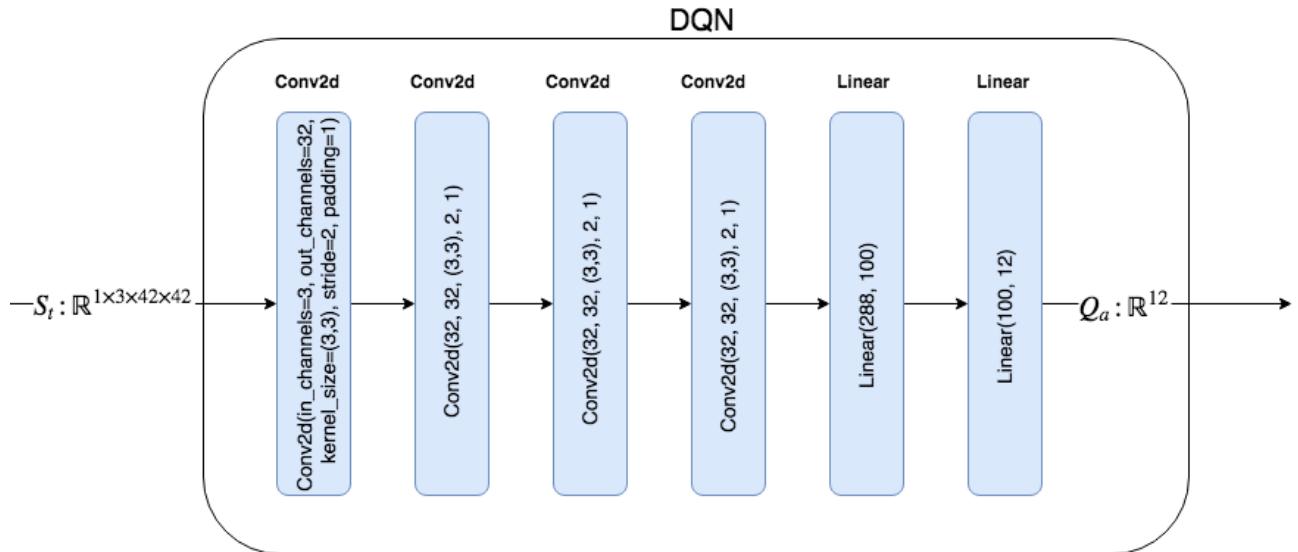
## **8.5 Setting up the Q-network and Policy Function**

As we mentioned, we will use a Deep Q-network (DQN) for the agent. Recall that a DQN takes a state and produces action-values, i.e. predictions for the expected rewards for taking each possible action. We use these action-values to determine a policy for action selection. For this particular game there are 12 discrete actions, therefore the output layer of our DQN will produce a vector of length 12, where the first element is the predicted value of taking action 0, and so on.

Remember that action-values are (in general) unbounded in either direction, they can be positive or negative if the rewards can be positive or negative (which they can be in this

game) so we do not apply any activation function on the last layer. The input to the DQN is a tensor of shape `Batch x 3 x 42 x 42`, where, remember the channel dimension (3) is for the most recent 3 frames of game play.

For the DQN, we use an architecture consisting of 4 convolutional layers and 2 linear layers. The exponential linear unit (ELU) activation function is used after each convolutional layer and the first linear layer (but no activation function after the last linear layer). The architecture is diagrammed in Figure 8.4. As an exercise you can add an LSTM or GRU layer that can allow the agent to learn from long-term temporal patterns.



**Figure 8.12:** This is the diagram of the DQN architecture we will use. The state tensor is the input and is passed through 4 convolutional layers then two linear layers. The ELU activation function is applied after the first 5 layers but not the output layer because the output needs to be able to produce arbitrarily scaled Q-values.

So our DQN will learn to predict the expected rewards for each possible action given the state (i.e. action-values or Q-values), and we use these action-values to decide which action to take. Naively we should just take the action associated with the highest value, however, our DQN will not produce accurate action-values in the beginning, so we need to have a policy that allows for some exploration so the DQN can learn better action-value estimates.

Earlier we discussed using the epsilon-greedy policy where we take a random action with probability  $\epsilon$  and take the action with the highest value with probability  $1-\epsilon$ . We usually set  $\epsilon$  to be some reasonably small probability like 0.1, and often we'll slowly decrease  $\epsilon$  during training so that it becomes more and more likely to just choose the highest value action.

We also discussed sampling from a softmax function as our policy. The softmax function essentially takes a vector input with arbitrary real numbers and then outputs a same-sized vector where each element is a probability and all elements sum to 1. If the input vector is action-values, then the softmax will return a discrete probability distribution over the actions based on their action-values, such that the action with the highest action-value will be assigned the highest probability. If we sample from this distribution then the actions with the highest values will be chosen more often, however, other actions will also be chosen. The problem with this approach is that if the best action (according to action-value) is only slightly better than other options, then the worse actions will still be chosen with fairly high frequency. For example, below we take an action-value tensor for 5 actions and apply the softmax function from PyTorch's functional module.

```
>>> torch.nn.functional.softmax(th.Tensor([3.6, 4, 3, 2.9, 3.5]))
tensor([0.2251, 0.3358, 0.1235, 0.1118, 0.2037])
```

As you can see the best action (index 1) is only slightly better than the others, so all the actions have pretty high probability and this policy is not that much different from a uniformly random policy. We will use a policy that begins with a softmax policy to encourage exploration and after a fixed number of game steps we will switch to an epsilon-greedy strategy which will continue to give us some exploration capacity but mostly just takes the best action.

#### **Listing 8.5: The policy function**

```
def policy(qvalues, eps=None): #A
    if eps is not None:
        if torch.rand(1) < eps:
            return torch.randint(low=0,high=7,size=(1,))
        else:
            return torch.argmax(qvalues)
    else:
        return torch.multinomial(F.softmax(F.normalize(qvalues)), num_samples=1) #B
```

#A The policy function takes a vector of action-values and an epsilon (eps) parameter.

#B If eps is not provided, use a softmax policy. We sample from the softmax using the multinomial function.

The other big component we need for the DQN is an experience replay memory. Gradient-based optimization does not work well if you only pass one sample of data at a time because the gradients are too noisy. In order to average over the noisy gradients, we take sufficiently large samples (called batches or mini-batches) and average or sum the gradients over all the samples. Since we only see one sample of data at a time when playing a game, we instead store the experiences in a "memory" store and then sample minibatches from the memory for training. We will build an experience replay class that contains a list to store tuples of experiences, where each tuple is of the form  $(S_t, a_t, r_t, S_{t+1})$ . The class will also have methods to add a memory and sample a minibatch.

**Listing 8.6: Experience Replay**

```

from random import shuffle
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F

class ExperienceReplay:
    def __init__(self, N=500, batch_size=100):
        self.N = N #A
        self.batch_size = batch_size #B
        self.memory = []
        self.counter = 0

    def add_memory(self, state1, action, reward, state2):
        self.counter +=1
        if self.counter % 500 == 0: #C
            self.shuffle_memory()

        if len(self.memory) < self.N: #D
            self.memory.append( (state1, action, reward, state2) )
        else:
            rand_index = np.random.randint(0,self.N-1)
            self.memory[rand_index] = (state1, action, reward, state2)

    def shuffle_memory(self): #E
        shuffle(self.memory)

    def get_batch(self): #F
        if len(self.memory) < self.batch_size:
            batch_size = len(self.memory)
        else:
            batch_size = self.batch_size
        if len(self.memory) < 1:
            print("Error: No data in memory..")
            return None
        #G
        ind = np.random.choice(np.arange(len(self.memory)),batch_size,replace=False)
        batch = [self.memory[i] for i in ind] #batch is a list of tuples
        state1_batch = torch.stack([x[0].squeeze(dim=0) for x in batch],dim=0)
        action_batch = torch.Tensor([x[1] for x in batch]).long()
        reward_batch = torch.Tensor([x[2] for x in batch])
        state2_batch = torch.stack([x[3].squeeze(dim=0) for x in batch],dim=0)
        return state1_batch, action_batch, reward_batch, state2_batch

```

#A N is the maximum size of the memory list

#B batch\_size is the number of samples to generate from the memory with the get\_batch(...) method

#C Every 500 iterations of adding a memory, shuffle the memory list to promote a more random sample

#D If the memory is not full, add to the list, otherwise replace a random memory with the new one

#E Use Python's built-in shuffle function to shuffle the memory list

#F Randomly samples a minibatch from the memory list

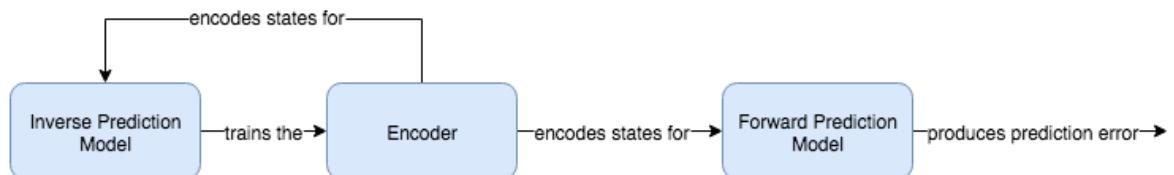
#G Create an array of random integers representing indices

The experience replay class essentially wraps a list with extra functionality. We want to be able to add tuples to the list but only up to a maximum number and we want to be able to sample from the list. When we sample with the `get_batch(...)` method we create an array of random integers representing indices in the memory list. We index into the memory list with these indices, retrieving a random sample of memories. Since each sample is a tuple  $(S_t, a_t, r_t, S_{t+1})$ , we want to separate out the different components and stack them together into a  $S_t$  tensor,  $a_t$  tensor and so on where the first dimension of the array is the batch size. For example, the  $S_t$  tensor we want to return should be of dimension `batch_size × 3 (channels) × 42 (height) × 42 (width)`. PyTorch's `stack(...)` function will concatenate a list of individual tensors into a single tensor. We also make use of the `squeeze(...)` and `unsqueeze(...)` methods to remove and add dimensions of size 1.

With all of that setup, we have just about everything we need to train a vanilla DQN besides the training loop itself. So in the next section we will implement the intrinsic curiosity module (ICM).

## 8.6 Intrinsic Curiosity Module

As we described earlier, the ICM is composed of 3 independent neural network models: the forward model, inverse model, and encoder. The forward model is trained to predict the next (encoded) state given the current (encoded) state and an action. The inverse model is trained to predict the action that was taken given two successive (encoded) states  $\phi(S_t)$  and  $\phi(S_{t+1})$ . The encoder simply transforms a raw 3 channel state into a single low-dimensional vector. The inverse model acts indirectly to train the encoder to encode states in a way that only preserves information relevant to predicting the action.



**Figure 8.13:** High-level overview of the intrinsic curiosity module (ICM). The ICM has 3 components that are each separate neural networks. The encoder model encodes states into a low-dimensional vector and it is trained indirectly through the inverse model that tries to predict the action that was taken given two consecutive states. The forward model predicts the next encoded state and its error is the prediction error that is used as the intrinsic reward.

The forward model is a simple two layer neural network with linear layers. The input to the forward model is constructed by concatenating the state  $\phi(S_t)$  with the action  $a_t$ . The encoded state  $\phi(S_t)$  is a tensor  $B \times 288$  and the action  $a_t : B \times 1$  is a batch of integers indicating the action index, so we make a one-hot encoded vector by just creating a vector of size 12 and setting the respective  $a_t$  index to 1. Then we concatenate these two tensors to create a Batch

$\times 288 + 12 = \text{Batch} \times 300$  dimensional tensor. We use the rectified linear unit (ReLU) activation unit after the first layer but we do not use an activation function after the output layer. The output layer produces a  $B \times 288$  tensor.

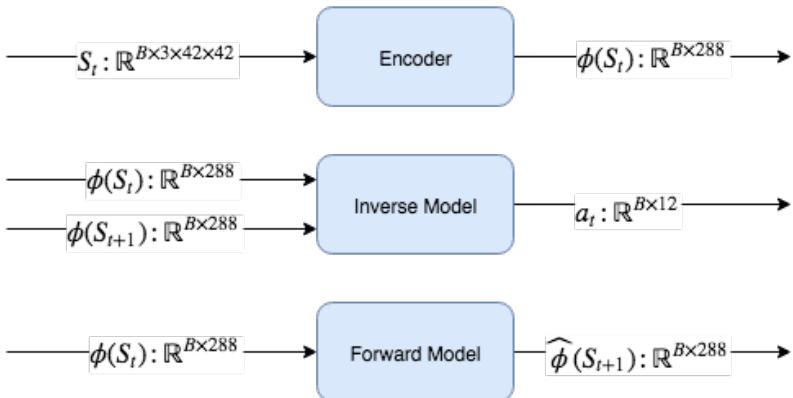


Figure 8.14: This figure shows the type and dimensionality of the inputs and outputs of each component of the ICM.

The inverse model is also a simple two layer neural network with linear layers. The input is two encoded states  $S_t$  and  $S_{t+1}$  concatenated together to form a tensor of dimension Batch  $\times 288 + 288 = \text{Batch} \times 576$ . We use a ReLU activation function after the first layer. The output layer produces a tensor of dimension Batch  $\times 12$  with a softmax function applied resulting in a discrete probability distribution over actions. When we train the inverse model, we compute the error between this discrete distribution over actions and a one-hot encoded vector of the true action taken.

The encoder is a neural network composed of 4 convolutional layers (with an identical architecture to the DQN) with the ELU activation function after each layer. The final output is then flattened to get a flat 288 dimensional vector output.

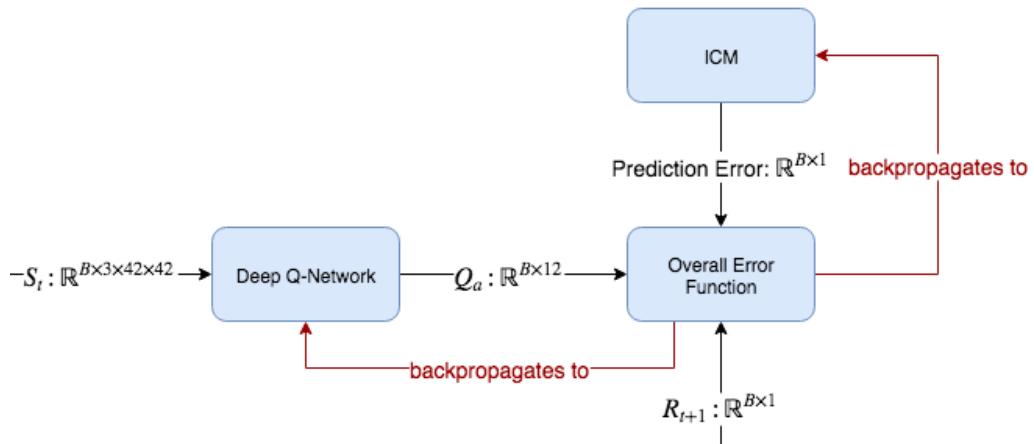


Figure 8.15: The DQN and the ICM contribute to a single overall loss function that is given to the optimizer to minimize with respect to the DQN and ICM parameters. The DQN's q-value predictions are compared to the observed rewards. The observed rewards, however, are summed together with the ICM's prediction error to get a new reward value.

The whole point of the ICM is to produce a single quantity, the forward model prediction error. We literally take the error produced from the loss function and use that as the intrinsic reward signal for our DQN. We can add this intrinsic reward to the extrinsic reward to get the final reward signal  $r_t = r_i + r_e$ . We can scale the intrinsic or extrinsic rewards to control the proportions of the total reward.

Below is a figure of the ICM with more detail and including the agent model (DQN).

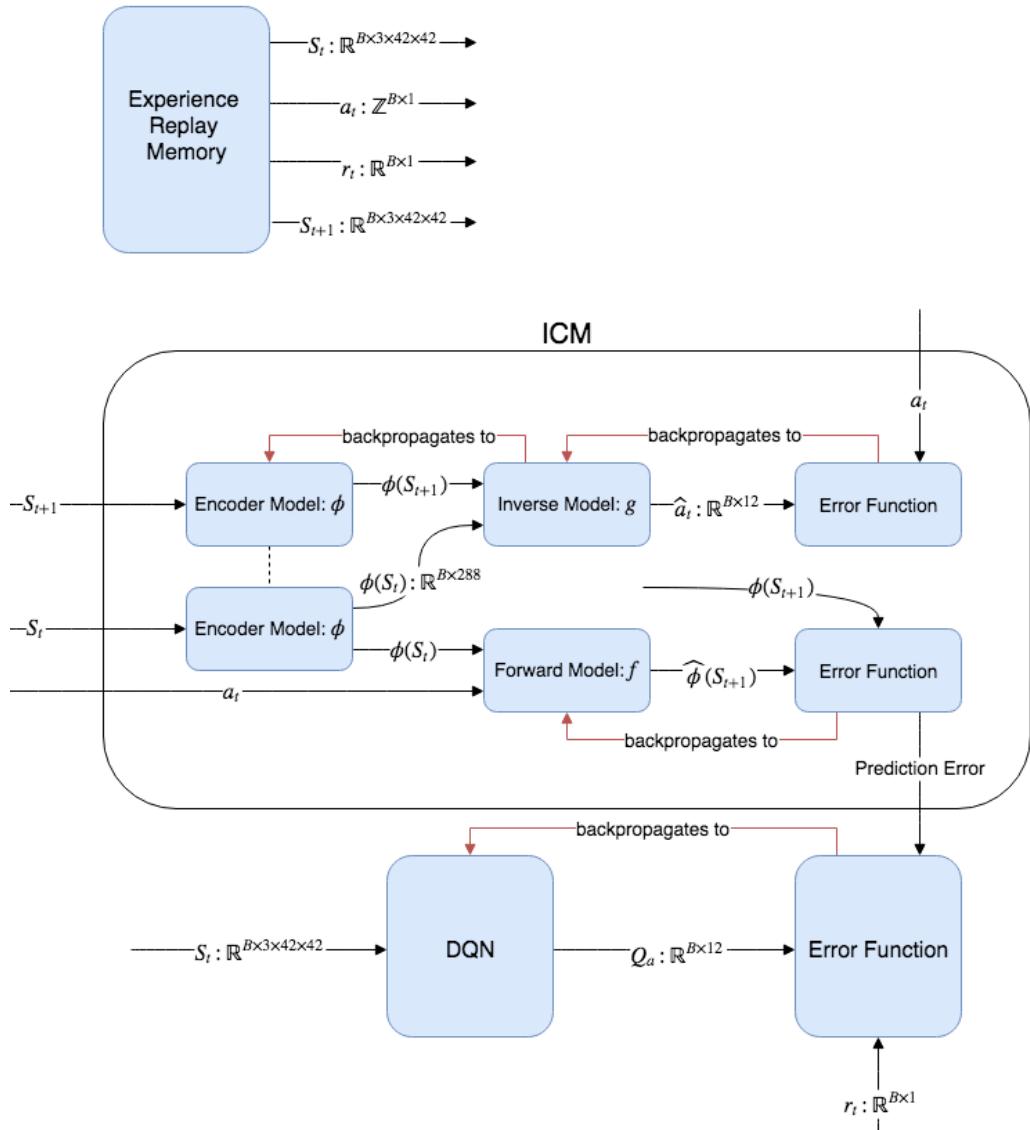


Figure 8.16: A complete view of the overall algorithm including the ICM. First we generate  $B$  samples from the experience replay memory and use these for the ICM and DQN. We run the ICM forward to generate a prediction error, which is then provided to the DQN's error function. The DQN then learns to predict action-values that reflect not only extrinsic (environment provided) rewards but also an intrinsic (prediction error-based) reward.

Let's see the code for the components of the ICM.

**Listing 8.7: ICM Components**

```

class Phi(nn.Module): #A
    def __init__(self):
        super(Phi, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)
        self.conv4 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=2, padding=1)

    def forward(self,x):
        x = F.normalize(x)
        y = F.elu(self.conv1(x))
        y = F.elu(self.conv2(y))
        y = F.elu(self.conv3(y))
        y = F.elu(self.conv4(y)) #size [1, 32, 3, 3] batch, channels, 3 x 3
        y = y.flatten(start_dim=1) #size N, 288
        return y

class Gnet(nn.Module): #B
    def __init__(self):
        super(Gnet, self).__init__()
        self.linear1 = nn.Linear(576,256)
        self.linear2 = nn.Linear(256,12)

    def forward(self, state1,state2):
        x = torch.cat( (state1, state2) ,dim=1)
        y = F.relu(self.linear1(x))
        y = self.linear2(y)
        y = F.softmax(y,dim=1)
        return y

class Fnet(nn.Module): #C
    def __init__(self):
        super(Fnet, self).__init__()
        self.linear1 = nn.Linear(300,256)
        self.linear2 = nn.Linear(256,288)

    def forward(self,state,action):
        action_ = torch.zeros(action.shape[0],12) #D
        indices = torch.stack( (torch.arange(action.shape[0]), action.squeeze()), dim=0)
        indices = indices.tolist()
        action_[indices] = 1.
        x = torch.cat( (state,action_) ,dim=1)
        y = F.relu(self.linear1(x))
        y = self.linear2(y)
        return y

```

#A Phi is the encoder network

#B Gnet is the inverse model

#C Fnet is the forward model

#D The actions are stored as integers in the replay memory so we convert to a one-hot encoded vector

None of these components have complicated architectures, they're fairly mundane, but together they form a powerful system. We've covered the ICM components, now let's put them together. We're going to define a function that accepts  $(S_t, a_t, S_{t+1})$ , and returns the forward model prediction error and the inverse model error. The forward model error will be used not only to backpropagate and train the forward model but also as the intrinsic reward for the DQN. The inverse model error is only used to backpropagate and train the inverse and encoder models. First we'll show the hyperparameter setup and instantiation of the models.

#### **Listing 8.7: Hyperparameters and Model instantiation**

```
params = {
    'batch_size':150,
    'beta':0.2,
    'lambda':0.1,
    'eta': 1.0,
    'gamma':0.2,
    'max_episode_len':100,
    'min_progress':15,
    'action_repeats':6,
    'frames_per_state':3
}

replay = ExperienceReplay(N=1000, batch_size=params['batch_size'])
Qmodel = Qnetwork()
encoder = Phi()
forward_model = Fnet()
inverse_model = Gnet()
forward_loss = nn.MSELoss(reduction='none')
inverse_loss = nn.CrossEntropyLoss(reduction='none')
qloss = nn.MSELoss()
all_model_params = list(Qmodel.parameters()) + list(encoder.parameters()) #A
all_model_params += list(forward_model.parameters()) +
    list(inverse_model.parameters())
opt = optim.Adam(lr=0.001, params=all_model_params)
```

#A We can add the parameters from each model into a single list and pass that into a single optimizer.

Some of the parameters in the `params` dictionary will look familiar, such as `batch_size`, but the others are probably not, we'll go over them, but let's take a look at the overall loss function.

Here's the formula for the overall loss for all 4 models (including the DQN):

$$\text{minimize } [\lambda \cdot Q_{\text{loss}} + (1-\beta) F_{\text{loss}} + \beta \cdot G_{\text{loss}}]$$

This formula just adds the DQN loss with the forward and inverse model losses each scaled by a coefficient. The DQN loss has a free scaling parameter  $\lambda$  whereas the forward and inverse model losses share a scaling parameter  $\beta$  so that they're inversely related. This is the only loss function we backpropagate through, so at each training step we backpropagate through all 4 models starting from this single loss function.

The `max_episode_len` and `min_progress` parameters are used to set a minimum amount of forward progress Mario must make or we'll reset the environment. Sometimes Mario will get stuck behind an obstacle and will just keep taking the same action forever, so if Mario doesn't move forward enough in a reasonable amount of time we just assume he's stuck and reset the environment.

During training if the policy function says to take action 3 (for example) then we will actually repeat that action 6 times (set according to the `action_repeats` parameter) instead of just once. This helps the DQN learn the value of actions more quickly. During testing (i.e. inference), we only take the action once. The gamma parameter is the same gamma parameter from the DQN chapter. When training the DQN, the target value is not just the current reward  $r_t$  but the highest predicted action-value for the next state, so the full target is  $r_t + \gamma \cdot \max(Q(S_{t+1}))$ . Lastly the `frames_per_state` is set to 3 since each state is the last 3 frames of the game play.

#### **Listing 8.8: The loss function**

```
def loss_fn(q_loss, inverse_loss, forward_loss):
    loss_ = (1 - params['beta']) * inverse_loss
    loss_ += params['beta'] * forward_loss
    loss_ = loss_.sum() / loss_.flatten().shape[0]
    loss = loss_ + params['lambda'] * q_loss
    return loss
```

Finally, we get to the actual ICM function.

#### **Listing 8.9: The ICM Prediction Error Calculation**

```
def ICM(state1, action, state2, forward_scale=1., inverse_scale=1e4):
    state1_hat = encoder(state1) #A
    state2_hat = encoder(state2)
    state2_hat_pred = forward_model(state1_hat.detach(), action.detach()) #B
    forward_pred_err = forward_scale * forward_loss(state2_hat_pred, \
                                                    state2_hat.detach()).sum(dim=1).unsqueeze(dim=1)
    pred_action = inverse_model(state1_hat, state2_hat) #C
    inverse_pred_err = inverse_scale * inverse_loss(pred_action, \
                                                    action.detach().flatten()).unsqueeze(dim=1)
    return forward_pred_err, inverse_pred_err
```

#A First we encode `state1` and `state2` using the `encoder` model

#B We run the forward model using the encoded states but we make sure to detach them from the graph

#C The inverse model returns a softmax probability distribution over actions

It must be repeated how important is to properly detach nodes from the graph when running the ICM. Recall that PyTorch (and pretty much all other machine learning libraries) builds a computational graph where nodes are individual tensors and connections between nodes are the operations between them. By calling the `.detach()` method we disconnect the tensor from the computational graph and treat it just like raw data; this prevents PyTorch from backpropagating through that node. If we don't detach the `state1_hat` and `state2_hat`

tensors when we run the forward model and its loss then the forward model will backpropagate into the encoder and will corrupt the encoder model.

We've now approached the main training loop. Remember, since we're using experience replay, training only happens when we sample from the replay buffer. We'll set up a function that samples from the replay buffer and computes the individual model errors.

#### **Listing 8.9: Minibatch training using experience replay**

```
def minibatch_train(use_extrinsic=True):
    state1_batch, action_batch, reward_batch = replay.get_batch()
    action_batch = action_batch.view(action_batch.shape[0],1) #A
    reward_batch = reward_batch.view(reward_batch.shape[0],1)

    forward_pred_err, inverse_pred_err = ICM(state1_batch, action_batch,
                                              state2_batch) #B
    i_reward = (1. / params['eta']) * forward_pred_err #C
    reward = i_reward.detach() #D
    if use_explicit: #E
        reward += reward_batch
    qvals = Qmodel(state2_batch) #F
    reward += params['gamma'] * torch.max(qvals)
    reward_pred = Qmodel(state1_batch)
    reward_target = reward_pred.clone()
    indices = torch.stack( (torch.arange(action_batch.shape[0]), \
                           action_batch.squeeze()), dim=0)
    indices = indices.tolist()
    reward_target[indices] = reward.squeeze()
    q_loss = 1e5 * gloss(F.normalize(reward_pred),
                          F.normalize(reward_target.detach()))
    return forward_pred_err, inverse_pred_err, q_loss
```

#A We reshape these tensors to add a single dimension to be compatible with the models

#B Run the ICM

#C Scale the forward prediction error using the Eta parameter

#D Start totaling up the reward; make sure to detach the i\_reward tensor

#E The use\_explicit boolean variable will let us decide whether or not to use explicit rewards in addition to the intrinsic reward

#F We compute the action-values for the next state and use this to add to our target reward term

#G Since the action\_batch is a tensor of integers of action indices, we convert this to a tensor of one-hot encoded vectors

Okay, let's tackle the main training loop. We initialize the first state using the `prepare_initial_state(...)` function we defined earlier which just takes the first frame and repeats it 3 times along the channel dimension. We also setup a `deque` instance to which we will append each frame as we observe them. The deque is set to a maxlen of 3 so only the most recent 3 frames are stored. We convert the deque first to a list then to a PyTorch tensor of dimension  $1 \times 3 \times 42 \times 42$  before passing to the Q-network.

#### **Listing 8.10: The Training Loop**

```
epochs = 5000
env.reset()
```

```

state1 = prepare_initial_state(env.render('rgb_array'))
eps=0.15
losses = []
episode_length = 0
switch_to_eps_greedy = 1000
state_deque = deque(maxlen=params['frames_per_state'])
e_reward = 0.
last_x_pos = env.env.env._x_position #A
for i in range(epochs):
    opt.zero_grad()
    episode_length += 1
    q_val_pred = Qmodel(state1) #B
    if i > switch_to_eps_greedy: #C
        action = int(policy(q_val_pred,eps))
    else:
        action = int(policy(q_val_pred))
    for j in range(params['action_repeats']): #D
        state2, e_reward_, done, info = env.step(action)
        last_x_pos = info['x_pos']
        if done:
            state1 = reset_env()
            break
        e_reward += e_reward_
        state_deque.append(prepare_state(state2))
    state2 = torch.stack(list(state_deque),dim=1) #E
    replay.add_memory(state1, action, e_reward, state2) #F
    e_reward = 0
    if episode_length > params['max_episode_len']: #G
        if (info['x_pos'] - last_x_pos) < params['min_progress']:
            done = True
        else:
            last_x_pos = info['x_pos']
    if done:
        ep_lengths.append(info['x_pos'])
        state1 = reset_env()
        last_x_pos = env.env.env._x_position
        episode_length = 0
    else:
        state1 = state2
    if len(replay.memory) < params['batch_size']:
        continue
    forward_pred_err, inverse_pred_err, q_loss = minibatch_train(use_extrinsic=False)
    #H
    loss = loss_fn(q_loss, forward_pred_err, inverse_pred_err) #I
    loss_list = (q_loss.mean(), forward_pred_err.flatten().mean(), \
                inverse_pred_err.flatten().mean())
    losses.append(loss_list)
    loss.backward()
    opt.step()

```

#A We need to keep track of the last x position in order to reset if no forward progress

#B Run the DQN forward to get action-value predictions

#C After the first 1000 epochs, switch to epsilon-greedy policy

#D Repeat whatever action the policy says 6 times to speed up learning

#E Convert the deque object into a tensor

#F Add the single experience to the replay buffer

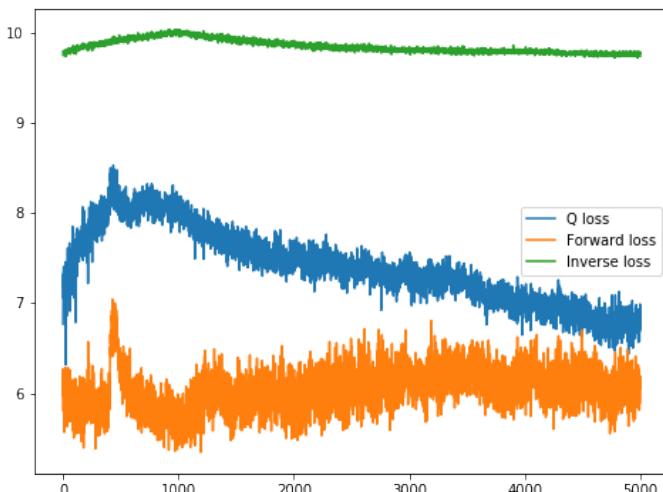
#G If Mario is not making sufficient forward progress, restart the game and try again

```
#H Get the errors for one minibatch of data from the replay buffer
#I Compute the overall loss
```

While a bit lengthy, this training loop is pretty simple. All we do is prepare a state, input to the DQN, get action-values (Q-values), input to the policy, get an action to take, and then call the `env.step(action)` method to perform the action. We then get the next state and some other metadata. We add this full experience as a tuple  $(S_t, a_t, r_t, S_{t+1})$ . to the experience replay memory. Most of the action is happening in the minibatch training function we already covered.

That is all the main code you need to build an end-to-end DQN and ICM to train on Super Mario Bros. Let's test it out by training for 5000 epochs, which takes about 30 or so minutes running on a MacBook Air (no GPU). We will train with `use_extrinsic=False` in the minibatch function, so it is learning only from the intrinsic reward. You can plot the individual losses for each of the ICM components and the DQN with the following code. We log-transform the loss data to keep them on a similar scale.

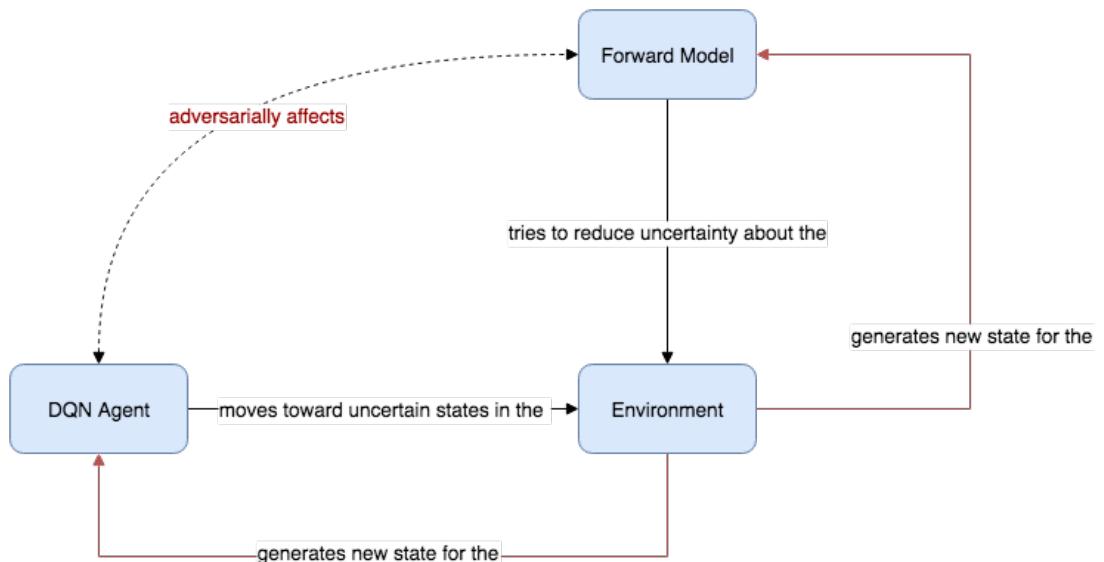
```
>>> losses_ = np.array(losses)
>>> plt.figure(figsize=(8,6))
>>> plt.plot(np.log(losses_[:,0]),label='Q loss')
>>> plt.plot(np.log(losses_[:,1]),label='Forward loss')
>>> plt.plot(np.log(losses_[:,2]),label='Inverse loss')
>>> plt.legend()
>>> plt.show()
```



**Figure 8.17:** These are the losses for the individual components of the ICM and the DQN. The losses do not smoothly decrease like we're used to with a single supervised neural network because the DQN and ICM are trained adversarially.

The DQN loss is clearly trending downward after initially increasing. The forward loss just looks pretty noisy with no clear up or down trend. The inverse model looks sort of flat-lined but if you zoom in, it does seem to very slowly decrease over time. The loss plots look a lot nicer if you set `use_extrinsic=True` and use the extrinsic rewards. But don't be let down by the loss plots. If we test the trained DQN, you will see it does a lot better than the loss plots suggest. This is because the ICM and DQN are behaving like an adversarial dynamical system since the forward model is trying to lower its prediction error but the DQN is trying to maximize the prediction error (by steering the agent toward unpredictable states of the environment).

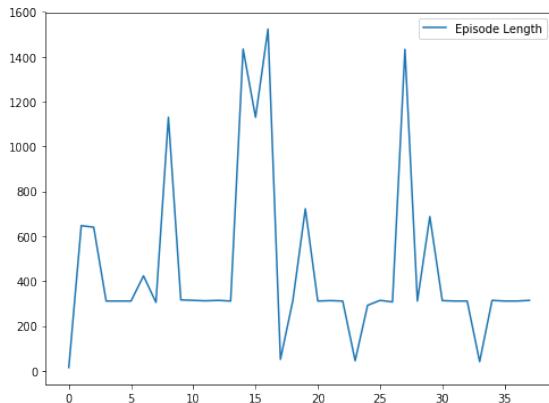
If you look at the loss plot for a generative adversarial network (GAN), the generator and discriminator loss look somewhat similar to our DQN and forward model loss with `use_extrinsic=False`. The losses do not smoothly decrease like you're used to when you train a single machine learning model.



**Figure 8.18:** The DQN agent and forward model are trying to optimize antagonistic objectives and hence form an adversarial pair.

A better assessment of how well the overall training is going is to track the episode length over time. The episode length should be increasing if the agent is learning to how to progress through the environment more effectively. In our training loop, whenever the episode finished (i.e. the `done` variable becomes True because agent dies or agent doesn't make sufficient forward progress), we save the current `info['x_pos']` to the `ep_lengths` list. We expect to see that the maximum episode lengths will get longer and longer over training time.

```
>>> plt.figure()
>>> plt.plot(losses_[:,3], label='Episode length')
```



**Figure 8.19:** Training time is on the x-axis and episode length is on the Y-axis. We see bigger and bigger spikes over training time, which is what we expect.

We see that early on the biggest spike is getting to the 650 mark (i.e. x position in the game), but by the 15<sup>th</sup> episode the agent is able to get farther than the 1500 x position.

The episode length plot looks promising, but let's actually render a video of our trained agent playing Super Mario Bros. If you're running this off your own computer, the OpenAI Gym provides the render function which will open a new window with live game play. Unfortunately, this won't work if you're using a remote machine or cloud virtual machine. In those cases, the easiest alternative is to just use `plt.imshow(env.render("rgb_array"))` in a loop.

#### **Listing 8.11: Testing the trained agent**

```
eps=0.1
done = True
state_deque = deque(maxlen=params['frames_per_state'])
for step in range(5000):
    if done:
        env.reset()
        state1 = prepare_initial_state(env.render('rgb_array'))
    q_val_pred = Qmodel(state1)
    action = int(policy(q_val_pred,eps))
    state2, reward, done, info = env.step(action)
    state2 = prepare_multi_state(state1,state2)
    state1=state2
    env.render()
env.close()
```

There's not much to explain here if you followed the training loop since we're just extracting the part that runs the network forward and takes an action. Notice we still use an epsilon greedy policy with epsilon set to 0.1. Even during inference the agent needs a little bit of randomness to keep it from getting stuck. One difference to notice is that in test (or inference) mode, we only enact the action once and not 6 times like we did in training. Assuming you get the same results as us, your trained agent should impressively make fairly consistent forward progress and should be able to jump over obstacles. Congratulations!



Figure 8.20: Mario agent trained only from intrinsic reward successfully jumping over a chasm. This demonstrates it has learned basic skills without any explicit rewards to do so. With a random policy the agent would not even be able to move forward let alone learn to jump over obstacles.

If you're not getting the same results, try changing the hyperparameters, particularly the learning rate, minibatch size and maximum episode length and minimum forward progress. Training for 5000 epochs with intrinsic reward only works but in our experience is sensitive to these hyperparameters.

**HOW WILL THIS WORK IN OTHER ENVIRONMENTS?** While we trained a DQN agent with an ICM-based reward on a single environment, Super Mario Bros, the paper "Large-Scale Study of Curiosity-Driven Learning" Burda et al 2018 demonstrated how effective intrinsic rewards alone can be. They ran a number of

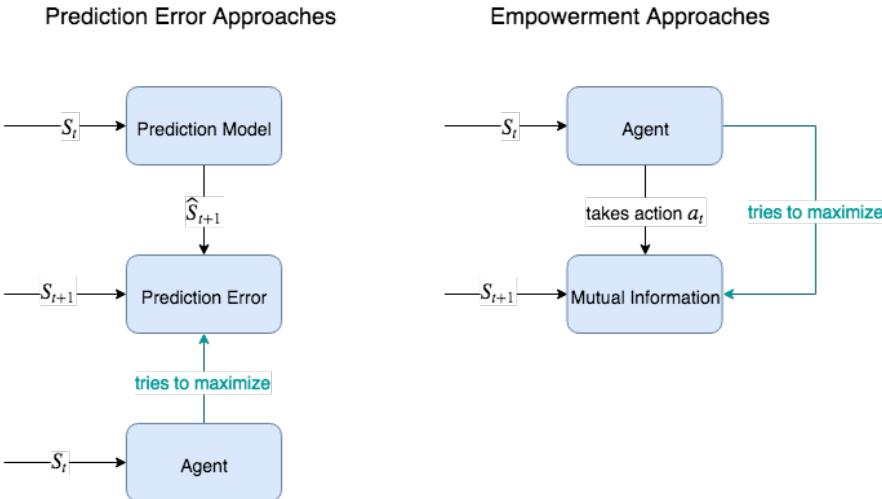
experiments using curiosity-based rewards only across multiple games, finding that a curious agent could progress through 11 levels in Super Mario Bros. and learn to play pong, among others. They used essentially the same ICM we just built, except they used a more sophisticated agent model called proximal policy optimization (PPO) rather than DQN. An experiment you can try is to replace the encoder network with a random projection. A random projection just means multiplying the input data by a randomly initializing matrix (e.g. a randomly initialized neural network that is fixed and not trained). The Burda et al 2018 paper demonstrated that a random projection works almost as well as the trained encoder.

## 8.7 Alternative Intrinsic Reward Mechanisms

In this chapter we've described the serious problem faced by RL agents in environments with sparse rewards. We considered the solution as imbuing agents with a sense of curiosity. In this chapter, we implemented an approach from the Pathak et al 2017 paper, one of the most widely cited papers in reinforcement learning research in recent years. We chose to demonstrate this approach not just because it is popular, but because it builds off what we've learned in previous chapters without introducing too many new notions. Curiosity-based learning (which goes by many names) is a very active area of research and there are many alternative approaches, some of which we think are better than the ICM.

Many of the other exciting methods use Bayesian inference and information theory to come up with novel mechanisms to drive curiosity. The prediction error (PE) approach we used in this chapter is just one implementation under a broader PE umbrella. The basic idea, as we now know, is that the agent wants to reduce its PE (or in other words, its uncertainty about the environment) but it must do so by actively seeking out novelty lest it be surprised by something unexpected.

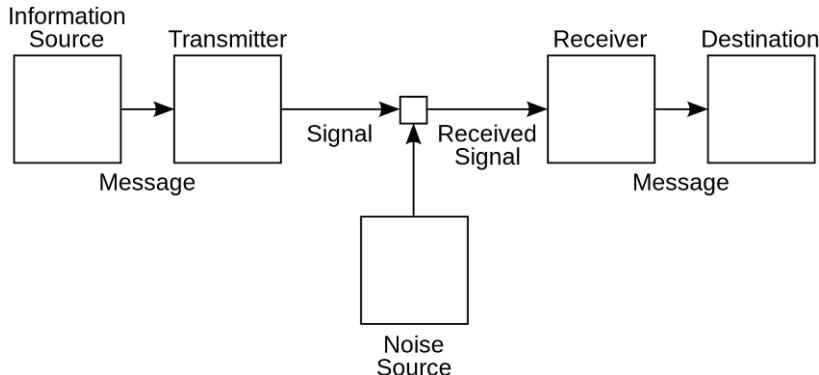
Another umbrella is that of agent *empowerment*. Rather than seeking to minimize prediction error and make the environment more predictable, empowerment strategies optimize the agent to maximize its control over the environment. One paper in this area is "Variational Information Maximisation for Intrinsically Motivated Reinforcement Learning" by Mohamed et al 2015. We can make this informal statement about maximizing control over the environment into a precise mathematical statement (which we will only approximate here).



**Figure 8.21:** The two main approaches for solving the sparse reward problem with curiosity-like methods are prediction error methods like the one we used in this chapter and empowerment methods. Rather than trying to maximize the prediction error between a given state and the next predicted state, in empowerment methods the goal is to maximize the mutual information (MI) between the agent's actions and the next states. If the MI between the agent's action and the next state is high, then that means the agent has a high-level of control (or power) over the resulting next states, i.e. if you know which action the agent took you can predict the next state well. This incentivizes the agent to learn how to maximally control the environment.

The premise relies on the quantity called mutual information. We will not define it mathematically here, but informally, mutual information (MI) measures how much information is shared between two sources of data called random variables (because usually we deal with data that has some amount of randomness or uncertainty). Another less tautological definition is that MI measures how much your uncertainty about one quantity  $x$  is reduced given another quantity  $y$ .

Information theory was first developed with real world communication problems in mind, where one problem is how to best encode messages across a possibly noisy communication channel so that the received message is the least corrupted. So we have an original message  $x$  we want to send across a noisy communication line (for example, using radio waves) and we want to maximize the mutual information between  $x$  and the received message  $y$ . We do this by developing some way of encoding  $x$ , which may be a textual document for example, into a pattern of radio waves that minimizes the probability of the data being corrupted by noise. So once someone else receives the de-coded message  $y$ , they can be assured that their received message is very close to the original message.



**Figure 8.22: Claude Shannon developed communication theory, which was born from the need to encode messages efficiently and robustly across noisy communication channels as depicted here. The goal is to encode the message such that the mutual information between the received message and the sent message is maximal.**

In the previous example  $x$  and  $y$  were both some sort of written message, but  $x$  and  $y$  need not be the same type of quantities. For example, we can ask what the mutual information is between the 1 year stock price history of a company and its annual revenue, i.e. if I start with a very uncertain estimate about the annual revenue of a company and then I learn the 1 year stock price history, how much is my uncertainty reduced? If it's reduced a lot, then the MI is high. These are different quantities but both using the units of dollars, but that need not be the case either. We could ask what the mutual information is between the daily temperature and the sales of ice cream shops.

In the case of agent empowerment in reinforcement learning, the objective is to maximize the mutual information between an action (or sequence of actions) and the resulting future state(s). Maximizing this objective means that if you know what action the agent took, you will have a high confidence about what the resulting state was, which means the agent has a high degree of control over the environment since it can reliably reach states given its actions. Hence, a maximally empowered agent has the maximal degrees of freedom.

This is different than the prediction error approach because minimizing PE directly encourages exploration, whereas maximizing empowerment may induce exploratory behavior as a means to learn empowering skills, but only indirectly. Consider a young woman, Sarah, who decides to travel the world and explore as much as possible. She is reducing her uncertainty about the world. Compare her to Bill Gates, who by being extraordinarily rich, has a high degree of power. He may not be interested in traveling as much as Sarah, but he can if he wants and no matter where he is at any time he can go where he wants to go.

Both empowerment and curiosity objectives have their use cases. Empowerment-based objectives have been shown to be useful for training agents to acquire complex skills without any extrinsic reward (e.g. robotic tasks or sports games), whereas curiosity-based objectives

tend to be more useful for exploration (e.g. games like Super Mario Bros. where the goal is to progress through levels). In any case, these two metrics are more similar than they are different.

## 8.8 Summary

- In this chapter we learned about the problem of sparse rewards and covered techniques aimed to solving it.
- We implemented a Deep Q-network agent and paired it with an intrinsic curiosity module.
- The intrinsic curiosity module consists of three independent neural networks: a forward prediction model, inverse model and encoder.
- The encoder encodes high-dimensional states into a low-dimensional vector with high-level features (removes noise and trivial features)
- The forward prediction model predicts the next encoded state and its error provides the curiosity signal
- The inverse model trains the encoder by taking two successive encoded states and predicting the action that was taken
- We learned about the broader subfield of intrinsic motivation in reinforcement learning agents and discussed the empowerment approach as an alternative to the ICM.