Deep Reinforcement Learning IN ACTION

Alex Zai
Brandon Brown

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Deep Reinforcement Learning in Action
Version 2

For more information on this and other Manning titles go to
www.manning.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: [URL] We hope you'll enjoy the ride!

— Alex Zai and Brandon Brown

brief contents

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 Minksy, 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.

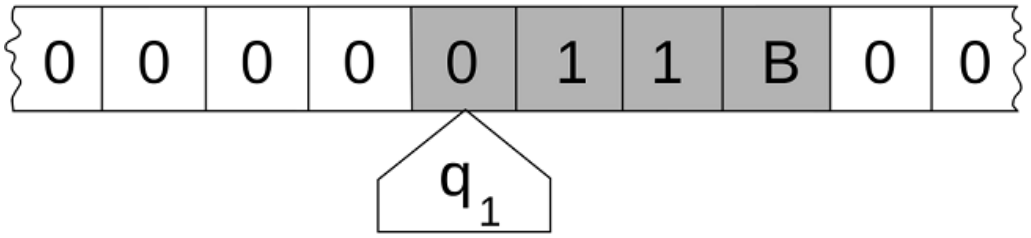

Figure 1.1: This depicts how a single-tape Turing machine works. The shape labeled "q1" is a read and write head that can move left or right across a tape that has printed values in evenly spaced cells. The read/write head follows a predefined set of rules and can read certain values on the tape, erase, 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.

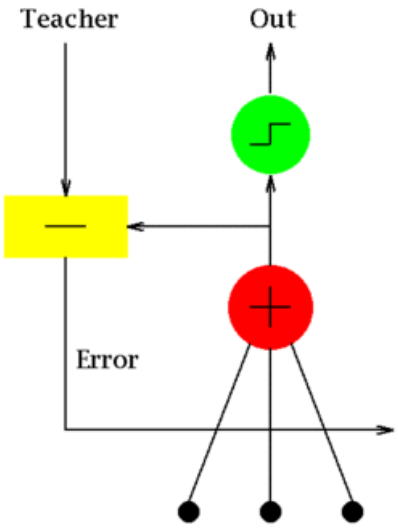

Figure 1.2: A diagram of one of the first neural networks called ADALINE that was used to filter out echoes in telephone lines. The network simply computed a weighted sum of its input variables and if the sum was greater than some threshold parameter, it would return 1, otherwise it returns 0 (called 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.

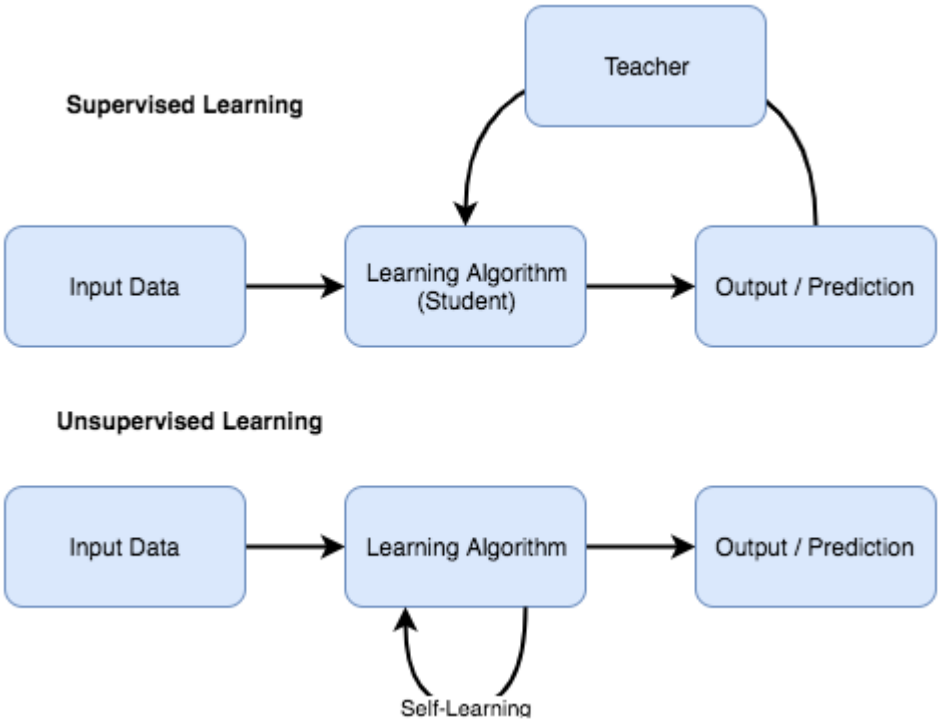

Figure 1.3: Supervised learning is the class of algorithms that exhibit a teacher-student relationship where the algorithm learns by being given explicit feedback by a teacher. Unsupervised learning algorithms are able to self-teach to some extent, and learn something about data on their own.

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.

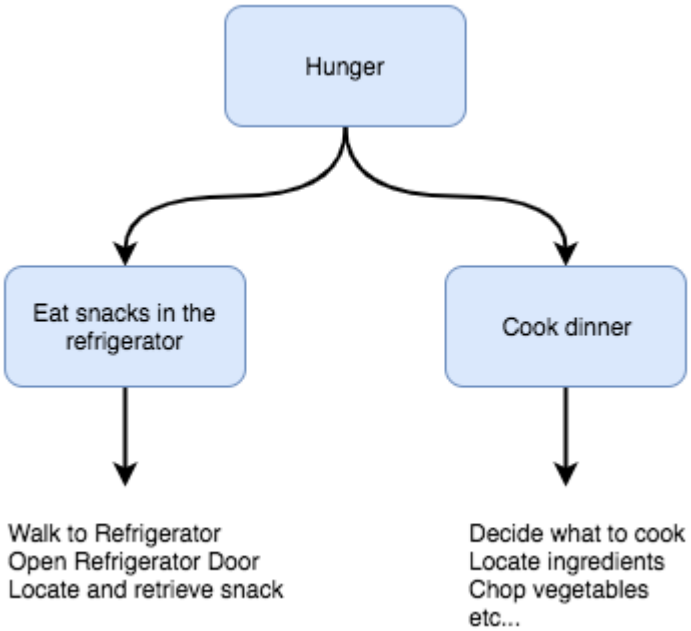

Figure 1.4: A high-level goal like satisfying one's hunger can be broken down into a tree of "local" sub-goals that each must be solved in order to achieve the global goal.

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 memo**r**ization, 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 then-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 accesible 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.

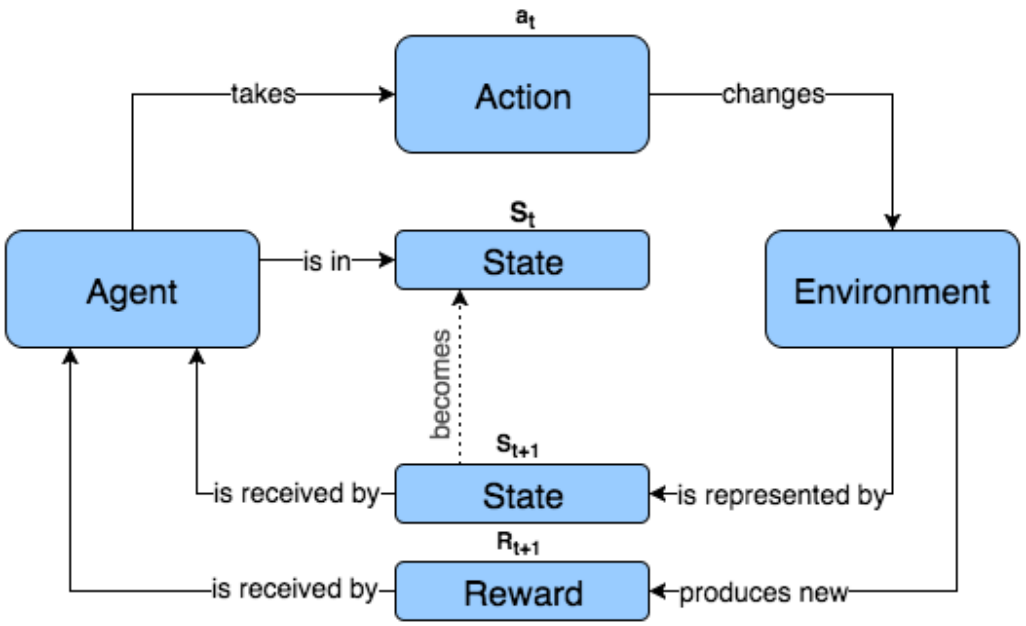

Figure 1.5: The standard model for reinforcement learning algorithms. RL algorithms involve an agent that learns the best way to interact in an environment. The agent takes an action in the environment - such as moving a chess piece - which then updates the state of the environment. For every action 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.

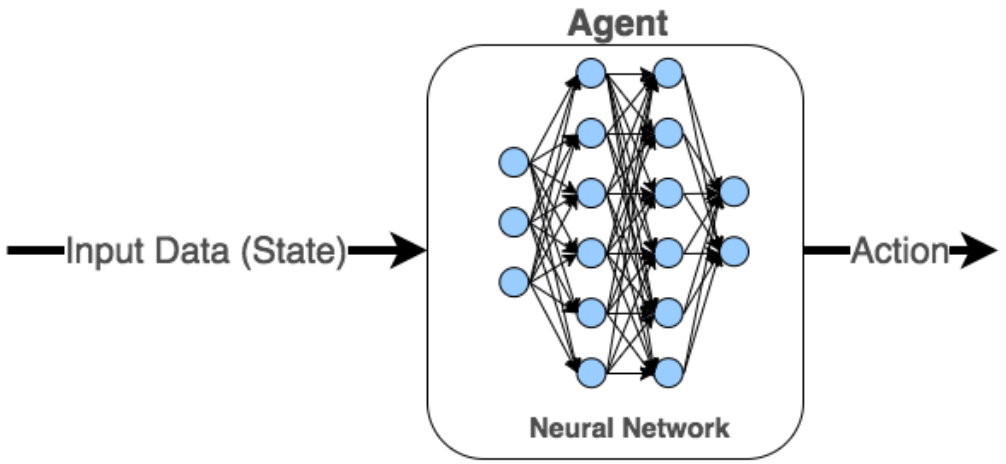

Figure 1.6: The input data (which is the state of the environment at some time point) is fed into the agent, which is implemented as a deep neural network in this book, which then evaluates that data to take an action. This process is a little more involved than shown here, but this captures the essence.

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 interaction 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()
```

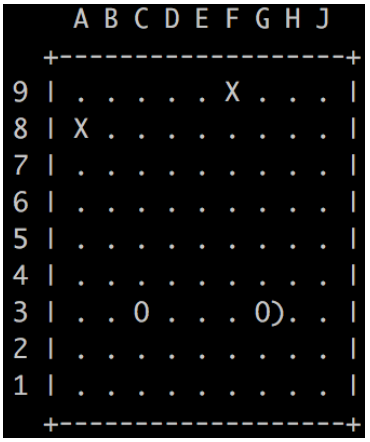

Figure 1.7. The OpenAI python library comes with many environments and an easy interface for a learning algorithm to interact with them. With just a few lines of code we've loaded up a 9x9 Go board.

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.

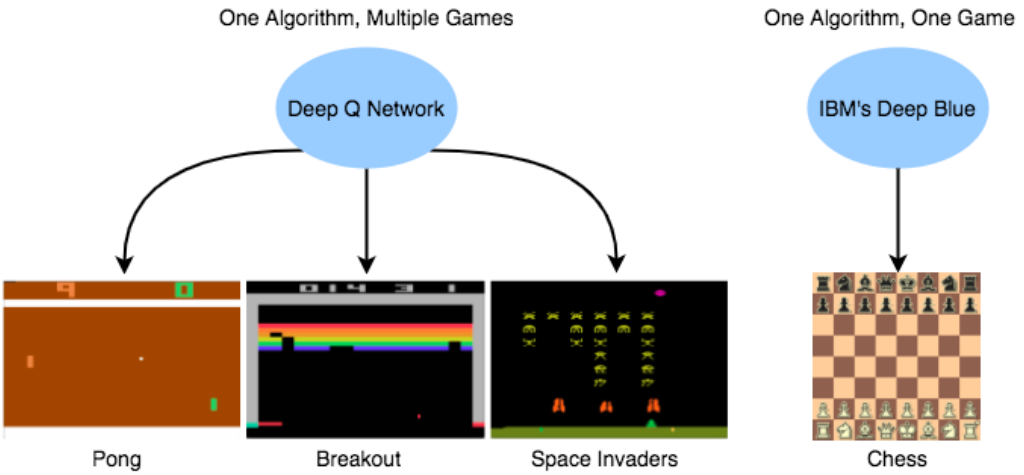

Figure 1.8: DeepMind's DQN algorithm successfully learned how to play seven Atari games with only the raw pixels as input and the players score as the objective to maximize. Previous algorithms, such as IBM's Deep Blue, needed to be fine-tuned to play a specific game.

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.

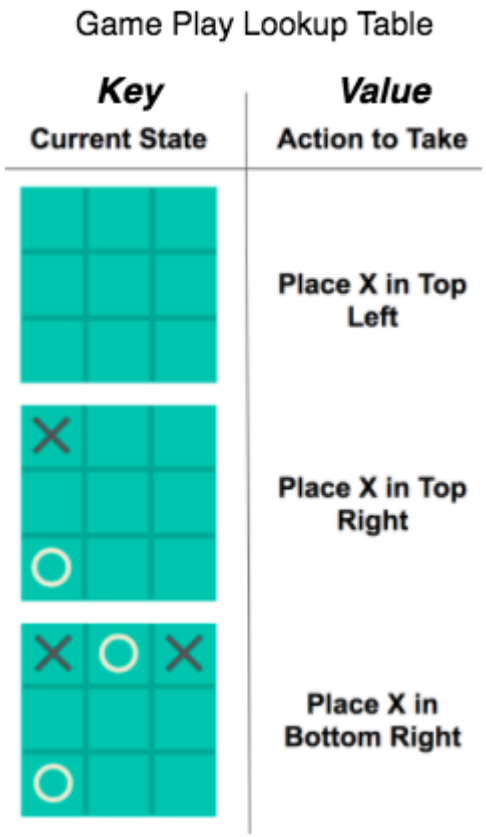

Figure 1.9: An action lookup table for Tic-Tac-Toe with only three entries where the player plays X. When the player is given a board position, the lookup table dictates the move that they should make next. There will be an entry for every possible state in the game.

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 84×84 grey-scaled images at each step, which would lead to 256^{28228} unique game states (256 different shades of grey per pixel, 4*84*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 210x160 pixel color images.

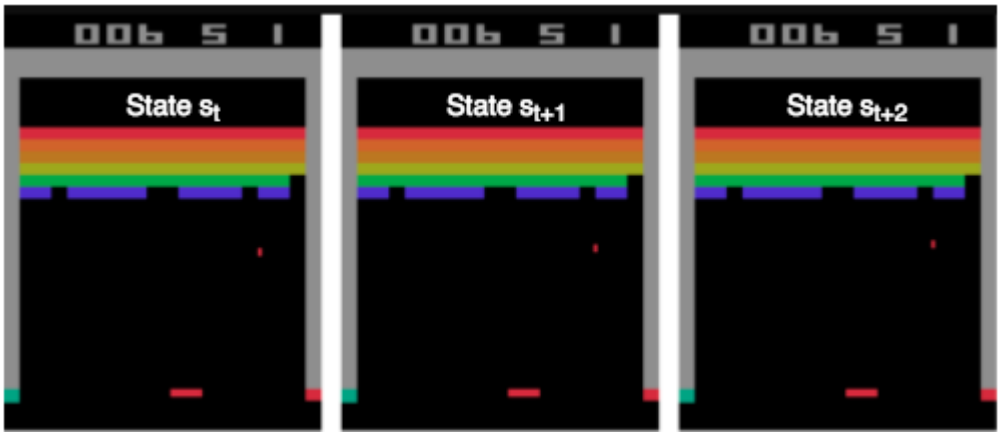

Figure 1.10. A series of three frames of breakout. The placement of the ball is slightly different in each frame. If using a lookup table, this would equate to storing three unique entries in the lookup table. A lookup table would be impractical as there are way too many game states to store.

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 8x8 filters, 32 4x4 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.

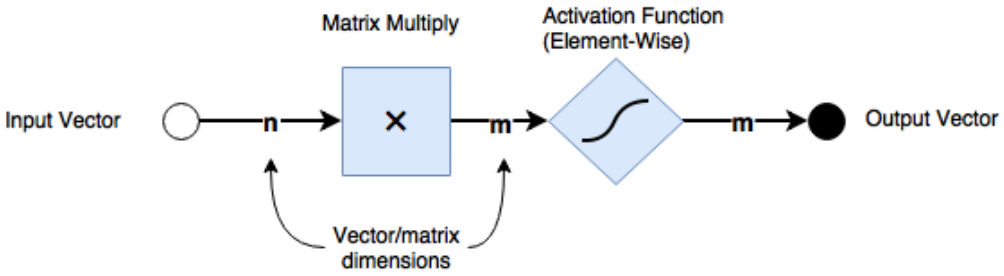

Figure 1.11: A string or flow diagram for a single layer neural network. Reading from left to right, this represents a function that accepts an input vector of dimension *n*, multiplies it by a matrix of dimensions *n* x *m*, returning a vector of dimension *m*. Then a non-linear (activation) function 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.

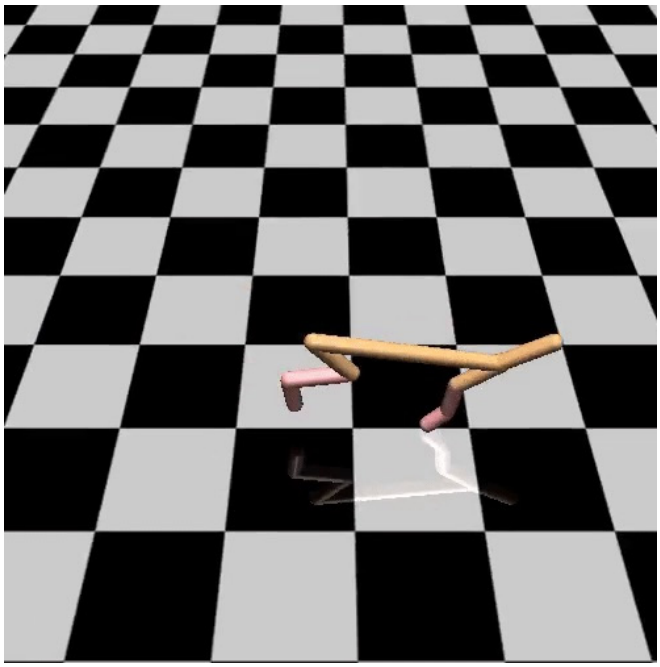

Figure 1.12. The Half-Cheetah Mujoco environment from the OpenAI Gym. We will be implement a Policy Gradient algorithm to solve them in Chapter 6.

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 >

- "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" 2nd 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