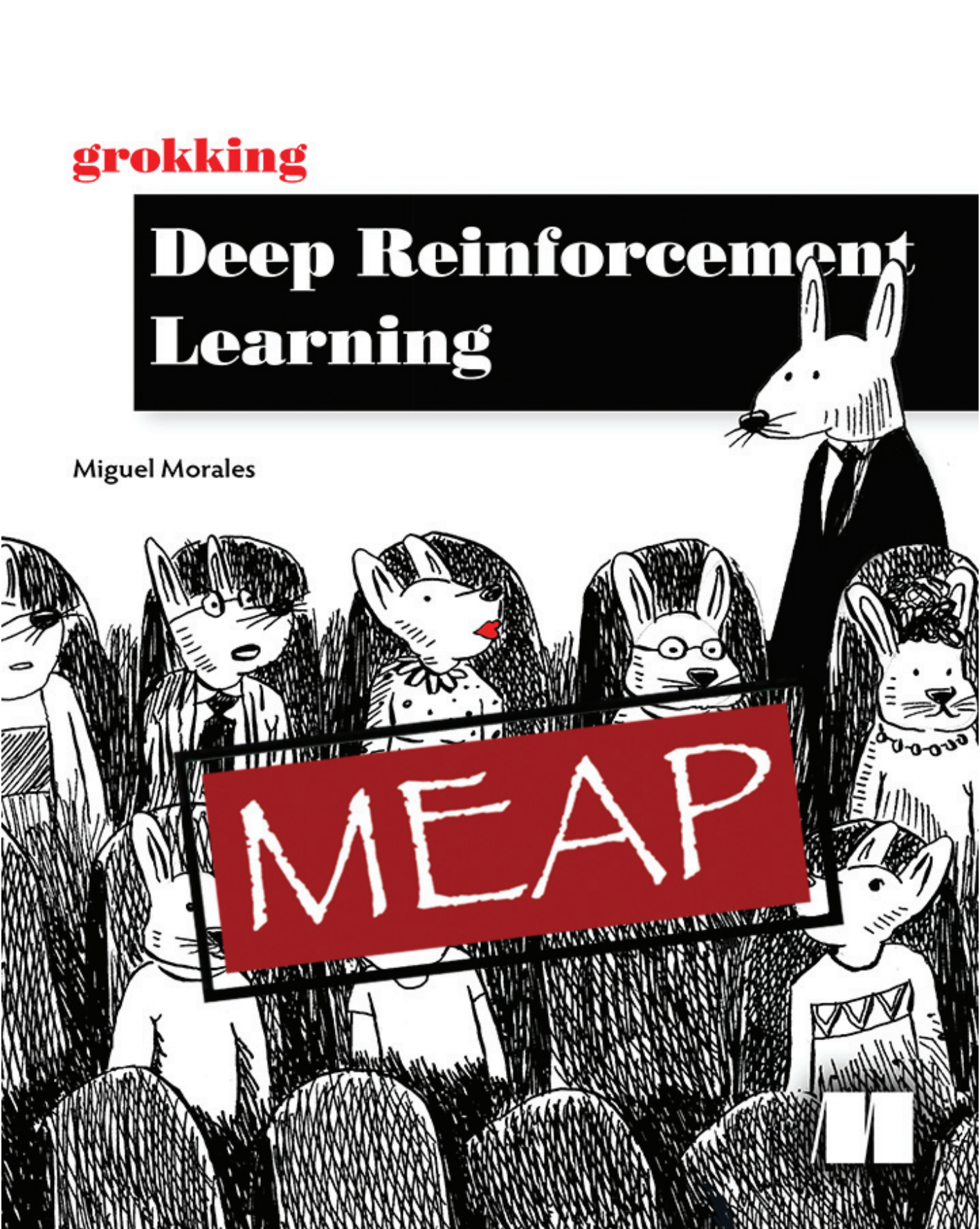
grokking

Deep Reinforcement Learning

Miguel Morales



MEAP Edition
Manning Early Access Program
Grokking Deep Reinforcement Learning
Version 2

For more information on this and other Manning titles go to
www.manning.com

welcome

Thanks for purchasing the MEAP for *Grokking Deep Reinforcement Learning*. My vision is that by buying this book, you will not only learn deep reinforcement learning but also become an active contributor to the field. Deep reinforcement learning has the potential to revolutionize the world as we know it. By removing humans from decision-making processes, we set ourselves up to succeed. Humans can't match the stamina and work ethic of a computer; we also have biases that make us less than perfect. Imagine how many decision-making applications could be improved with the objectivity and optimal decision making of a machine—healthcare, education, finance, defense, robotics, etc. Think of any process in which a human repeatedly makes decisions; deep reinforcement learning can help in most of them. Deep reinforcement learning can do great things as it is today, but the field is still not perfect. That should excite you, because it means we need people with the interest and skills to push the boundaries of this field forward. We are lucky to be part of this world at this point, and we should take advantage of it and make history. Are you up for the challenge?

I've been involved in Reinforcement Learning for a few years now. I first studied the topic in a course at Georgia Tech: Reinforcement Learning and Decision Making, which was co-taught by Drs. Charles Isbell and Michael Littman. It was inspiring to hear from top researchers in the field, interact with them daily, and listen to their perspectives. The following semester, I became a Teaching Assistant for the course and never looked back. Today, I'm an Instructional Associate at Georgia Tech and continue to help with the class daily. I've been privileged to interact with top researchers in the field and with hundreds of students, and I've become a bridge between the experts and the students for almost two years now. I understand the gaps in knowledge, the topics that are often the source of confusion, the students' interests, the foundational knowledge that is classic yet necessary, the classical papers that can be skipped, and many other things that put me in a position to write this book. In addition to teaching at Georgia Tech, I work full-time for Lockheed Martin, Missile and Fire Control - Autonomous Systems. We do top autonomy work, part of which involves the use of autonomous decision-making such as in deep reinforcement learning. I felt inspired to take my passion for both teaching and deep reinforcement learning to the next level by making this field available to anyone who is willing to put in the work.

I partnered with Manning to deliver a great book to you. Our goal is to help readers understand how deep learning makes reinforcement learning a more effective approach. In the first part of the book, we will dive into the foundational knowledge specific to reinforcement learning. Here you'll gain the necessary expertise to solve more complex decision-making problems. In the second part, I'll teach you to use deep learning techniques to solve massive, complex reinforcement learning problems. We will dive into the top deep reinforcement learning algorithms and dissect them one at a time. Finally, in

the third part, we will look at advanced applications of these techniques. We will put everything together then and help you see the potential of this technology.

Again, it is an honor to have you with me; I hope that I can inspire you to give your best and apply the knowledge you will obtain in this book to solve complex decision-making problems and make this a better place. Humans may be sub-optimal decision makers, but buying this book was without a doubt the right thing to do. Let's get working.

—Miguel Morales

brief contents

Introduction to Deep Reinforcement Learning 1

IN THIS CHAPTER

You'll learn what deep reinforcement learning is and where it comes from laying the foundation for later chapters.

You'll understand how deep reinforcement learning is embedded in a larger field of related approaches and how these relationships influence this field.

You'll recognize how this approach is different to other machine learning approaches and why it is important.

You'll identify what deep reinforcement learning can accomplish for a variety of problems.

"I visualize a time when we will be to robots what dogs are to humans, and I'm rooting for the machines."

— Claude Shannon
Father of the Information Age and
the field of Artificial Intelligence

Humans and animals naturally pursue feelings of happiness. From picking our daily meals to going after long-term goals, every action we choose is derived from our drive to experience rewarding moments in life. Whether these moments are self-centered pleasures or the more altruistic of goals, they are still our perception of how important and rewarding they are. And this is, to some extent, our reason for living.

Our ability to achieve these rewarding moments, especially those that take time to come to fruition, seems to be correlated with intelligence; intelligence being defined as the ability to *acquire* and *apply* knowledge and skills. People that are deemed by society as intelligent display an ability to balance immediate and long-term goals. Goals that take longer to materialize are normally the hardest to achieve, and it is those who are able to withstand the challenges along the way that are the exception, the leaders, the intellectuals of society.

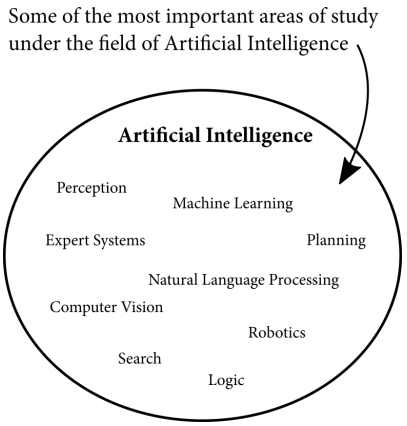
In this book, you will learn a computer science approach known as deep reinforcement learning. Deep reinforcement learning studies the design and creation of machine agents that can mimic human intelligence by receiving a stream of data, acting in the environment around them and learning from trial and error.

What is Deep Reinforcement Learning?

Deep reinforcement learning is part of a broader field—artificial intelligence—that studies the creation of intelligent computer programs. Before we dive into the specifics of deep reinforcement learning, let's make sure you understand how it fits into this branch of computer science.

Artificial Intelligence

Artificial intelligence (AI) is a branch of computer science that studies computer programs capable of displaying intelligence. "Intelligence" may seem like a broad term, but traditionally, any piece of software that displays cognitive abilities such as perception, search, planning, and learning is considered part AI. Some examples



of functionality produced by AI software are:

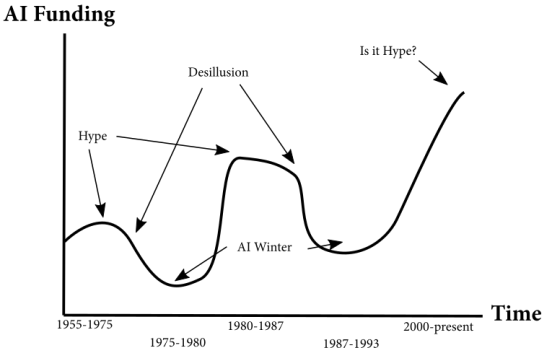
- The pages to be most likely returned by your search engine of choice.
- The route produced by your GPS app.
- The voice recognition and the artificial voice of your phone assistant app.
- The list of items related to your most recent purchase at your e-commerce site of choice.
- The follow-me feature on some of the most popular brands of drones.

Humans have been intrigued by the possibility of intelligent creatures other than ourselves since antiquity, but Alan Turing's work in the 1940s and '50s paved the way for modern computer science and AI by laying down important theoretical foundations that later scientists leveraged. The most well known of these is the **Turing Test**, which proposes a standard for measuring machine intelligence: if a human interrogator is unable to distinguish a machine from another human on a chat Q&A session, then the computer is said to count as intelligent. Though rudimentary, the Turing Test allowed generations to wonder about the possibilities of creating intelligent machines by setting a goal that researchers could pursue.

Also starting in the 1950s, an influential AI researcher named John McCarthy made several important contributions to the field. To name a few, McCarthy is credited with coining the term "artificial intelligence" in 1955, leading the first AI conference in 1956, inventing the Lisp programming language in 1958, cofounding the MIT AI Lab in 1959, and contributing major papers on AI over several decades.

All the work and progress at the time created a great deal of excitement, but there were major setbacks. Prominent researchers bet that we would be able to create an agent with human-like intelligence within just a few years, but this was an overly optimistic expectation. To make things worse, a well-known researcher named James Lighthill compiled a report criticizing the state of academic research in

Beyond actual numbers, AI has followed a pattern of hype and desillusion for years. What does the future hold?



artificial intelligence. These things contributed to a long period of reduced funding and interest in artificial intelligence research known as the first AI winter. The field has continued this pattern throughout the years: Researchers make progress, then overestimate and overcommit to results and miss deadlines, and this leads the government and industry partners to reduce or altogether cut off research funding. The recent hype around AI might lead us to worry about another winter, but the success of AI in recent years seems different.

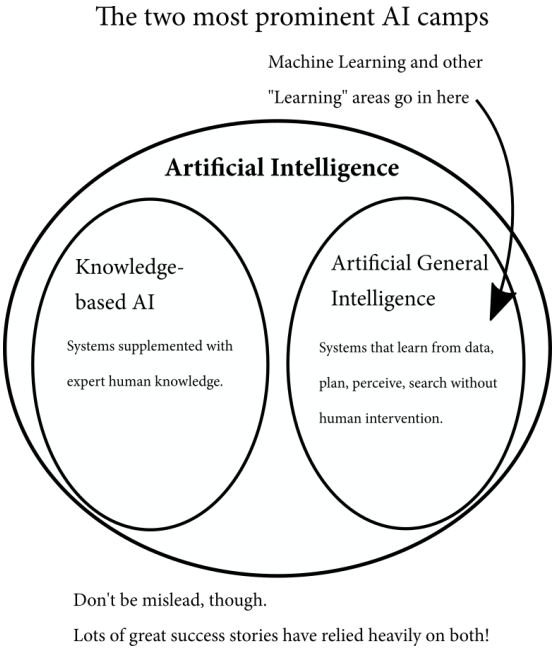
Today, the most powerful companies in the world make the largest investments to AI research. Companies like Google, Facebook, Microsoft, Amazon and Apple have invested in AI research since their founding and have become highly profitable thanks, in part, to the AI systems they've developed and acquired. Their large and steady investments have created the perfect environment for the current pace of AI research. Current researchers have the best computing power available and large amounts of data for their research, and teams of top researchers are working together on the same problems, in the same location, at the same time. Current AI research has become more stable and more productive. We have been witnessing one AI success after another, and it is not likely to stop.

Regardless of what happens in the next decades, artificial intelligence will be here for the long haul. For many, artificial intelligence is not only about creating intelligent computer software; it also addresses a more philosophical need to understand ourselves, our existence—how we think and why we do things the way we do.

Machine Learning

There are two prominent camps of artificial intelligence, one side of the spectrum a pragmatic approach, the other side a purist one. The pragmatic approaches were commonly referred to as *strong* AI because it usually performed better than the purist, *weak* AI, approaches in real-world problems. On the pragmatic side, we have **knowledge-based AI** (KBAI). KBAI consists of computer scientists who study and create AI systems supplemented with human expert knowledge. The goal of this approach is to specialize, to create expert, knowledge-based systems. It is considered strong despite the injection of human expertise. On the other side of the spectrum, we have **artificial general intelligence** (AGI). The goal of AGI, a more purist approach to creating intelligent systems, is to create an intelligent agent that can learn on its own and from scratch on a wide-ranging set of tasks. In the past, these meth-

ods were limited by low computing power and data availability, so AGI was known as weak AI. But, as technology progresses, human expertise becomes the bottleneck and the purist methods previously seen as weak become better and stronger. Deep reinforcement learning, as you will see soon, is part of the AGI side and in this book we will devote no time to KBAI.



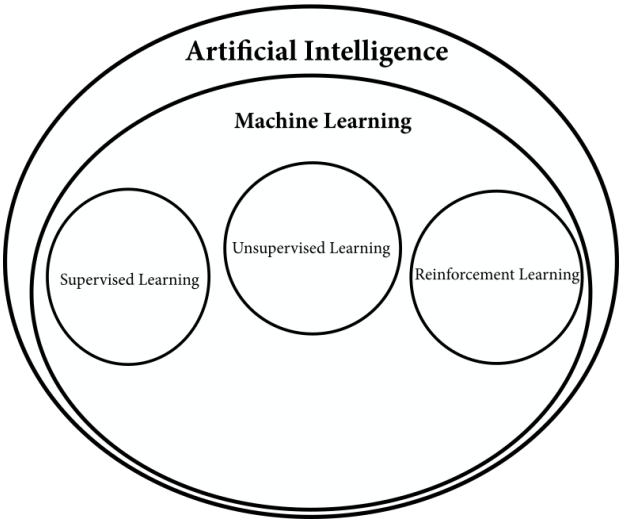
Machine learning (ML) is part of the purist AI camp as it is about learning from data. There are three main ML branches: supervised learning, unsupervised learning, and reinforcement learning. **Supervised learning** (SL) is the task of learning from labeled data. In SL, a human decides which data to collect and how to label it. A basic example of SL would be a program that can identify pictures of cats. In SL, the model learns to generalize to unseen samples. For example: a human would collect images with and without cats, label those images as such, and then train a model to classify the images as having cats or not. The trained model would then be able to classify new images as having cats or not.

Unsupervised learning (UL) is the task of learning from unlabeled data. Even though data no longer needs labeling, the methods used by the computer to gather data still need to be designed by a human. The goal in UL is to group data into

meaningful clusters. For example: a human would collect data on customers and then train a model to group those customers so that, by known what some of those customers buy, we can offer products to other customers based on common customer traits. We would not tell the machine learning model what relevant traits to use for grouping; this is what the model would learn.

Reinforcement learning (RL) is the task of learning through interaction. In this type of task, no human labels data and no human collects or designs the collection of data. These machine learning algorithms can be thought of as agents because of the need for interaction. The agents need to learn to perform a specific task, like in other machine learning paradigms. They also need to collect the most relevant data. Very often, in RL, you must provide a reward signal. This signal is fundamentally different from the labels in supervised learning. In RL, agents receive reward signals for achieving a goal and not for specific agent behaviors. Additionally, this signal is related to an obviously desired state like winning a game, reaching an objective or location, and so on, which means that humans do not need to intervene by labeling millions of samples.

Main branches of machine learning



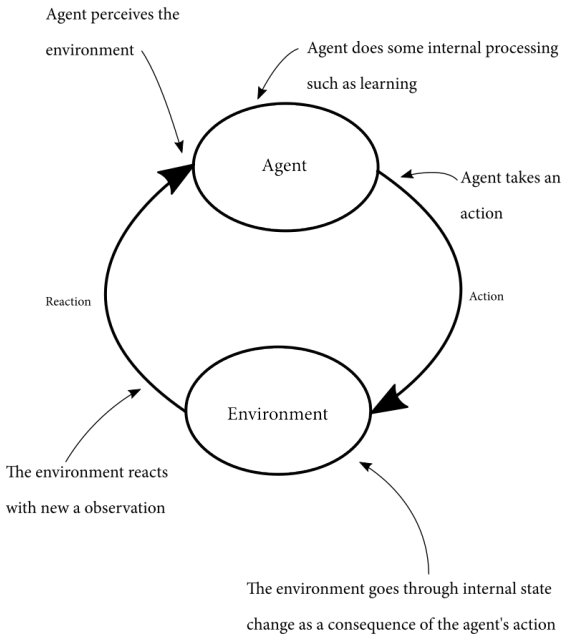
These types of machine learning tasks are all important, and they are not mutually exclusive. In fact, the best examples of artificial general intelligence are those that combine different machine learning techniques.

Reinforcement Learning

The fields of mathematics, engineering, psychology, economics, neuroscience, computer science are all interested in the problem of optimal action selection—how to come up with action policies for complex systems? Reinforcement learning is the manifestation of this need. RL can be thought of as computational behaviorism. Behaviorism is a psychological approach that studies human and animal behavior and states that interaction with the environment is the source of all behavior. The environment tries to reinforce or discourage certain behavior, and it is ultimately the human or animal that decides what action is best under specific circumstances. There is no formal connection between RL and behaviorism, but human and animal behavior can help us gain an intuitive understanding of reinforcement learning.

Think of how you would train a dog to sit. When the dog properly sits on command, you give him a treat. You know your dog likes treats, and he would probably want to know how to get more of them. He interacts with you by trying different actions: he might run, bark, stay, and many other things before sitting. Imagine how complicated it is for your dog to learn what action led to the treat. Was it the running at first? Was it giving up? Your dog needs to look back in time and assign credit to possible actions that led to the treat. It also needs to explore. Not only that, but it needs to learn to balance exploring new actions and retrying the actions that have worked best so far.

The reinforcement learning interaction cycle

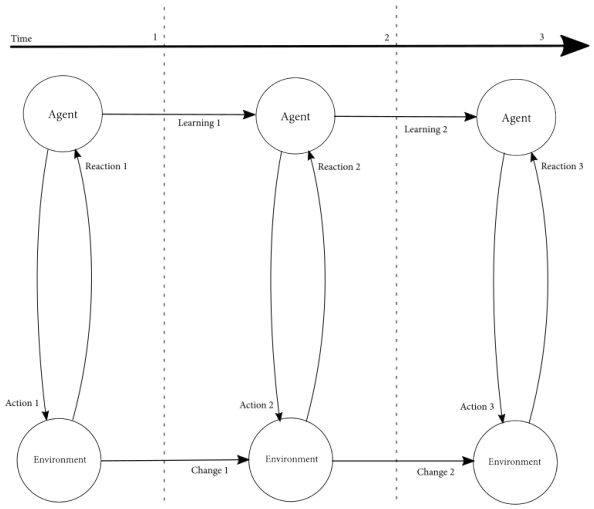


Reinforcement learning is an interaction cycle between a controller, known as the agent, and a system, known as the environment. In the dog training example, your dog is an agent and you are your dog's environment. The cycle begins with the agent observing the environment. The agent does some internal

processing of the observation like learning or memorizing. The agent then takes an action that will affect the environment in some way. Usually, the action affects the environment in two different ways: First, there is an internal change in the environment that the agent may or may not be able to see. This internal change could have delayed consequences that only manifest in future interactions. Second, the environment reacts externally, in a way that the agent can see. The environment's response could be a direct consequence of the last agent's action, but it could also be related to some earlier interaction. The whole cycle—observation, processing, action, responses from the environment—then repeats.

RL has three distinct characteristics worth highlighting. First, RL is concerned with *sequential decision-making*. In cases in which there are sequences of decisions, as opposed to single-shot decisions, your decisions may not affect only the immediate feedback signal you get, but may influence all possible future feedback signals. Being able to affect the environment complicates decision-making because we can't simply choose the action with the highest immediate reward for the current system state; instead, our decision-making must also account for the change to the environment that we will bring about.

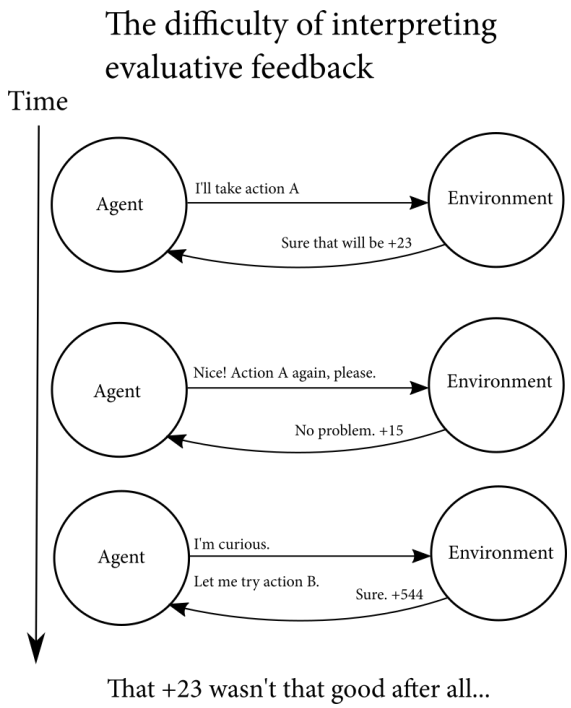
A sequential decision-making problem



The environment's reaction is an immediate feedback and might seem good at first, but the change in the environment will influence all future reactions the agent is able to perceive.

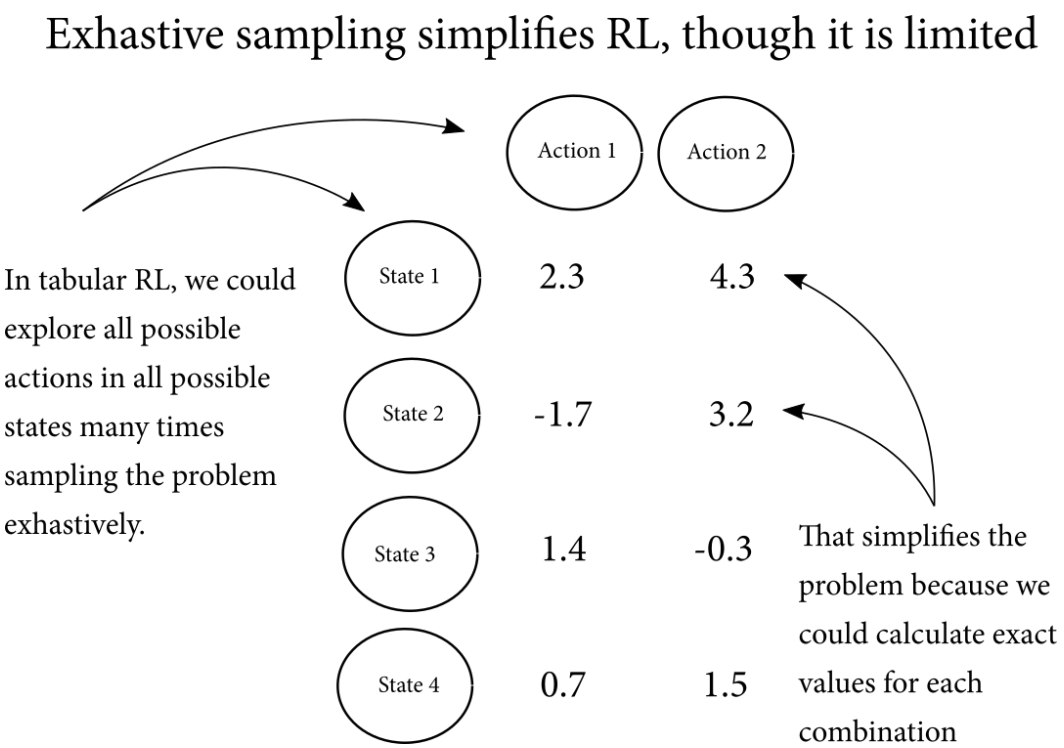
For example: deforestation is one of many sequential decision-making problems. Even though logging operations provide us with wood and paper products that we dearly need, removing trees also affects the environment in ways that could be harmful, even to ourselves. So, we cannot simply think of the action that maximizes our immediate profit, but we must think of how we will affect the environment and how that will, in turn, affect us in the future. In other machine learning paradigms like SL, this sequential nature does not exist. The problems are one-shot or single-shot problems. For example, identifying a cat in an image will not change what you will see in a next image.

The second distinct characteristic of RL is the use of *evaluative feedback*, rather than supervised. Supervised feedback, as is used in SL, is simple and straightforward. You either classified that image correctly or you didn't. Evaluative feedback, on the other hand, is more complex to interpret and categorize as good or bad, correct or incorrect. For example, if you get offered a salary of $60k, is that good or bad? What if you learn one of your future coworkers makes $75k? How good is the offer now? What if all your other future coworkers make $30k? Not so bad anymore, right?



See why evaluative feedback is so complicated? You must first gather data and then make comparisons. But there could always be some data that you haven't seen yet that would change the meaning of your current data.

RL third distinct characteristic is that it relies on an *exhaustive sampling* of the environment. In "tabular" reinforcement learning, you typically fit all possible environment state and action combinations on a table, then simulate millions of interactions with the environment, therefore exposing all possible situations you could ever encounter in that environment. Supervised learning, on the other hand uses sampled feedback. That is, you do not see all possible situations. Instead, the model is trained to generalize to unseen possibilities.

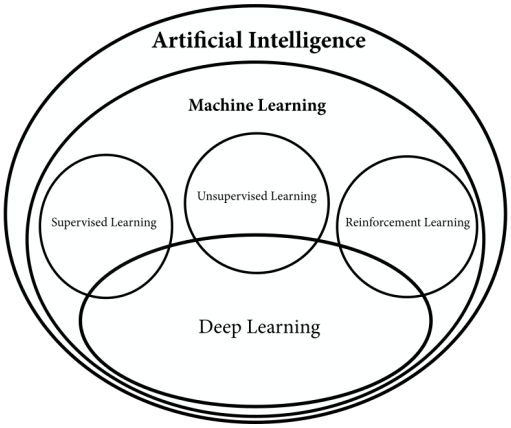


Most of the theory of reinforcement learning exists only for exhaustive sampling cases, but the union of reinforcement learning and supervised learning allows us to build reinforcement learning agents capable of solving complex problems in which exhaustive sampling is inefficient or even impossible.

Deep Learning

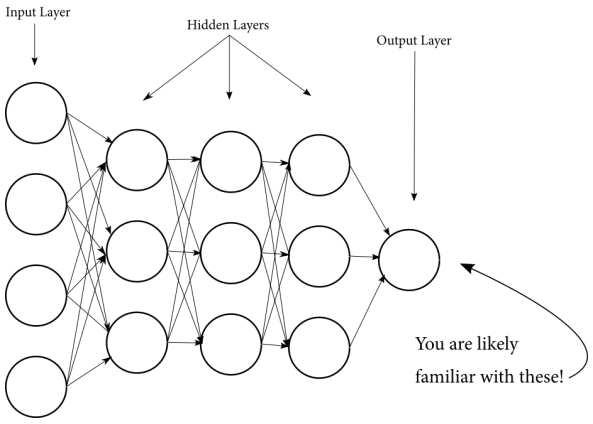
You may think that deep learning is this *new* hot technology that is allowing us to talk about self-driving cars and virtual reality. This is partially true, but deep learning is an approach to machine learning that refers to the training of expressive multi-layer models, most often artificial neural networks. And artificial neural networks (NN) have been around for decades. What makes deep learning special is that, in recent years, we have been able to go from a hard limit of three layers to networks with a double digit number of layers.

Deep learning improving all areas of Machine Learning



In **deep learning** (DL), each layer learns an abstraction of the next layer. For example, if you are training a convolutional neural network (CCN) on images of animals, the ouput layer would learn to classify the animals and each preceeding layer would learn to abstract the images into more general features. To think about it backward is not the most intuitive way, but it is important to know why I describe it this way.

A simple feedforward neural network

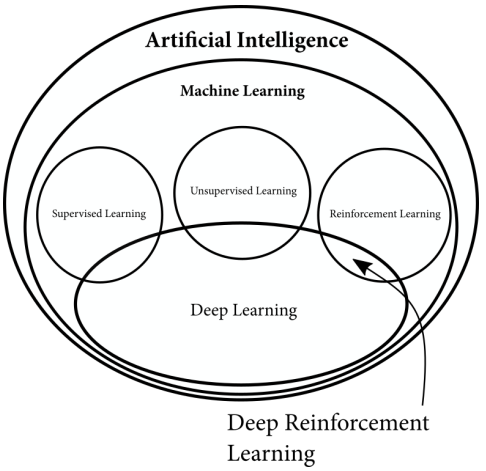


The most popular algorithm used to trained NN is **backpropagation** and it works by propagating the partial computation of the gradient from one layer backwards to compute the gradient of the previous layer. It works from output to input. However, it is arguably more intuitive to think of NN the other way around, from the input to the output layers. This other way, the first layer would learn primitive geometric features such as points, lines and curves. The next layer would learn basic geometric figures such as circles, squares and other shapes. The next layer would learn features more specific to the animals that it trained on such as ears, noses, eyes and so on. The final layer at the top would learn to classify the different animals. Because of the 3-layer limit we previously had, the number of abstractions that we could perform was not that great. Instead, experts had to create relevant abstractions by hand, so they could then train a simple neural network model on these feature vectors.

However, in 2006 a group of Canadian researchers introduced algorithms capable of breaking this limit and applied these new techniques to recognition tasks—first, handwritten digits and later, speech. The results they achieved shocked the machine learning community and revolutionized the field of AI. Every challenge that had previously been attempted using hand-crafted features for training became a great opportunity for researchers to go back and obtain groundbreaking results just by switching to DL. As a result, the deep learning community has been influencing every branch of ML and affecting industry after industry and creating the current state of research.

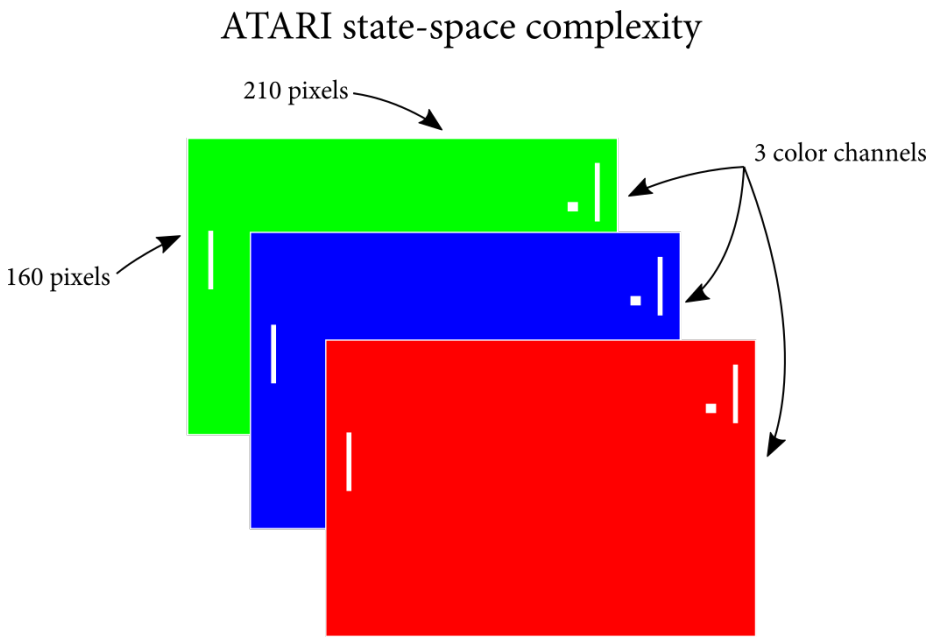
Deep Reinforcement Learning

So how is deep reinforcement learning different than reinforcement learning? As you might have guessed correctly, it's the deep part. **Deep reinforcement Learning** (DRL) is simply the use of multiple layers of powerful function approximators to solve complex sequential decision-making problems.



Imagine you want to train an agent to play ATARI games straight from the images the console outputs. ATARI games

output high dimensional data; images of 210 by 160 pixels with 3-channel video at 60 times per second. Still, this representation does not cover motion. For that, we have to use the last couple of frames to determine things like velocity of the ball and so on.

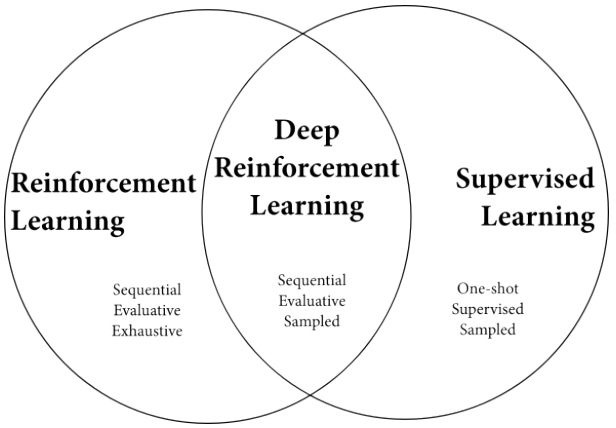


That is a very large state-space. If you compare it with the grid-world games RL is commonly associated with; you quickly sense the gap. Moreover, robotic control problems have continuous state spaces, and games like Go have more states than the number of atoms in the universe. So surely, even if you could engineer a way to store a table with all the values you could encounter in these environments, it would be very inefficient to learn this way. Instead, we should leverage generalization techniques to find approximate solutions. Given that it is very difficult, sometimes even impossible, to sample the state-space exhaustively, we need to turn towards the function approximation methods found in other areas of machine learning.

One way to create a DRL agent that could learn to play ATARI, then, is by creating a deep learning model, a convolutional neural network, to map a set of 4 consecutive images of the video stream to each of the possible actions in the ATARI game platform. The output nodes correspond to the predicted values of the individual

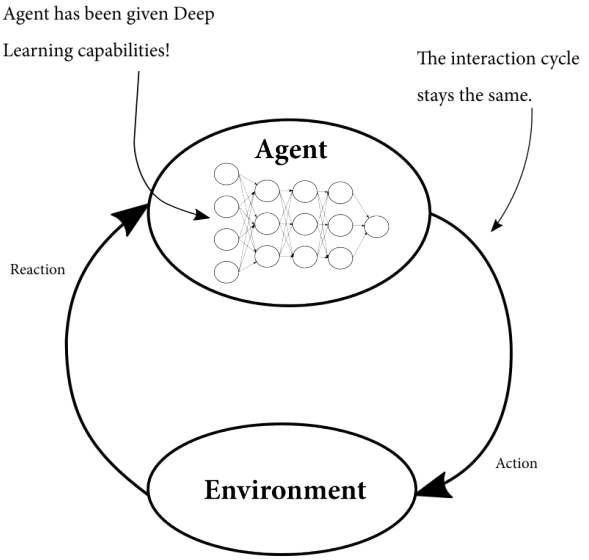
actions for the given state. At first, the agent selects random moves, but after several episodes, the agent learns to perform. I will cover how to solve this problem in more depth later chapters. For now, this should give you an idea how DRL works.

In the previous section, I mentioned how SL was able to deal with sampled feedback. That is, supervised approaches do not have to learn from all possible data, but instead, the task is to learn from a few samples and then generalize to unseen samples. Though not the only way to mix deep learning and reinforcement learning, the most popular use of deep reinforcement learning (DRL), is to use deep learning to approximate the state space of large reinforcement learning problems. By doing this, we can solve RL problems that were not possible to solve before.



We can now iterate over the reinforcement learning cycle we described earlier and simply give the agent deep learning capabilities to create the deep reinforcement learning cycle or API. Most of the time, the deep learning layer will be used to improve the agents' perception, but this is not the only way we can use deep learning to improve reinforcement learning agents as you will see through the chapters.

The deep reinforcement learning interaction cycle

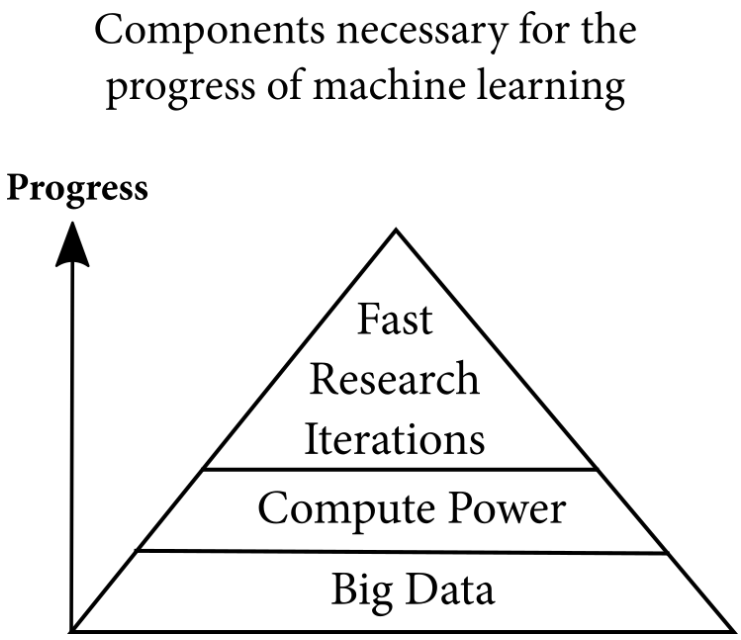


Why now?

You have come to the DRL party just in time. Deep reinforcement learning inherits a lot of history from both the deep learning and reinforcement learning fields. The events that are currently enabling DL research to advance allow DRL research to advance as well. The conditions are ideal for motivated individuals to start contributing, and current developments make the future of DRL bright.

The Learning Pyramid

The progress of machine learning research depends on three major conditions. First, it is vital to have *large amounts of quality data* to train ML models. Second, it is important to have *lots of compute power* available, because the process of training any ML model—especially the neural networks found in DL—is very computing intensive. Third, it is difficult to design and improve ML algorithms that train these models. The *ability to iterate rapidly* when creating a new algorithm has proven beneficial for the progress of the field. There three components is what I call the **learning pyramid**.



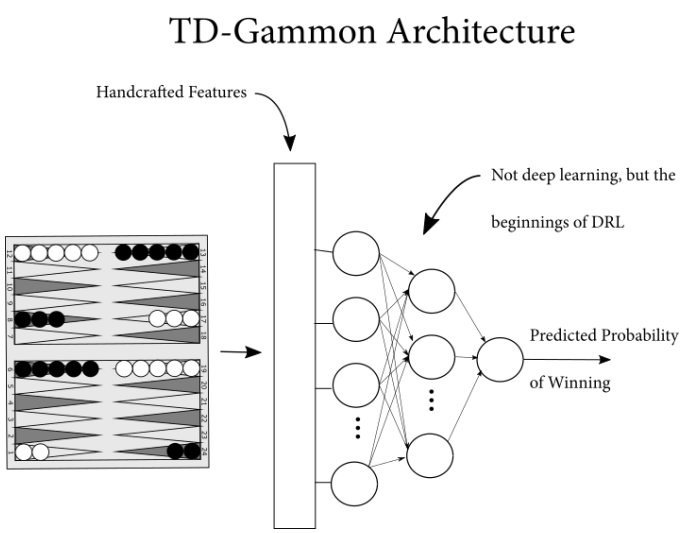
In the 1990's, there was a dramatic improvement to the first component of the learning pyramid with the invention of the internet. Lots of data is the oil that keeps the machine learning engine moving. It is only with lots of data that we can train deep models and make technological progress; do you remember how it was like before search engines? One of Google's strategic advantages over other search engines companies in the late 1990's was the collection and use of data to improve page rankings. Do you remeber AltaVista? Do you use Yahoo? Today, the use of deep learning models trained in large amounts of data is common. Don't believe me? Just ask Siri, Cortana, Bixby, Alexa or Google's Assistant. They might not only be able to tell you examples of DL applications, they are the examples. However, large amounts of data on their own are not sufficient. Training models on large datasets can still take days, weeks, and even months without enough computing power. But we have witnessed an exponential growth in the computing power available to us for decades, and this growth in the availability of computing power, particularly GPUs, that has enabled researchers to train neural network hundreds of times faster than before. In early 2010, graphics processing units (GPUs) were first used to train a neural network. GPUs are processing units like the CPUs regularly found on a computer. The main difference is that while CPUs are good at processing sequential data, GPUs are good at processing parallel data. When doing backpropagation in a deep NN, the calculation of the gradients can be parallelized and the speed up gains when doing so are large. For this reason, the processing of parallel data becomes a priority, and the availability of powerful GPUs a good thing for deep learning research.

Every year we continue to get more and more processing power, from both CPUs and GPUs. There are some that predict that this exponential growth will continue for years to come, possibly surpassing the computing power of a human brain in just a few decades. Some even suggest that we will surpass the processing power of all human brains combined in our lifetime. Think about that for a second!

Data and computing power form the basis for rapid iterative Machine Learning research because they allow researchers to iterate ideas quickly. You come up with an idea for a new algorithm, you code it up and send a few versions to the cloud, and you soon find yourself in the next iteration. Rapid development iterations allow researchers to come up with new and better algorithms much faster than they did before. Lately, we have been getting improvements and groundbreaking results from the deep learning community every month or so, and that by itself is amazing.

Success Stories

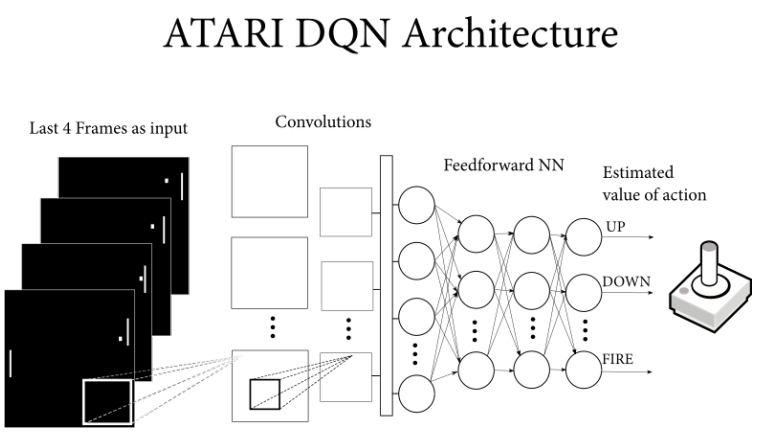
The use of artificial neural networks in reinforcement learning started around the 1990's. One of the well-known early RL successes was Gerald Tesauro's backgammon-playing computer program, called TD-Gammon. TD-Gammon learned to play backgammon by learning to evaluate table positions on its own through reinforcement learning. Back then, however, there was no possibility of training deep neural networks, so the best hand-crafted features were selected and passed to a regular network classifier instead. Even though the techniques implemented are not exactly considered deep reinforcement learning, TD-Gammon was one of the first widely-reported success stories of using NN to solve complex RL problems.



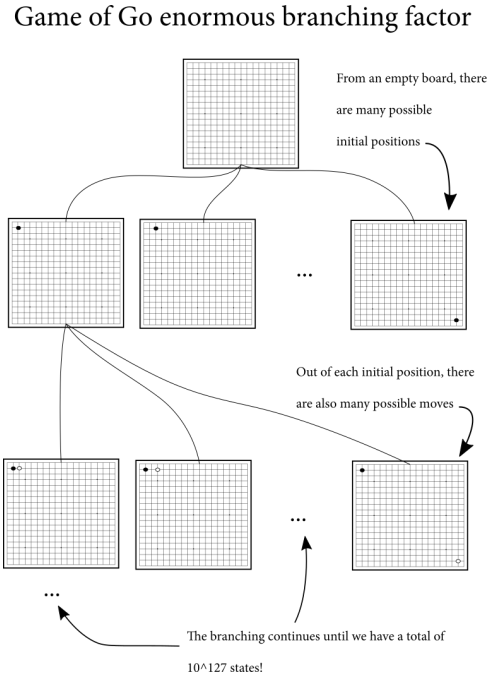
In 2004, Andrew Ng et al. developed an autonomous helicopter that teaches itself to fly stunts by observing hours of flights by experts. They used a technique known as **inverse reinforcement learning** (IRL), in which an agent uses expert demonstrations to derive the expert's goal. The same year, Kohl and Stone used a class of deep reinforcement learning methods known as **policy gradient** (PG) methods to develop a soccer playing robot for the RoboCup tournament. They used RL to teach the agent forward motion. After only three hours of training, the robot achieved the fastest forward moving speed of any other robot of the same hardware.

There were other successes in the 2000's, but the field of DRL really only started growing after the deep learning field took off in 2013, when DeepMind published a paper presenting the DQN algorithm. DeepMind trained an agent to play ATARI games straight from pixel images, using a single convolutional neural network and a single set of hyper-parameters. They trained the agent from scratch in almost 50

games. In 22 of the games, their algorithm reached better performance than a professional human player. This accomplishment started a revolution in the DRL community: Silver et al. released the DPG algorithm in 2014. Then, Lillicrap et al. iterated DPG with DDPG in 2015. In 2016, Schulman et al. released TRPO and GAE algorithms; Sergey Levine, Chelsea Finn, et al. released the GPS algorithm; and Silver et al. shocked the world with AlphaGo, following up with AlphaGo Zero and AlphaZero in 2017.



AlphaGo is a Go computer program that combines supervised learning, deep reinforcement learning, and Search. The program observed millions of professional players and then used reinforcement learning to perfect its game through self-play. It's important to note that the game of Go is one of the more complex board games in the world, reaching a state-space size of 10^{170}—more than the number of atoms in the universe, as I mentioned earlier. AlphaGo learned by bootstrapping on professional players; AlphaGo Zero, learned to play from scratch, only by its own trial-and-error learning, and it still beat AlphaGo 100 games to 0. AlphaZero,

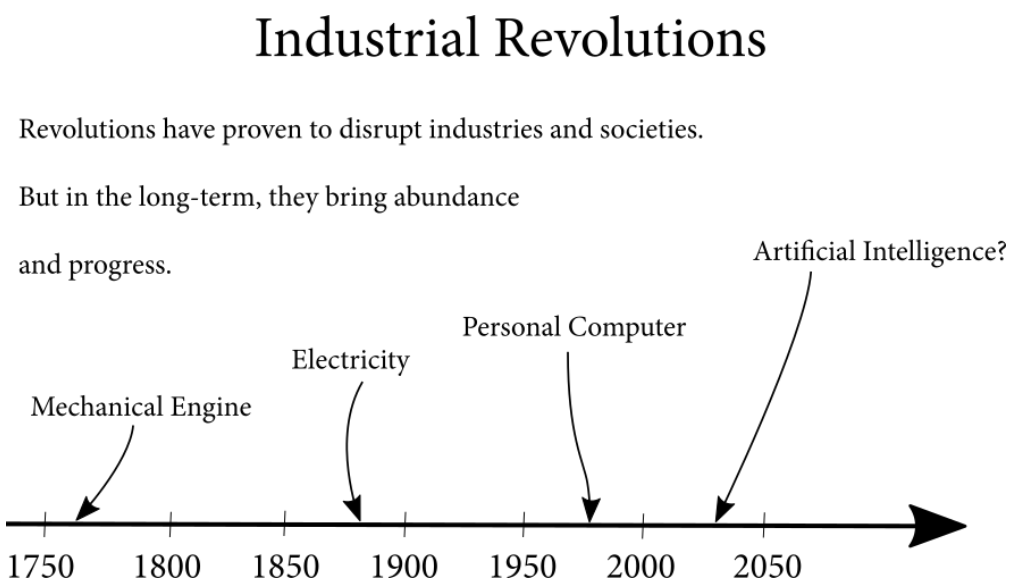


one of the biggest steps forward in deep reinforcement learning history, learned to play better than AlphaGo Zero, beating it 60 games to 40, and was trained in a shorter amount of time than all previous algorithms.

Thanks to deep reinforcement learning we've gone from training agents to play TD-Gammon, with its 10^{20} states, to training agents to play Go, with its 10^{170}, in just two decades. Imagine what the next two decades will bring us. Deep reinforcement learning is a booming field and is expected to revolutionize the artificial intelligence community.

Opportunities Ahead

Artificial intelligence is a field with unbounded potential for positive change regardless of what fearmongers say. Back in the 1750's, there was chaos due to the start of the industrial revolution. Machines were replacing repetitive manual labor and mercilessly displacing humans. Everybody was concerned; how could we become better than these machines and get our jobs back? What are we going to do now? The fact is these jobs were not only unfulfilling, but many of them were also dangerous. After 100 years of revolution, the long-term effects of these changes were benefitting communities. People that usually owned only a couple of shirts and a pair of pants were

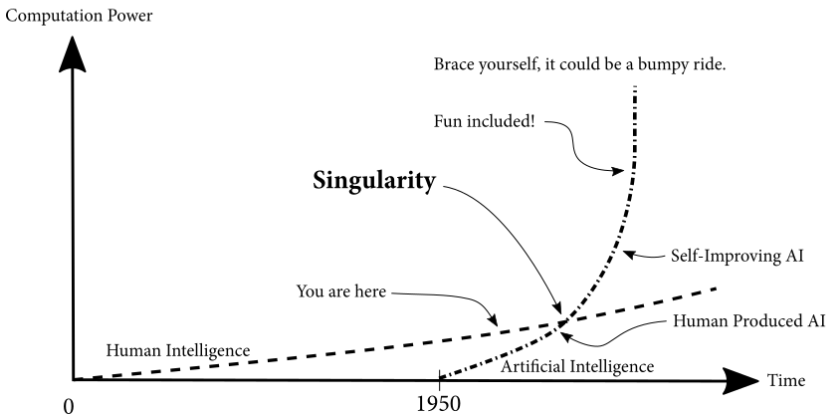


now able to get much more for a fraction of the cost. Surely, it was rough for society at first, but the long-term impact is the more important pursuit.

Artificial intelligence could be considered part of the last revolution called the digital revolution, but some proponents think AI is a revolution of its own. The digital revolution started in the 1970's with the introduction of the personal computers. Then the internet changed the way we do things. Because of the internet, we got big data and cloud computing. Machine learning used this fertile ground to sprout into what it is today. In the next couple of decades, the changes and impact to society will be difficult to accept at first, thus fearmongers. Sure, lots of people may lose their jobs as multiple industries change. Self-driving cars, autonomous drones, smart cities, speech recognition, personal assistants, virtual reality, just to name a few. As you can see, the long-lasting effect will be far superior to any setback along the way. I expect in some decades humans will no longer need to work for food, clothing or shelter as these things will be produced by AI continuously making our society thrive with abundance. How many people out there hate their jobs? Well, being freed from unsatisfying jobs should be one of the top humanity's goals for the century.

As we push the intelligence of machine to levels superior to that of ours, we will have access to creating intelligent machines we weren't capable before. This principle is called singularity; machines that are more intelligent than humans would allow for the improvement of intelligent machines at a faster pace, given that the self-improvement cycle would get rid of the bottleneck, namely, humans.

Singularity could be just a few decades away



But it is too soon to start dwelling on this. Singularity is part of the future and nobody knows with certainty when it will happen or even if it will happen. Thus, it is not worth spending too much time on it. If it can happen, it will but it is hard to tell when. Regardless, the best you and I can do is continue to work to advance artificial intelligence and forget about the rest. There is plenty of work to do, and that's why you are here.

When to use Deep Reinforcement Learning?

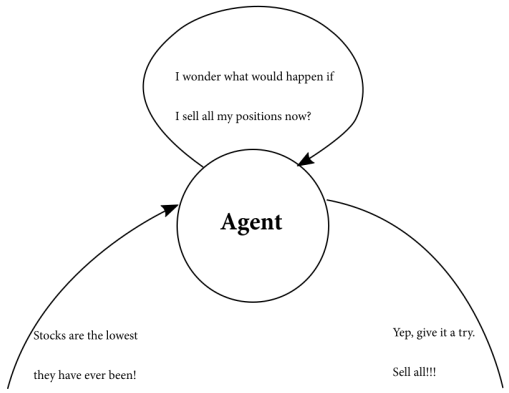
You could formulate any machine learning problem as a deep reinforcement learning problem, but this is not always a good idea, and for multiple reasons. You should know the pros and cons of using DRL in general, and you should be able to identify what deep reinforcement learning is good at and what it is not so good at, from a technological perspective.

What are the pros and cons?

Beyond a technological comparison, I would like you to think about the intuitive advantages and disadvantages of using deep reinforcement learning for your next project. You will see that each of the points highlighted can be either an advantage or a disadvantage depending on what kind of problem you are trying to solve. For example, this field is about letting the machine take control. Are you OK with letting the machine take the decisions for you?

There is a reason that deep reinforcement learning successes are often games: it could be very costly and dangerous to have an agent training directly in the real world. Can you imagine a self-driving car agent learning not to crash by crashing? The machine will have to make mistakes. Are you able to afford that? Are you willing to risk the negative consequences—actual harm—to humans? Whether you are writing agents for a home project or work, these questions should be considered beforehand.

Deep reinforcement learning agents will explore!
Can you afford mistakes?



You will also need to think about what exploration strategy to use. Deep reinforcement learning agents learn by trial-and-error. This means that they will have to select actions for the sake of gaining information. Such actions maybe terrible ideas, but this is how your agent learns. Unfortunately, the most commonly used exploration strategy today is called epsilon-greedy. It means that with an epsilon probability, a very small number, the agent will try a random action. The consequences could be disastrous in the real world. Give it a thought, what would happen with a stock trading agent that randomly explores the set of all possible actions? So, before you begin your quest to develop a trading bot, make sure you think about better ways for your agent to explore.

Finally, training from scratch every time can be daunting, time-consuming and resource intensive. However, there are a couple of areas that study how to bootstrap previously acquired knowledge. First, we have **transfer learning** which is about transferring knowledge gained in different tasks to new ones. For example, if you want to teach a robot to use a hammer and a screwdriver, you could reuse low-level actions learned on the "pick up the hammer" task and apply this knowledge to start learning the "pick up the screwdriver" task from a more advanced stage. This should make intuitive sense to you as humans don't have to relearn low-level motions each time they learn a new task. Humans seem to form hierarchies of actions as we learn. The field of **hierarchical reinforcement learning** tries to replicate this in deep reinforcement learning agents.

The second area that leverages previously acquired knowledge has to do with transferring knowledge learned from simulation into the real world. For example, training on a simulated robot and then running the learned policy on a real-world robot. For some problems, this is not a straightforward transfer, and there is some fine-tuning to be done. Keep an eye out for these topics in later chapters and think about leveraging these techniques in your projects.

What are Deep Reinforcement Learning's Strengths?

Deep reinforcement learning is about mastering explicit tasks. Unlike supervised learning, in which generalization is the goal (an ImageNet classifier can distinguish between 1,000 classes, and an RL agent can often only do one thing well even if that one thing is "playing multiple ATARI games"), reinforcement learning is better at concrete, well-specified tasks. If you can define well what the task is and use that as

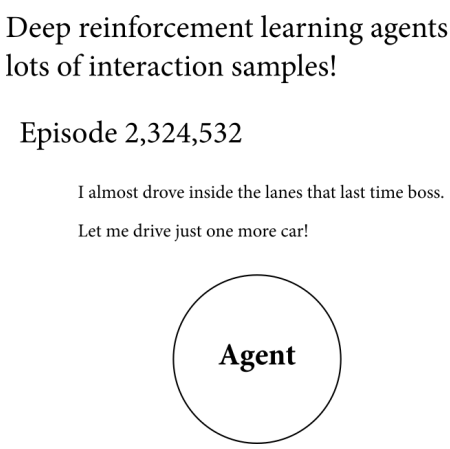
your reward function, you are more than halfway there. ATARI games, for example, have a very explicit goal. You even have the reward function that the agent should maximize shown explicitly on the screen. But sometimes the reward function is not clear. Deep reinforcement learning has difficulties when this is the case.

In deep reinforcement learning, we use generalization techniques to learn simple skills directly from raw sensory input. The performance of generalization techniques is one of the main improvements we've seen in recent years thanks to deep learning. These improvements have made deep reinforcement learning achieve stunning results in complex environments. If deep learning is good at some task, say image classification, then deep reinforcement learning problems that could leverage that will be partly solved.

Finally, another thing deep reinforcement learning has been good at for many years is learning from human demonstrations, also known as **apprenticeship learning** and **inverse reinforcement learning**. I talked briefly about the Stanford Helicopter that learned from watching the remote control moves by an expert, and we will touch on it in more detail later in the book.

What are Deep Reinforcement Learning's Weaknesses?

Of course, deep reinforcement learning is not perfect. One the biggest issues you will find is that agents need millions of samples to learn interesting policies. Humans, on the other hand, can learn from very few interactions. Sample efficiency is probably one of the top areas of deep reinforcement learning that could use some improvements. We will touch on this topic in several chapters as it is a very important topic in the field of DRL.



Another issue with deep reinforcement learning is understanding the meaning of rewards. If a human expert will be defining the rewards the agent is trying to maximize, does that mean that we are somewhat "supervising" this

agent? And more importantly, is this something we want to give up? We, as humans, don't have explicitly defined rewards. Often, the same person can see an event as positive or negative with only changing his or her perception of reality. What is a good reward? What is a bad reward? Additionally, rewards for a task such as walking are not very straightforward. Is it the forward motion that we are targeting, or is it not falling? There is ongoing research on reward signals. One very interesting research is called **intrinsic motivation**. Intrinsic motivation allows the agent to explore new actions just for the sake of it, out of curiosity. Agents that used intrinsic motivation showed an improved learning performance in environments with sparse rewards. I would expect more research to come out in this area in the future.

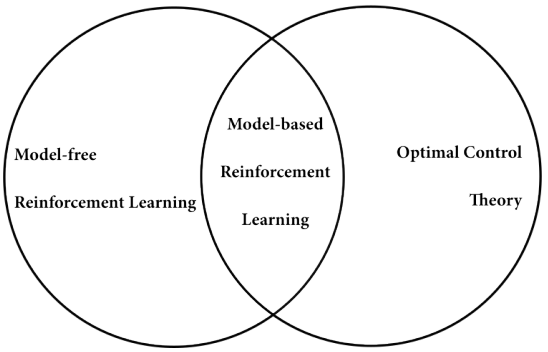
What will you learn in this book?

My goal is to take you, a machine learning enthusiast, from basic reinforcement learning techniques to state-of-the-art deep reinforcement learning. Therefore, we will focus on the reinforcement learning problem. You will first be shown the fundamental concepts in reinforcement learning. Then you'll build on those toward deep reinforcement learning techniques. In the last part of the book, I will walk you through the process of making deep reinforcement learning agents as general purpose as possible by touching on advanced topics related to artificial general intelligence.

Sequential Decision-Making

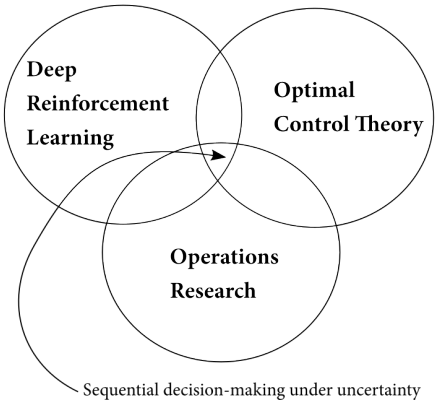
The reinforcement learning problem is about sequential decision-making under uncertainty. Sequential decision-making is a topic of interest to many fields, but each field looks at the problem through different lenses. Control theory studies ways to control complex known dynamical systems. Usually, we know mathematically how these systems behave before the interactions begin. Therefore, the methods and algorithms for solving control

Optimal Control Theory and Reinforcement Learning influence Model-Based Reinforcement Learning



theory problems make slightly different assumptions than deep reinforcement learning methods. However, the techniques used in control theory are often also used in deep reinforcement learning, and the other way around. Operations research also studies decision-making under uncertainty, but problems in this field often have much larger actions spaces than those commonly seen in deep reinforcement learning. Psychology studies human behavior, which is partly the same "sequential decision-making under uncertainty" problem. In fact, you can find research papers linking the different fields and proposing new theories of learning that all fields studying sequential decision-making are able to leverage. The bottom line is that you have come to a field that is influenced by a variety of others. Although this is a good thing, it also brings some inconsistencies in terminologies, notations and so on. I will use computer science notation and will let you know of different names you will find in the literature.

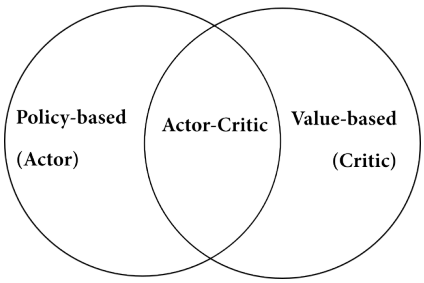
Many fields of science study decision-making. Probably the most important ones are

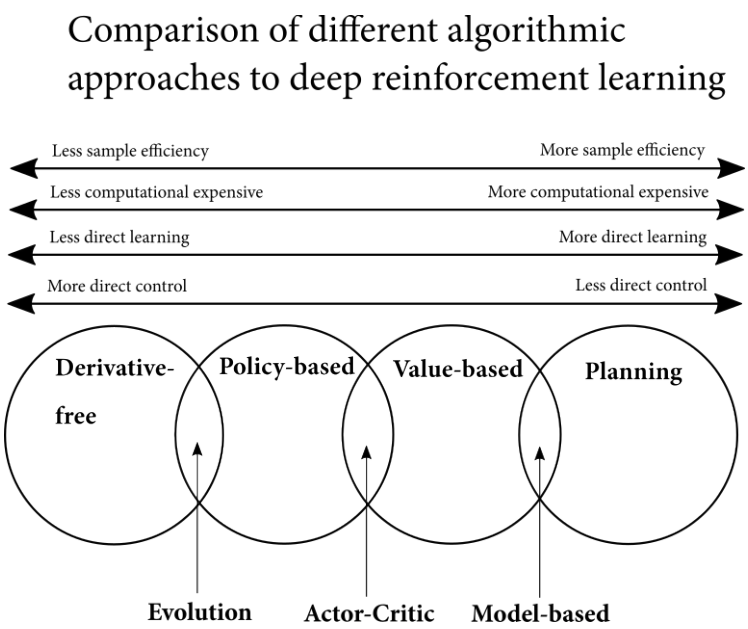


Algorithmic Approaches

Given all the different science fields studying sequential decision-making, the different approaches to solving reinforcement learning problems are many. In this book, you will learn most of them—planning methods, model-free prediction, model-free control, model-based reinforcement learning, and so on. Once we get to the "deep" reinforcement learning concepts, you will be presented with algorithms that are both value-based and policy-based, including some evolutionary approaches that are often not considered reinforcement learning techniques, despite solving reinforcement learning problems well. Then,

Hybrid policy and value based are often the more robust agents





you will be presented with the hybrid, actor-critic methods, followed by multi-agent reinforcement learning and concepts on artificial general intelligence, such as architectures, and safety.

I will take the hands-on approach to teaching these topics, starting with a very basic reinforcement learning environment that I will use to teach you different foundational concepts. Every chapter after that will start with a slightly more complex reinforcement learning problem, and I will present different ways to solve each one of these problems, introducing you to concepts, definitions, algorithm, code and yes, some math as well, as we progress.

Advanced Topics

In the last part of the book, I will address advanced deep reinforcement learning methods. First, I will go over advanced exploration strategies. The exploration vs. exploitation trade-off is unique and defining to reinforcement learning, so it is important we spend some time on it. After putting in the time, the realization of effects that different exploration strategies have on agent performance will come to manifestation.

Other advanced methods that you will discover in this book are hierarchical ap-

proaches, inverse reinforcement learning, and transfer learning. The goal of reinforcement learning is to solve intelligence in general, and this will become apparent as we advance. Therefore, you will uncover how some algorithms studied under the deep reinforcement learning umbrella also fall under the broader artificial general intelligence field. It's going to be fun.

What do you need?

Deep reinforcement learning is at the edge of artificial intelligence research, as such, you will need to bring a couple of things to the table.

Previous Knowledge

I presume you know what deep learning is beyond the refresher you just got in this chapter. Have you trained a convolutional neural network? Then great, if you haven't, please take some extra time to train a couple of networks. Trust me; you will get a lot more from this book if you do. However, you don't have to be a deep learning expert by any means. I'm going to teach deep reinforcement learning, and it is the reinforcement learning problems, concepts, and algorithms that sit at the core of this book. The deep learning networks used by deep reinforcement learning algorithms are often not nearly as complex as you see on the standalone deep learning field. I will show the deep learning architectures and comment briefly on them. However, you won't find detailed explanations as to why a network was used instead of a slightly different one. Most of this book will be on the application of these networks to deep reinforcement learning problems.

Ideally, you also have a basic machine learning knowledge. Are you familiar with supervised learning? Deep reinforcement learning at its core is the mix of reinforcement learning and supervised learning. It is important that you have a basic understanding of supervised learning techniques. The first chapter of the second part will give a refresher on supervised learning methods as related to RL, and this should be sufficient so long you already understand the basics.

Finally, as this will be a very hands-on take, make sure you feel comfortable with installing software, packages, and firing up your text editor of choice to code away problems. I will give you full implementations, but I hope that you type things along and experiment with them. It is amid the battle that heroes are born. My goal is to

unleash the best of you, so crack your knuckles, drink some coffee, and get ready.

Environment Setup

There are different challenges to consider when setting up and environment for DRL experimentation. One of them is the use of processing power. As expected, like deep learning techniques, GPUs are important to deep reinforcement learning. However, unlike deep learning, which does not use too much of the CPU, some deep reinforcement learning algorithms are asynchronous implementations that heavily rely on CPU processing power and most importantly thread count.

Another challenge is the need for graphical environments. You are probably familiar with using the cloud, headless servers, or Jupyter notebooks for training deep learning models. However, deep reinforcement learning methods have lots of interactions that can give you useful information just by looking at them. Though, it is still possible to send your code to the cloud, headless servers, or Jupyter notebooks, it is also an important debugging step to be able to see what your agents do during training. Have that in mind for when you start coding up your first couple of agents.

Finally, everyone has different needs, and it is difficult to provide a single environment that will satisfy all of the readers. Still, I will be providing a base environment setup for deep reinforcement learning development. However, feel free to customize your environment to better suit your needs. The provided environment is not designed to be better than any you could customize yourself. Invest the time, and you'll have fun galore.

A will to Learn (And be part of history)

I'm not trying to sound corny here; DRL gives you the opportunity to contribute to the world and be a part of history. To be part of history you don't necessarily need to be the best DRL researcher in the world, perhaps not even a "researcher", but you do need to be the best you. The motivation to learn—to read one more chapter—is all you'll need to get you through the difficult parts of this book. Learning is no different than working out; you may find challenges to get started, you may find it difficult to keep going, but the relentless drive to get through one more rep will bring the gains you desire. So, when you face one of those difficult lines, think about the long-term and push through.

Summary

By now, you should know a little bit of the history of artificial intelligence and that its goal is something that humans have pursued for many years. You should know that machine learning is one of the most popular and successful approaches to artificial intelligence. You should know that reinforcement learning is a one of the three branches of machine learning along with supervised learning and unsupervised learning. You should know that deep learning is not tied to any specific machine learning branch, but its power, has instead helped the entire machine learning community, including reinforcement learning, advance. You should know that deep reinforcement learning is simply the use of multiple layers of powerful function approximators known as neural networks to solve complex sequential decision-making problems. You should know that deep reinforcement learning has performed well in many control problems, but nevertheless, you should also have in mind that releasing human control for important decision making should not be taken lightly. You should know that some of the main issues in this field are the need for samples and the need for exploration. You should also have an idea of what is coming, the dangers of this technology, but more importantly you should be able to see the potential in this field and feel excited and compelled to bring your best and to embark in this journey. The opportunity of influence a change of this magnitude happen only every few generations. We should be glad it's us living these times. Let's be part of it.

More concretely, by now you:

- Understand what deep reinforcement learning is and how it began.
- Know how a larger field of related approaches share interests and concepts with deep reinforcement learning and how these relationships influence the field.
- Recognize why and how deep reinforcement learning is important and different than other approaches to machine learning.
- Can identify what deep reinforcement learning can do for different kinds of problems.

Planning For Sequential Decision-Making Problems 2

IN THIS CHAPTER

You'll represent sequential decision-making problems when actions have uncertain consequences.

You'll identify the objective of reinforcement learning problems and the different components that support common reinforcement learning algorithms.

You'll solve problems in which classical AI planning is not sufficient: when our actions no longer have deterministic effects, but instead, have stochastic effects.

You'll solve sequential decision-making problems by creating agents that find optimal policies of behavior in known environments.

"In preparing for battle I have always found that plans are useless, but planning is indispensable."

— Dwight D. Eisenhower
United States Army five-star general and
34th President of the United States

You pick up this book and decide to read one more chapter despite having limited free time. A coach sits their best player for tonight's match, ignoring the press criticism. A parent invests long hours of hard work and unlimited patience to show their child good manners. These are all examples of sequential decision making in which we must balance immediate and long-term goals.

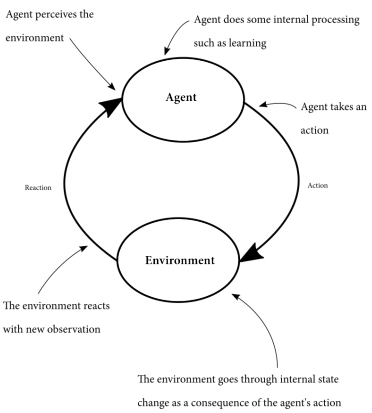
Our actions have uncertain consequences; we cannot know what will happen because of them. Our actions also have delayed consequences; it can be difficult to relate individual actions to moments of satisfaction in life. The conjunction of these two can make it difficult to correlate our behavior with our ability to achieve our goals and can be detrimental to our ability to obtain them.

In this chapter, you'll explore the first tradeoff a reinforcement learning agent faces when selecting actions: agents must balance immediate and long-term goals. I'll present a common mathematical framework used for representing sequential decision-making problems. This framework is central to reinforcement learning and the solutions we will explore in later chapters, so make sure to spend time reading and understanding the "*Show Me the Math*" boxes. They are pretty important! I'll also discuss the objective of a decision maker and the supporting components that will help a decision maker become a good one. Finally, I will show some of the most fundamental algorithms for solving sequential decision-making problems—balancing the tradeoff between immediate reward and long-term goals.

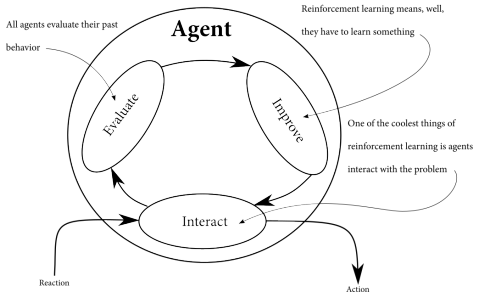
Architecture of a sequential decision-making problem

Let me illustrate again how the reinforcement learning cycle goes. We have a sequential decision-making problem; in reinforcement learning lingo, we call that the **environment**. We also have a decision maker, referred to as the **agent**. The environment is everything the agent has no *total* control over, including parts that may seem to belong to the agent. For example, if you were a decision-making agent (which you are), your hands would be

The reinforcement learning interaction cycle



The three internal steps that every reinforcement learning agent goes through

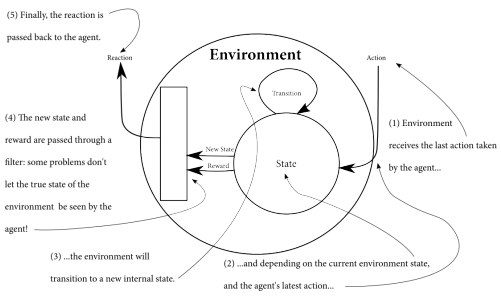


part of the environment and not the decision maker in you. Interesting, right? This strict boundary of agent and environment is counter-intuitive at first, but the reason is that we want the decision maker only to have a single role: *making decisions*. Your hands don't make decisions; thus, they are not part of the decision-making agent. If we were to zoom in, we would see that most agents have a three-step process: all agents interact with the environment, all agents evaluate their behaviors by using the results of them, and all agents improve something in their behavior.

The agent can influence the environment through actions. Actions can have *non-deterministic* consequences in the environment: if you select the same action when the environment is in the same state but in different occasions, the consequences of your action may not be the same. You may opt for studying hard for a final exam, but this does not mean you will get full marks every time.

At any given time, the environment has a defined configuration of variables that make up a *state* relevant to the decision maker. For example, the environment of a stock trading agent builds the environment state with the amount of cash available for trading, stocks prices, the day of the week, the week of the year, and a categorical variable determining the political state of the country, etc. Any given configuration of all those variables would make up a single state. Environments can have multiple states, and this makes decision making even more challenging.

Process the environment goes through as a consequence of agent's actions

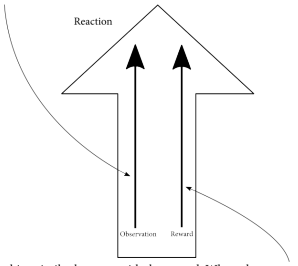


When the agent sends an *action* to the environment, the environment uses its internal state and the action sent by the agent to *transition* to a new state. Remember that these states are internal to the environment; the agent may or may not have access to the exact environment states. In some problems—a game of poker, for example—the environment has variables that are not fully-observable, like

the cards in other players' hands. After the environment transitions to a new state, it also *reacts* to the agent's action. In this reaction, the environment commonly sends back to the agent an *observation* that suggests what may be the environment's internal state.

An inner look at environments' reactions

"Observation" is commonly the full environment internal state for simple RL problems, so you could simply put "State" instead. But this is not always the case! Poker, for example, is a partially observable game. Agents don't have access to the full state of the game, but only see single cards. These are observations.



Something similar happens with the reward. Where does reward come from? As humans, we have somewhat power to change our perception of reward. Remember how rewarding was your first paycheck? Why did its value fade out? Some research is attempting to allow the agent to control the "reward". This is scary, but exciting stuff.

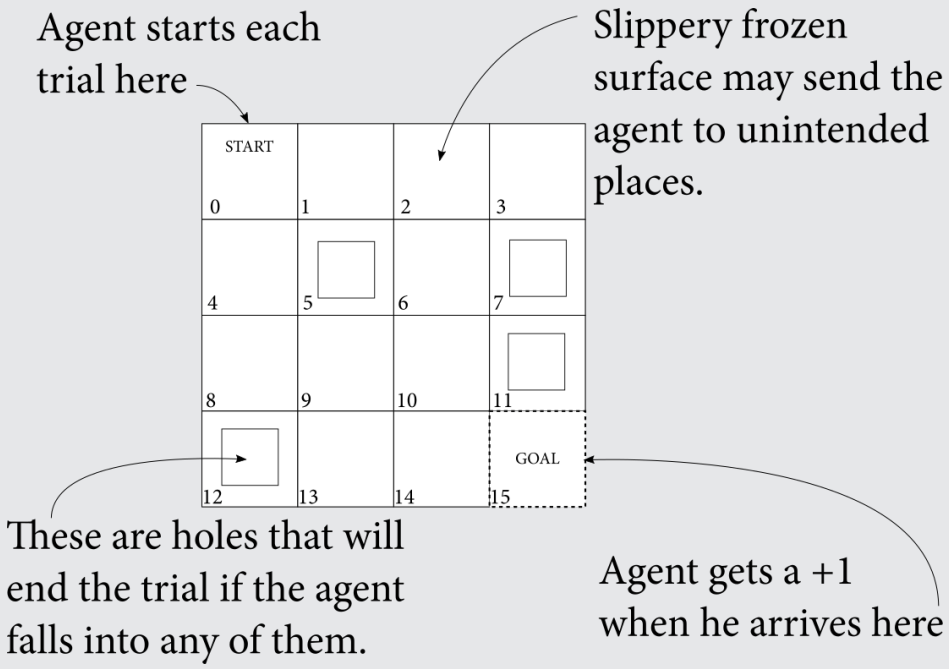
In this first part of the book and most of the second part, we assume the observation doesn't just suggest, but instead shows exactly the environment's new state. In addition to the observation, the environment also sends a scalar value, referred to as a *reward*, evaluating the last action taken by the agent on a state of the environment. Just like the observation, the reward is a component of reinforcement learning problems that receives a lot of attention. There has been recent research studying ways to make the reward depend not just on the environment, but also on something internal to the agent. Agents that do this develop curiosity in some sense and try things out just for the fun of it. Awesome, right? This is a very advanced aspect of reinforcement learning, and we will touch on it in the last part of the book.

For now, assume the environment is always responsible for telling the agent what's good and what's bad. The reward is a scalar value and can be any real number, positive or negative. The agent will use these components: the state the environment was in before the agent took action, the action selected by the agent, the new state, and the reward for learning. The cycle then continuously repeats. The combination of these components—a set of states, a set of actions, the representation of state changes and reward returns as a consequence of agent actions make up a framework known as Markov Decision Processes (MDP) and are commonly used to build sequential decision-making problems.

Let's look at a concrete example— the Frozen Lake environment—to illustrate the components that make up an MDP in more detail. First, let's break down and zoom into each of the components of an MDP. We will build an MDP from a text description of a sequential decision-making problem. Then, we will discuss the objectives of a decision maker. Finally, we will discuss two of the most important algorithms for solving MDPs.

THE CHALLENGE
The Frozen Lake

Imagine you need to create an MDP representation of this simple grid world, called Frozen Lake.

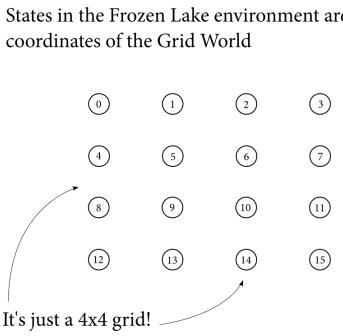


As you can see, this world has 16 positions the agent can be at any given time. The agent will start each trial, also called an episode, at the START cell. Moving into the GOAL location gives the agent a +1 reward, and the current episode will finish. The agent will show up again in the START cell. The surface of this grid world is slippery. The agent will move only with 33.3333% chance as intended, the remaining 66.6666% will be split evenly in orthogonal directions. For example, if the agent chooses to move down, there is a 33.3333% chance it will move down, 33.3333% chance it will move left and 33.3333% chance it will move right. If the agent tries to move out of the grid world, it will just bounce back to the cell from which it tried to move. There are four holes in the world (shown in the diagram). If the agent falls into one of these holes, the episode ends, and the agent goes back to the starting position, and a new trial begins.

Let's build an MDP for this world to illustrate better what's at the core of a reinforcement learning problem. To be able to build MDPs yourself is an important skill to have. However, often you will find reinforcement learning problems for which somebody else has already built MDPs or even reinforcement learning problems for which MDPs are not practical because of the complexity of the problem; reinforcement learning agents do not need to know the precise MDP of a problem to learn solid behavior. Regardless, MDPs (and derivatives) are often assumed to exist under the hood and knowing what's in there will make you more effective.

System states

In the Frozen Lake environment, it seems obvious that there are 16 total states. We can number them left to right, top to bottom, zero to fifteen.



It is a common practice, however, to create a single final state on an MDP to which all "terminal states" transition.

A **state** is a unique and self-contained configuration of the problem. We define the set of possible system states as S where the size of the set is **discrete** rather than continuous, and in the Frozen Lake example, states are also **finite** rather than infinite, that is $|S| = n$. As mentioned before, in the case of MDPs, the states are **fully-observable**. We can see the true state of the environment at each iteration. **Partially-Observable Markov Decision Processes** (POMDPs), which we will discuss in later chapters, are an exception to this rule.

Each environment state is a unique configuration of all necessary variables needed to make states independent of all other states. For example: in the Frozen Lake environment, you only need to know the current state of the agent to predict its next possible states, regardless of all the previous states the agent has been in. You know from state 2 it can only transition to state 1, 3, or 6, and this is true regardless of whether the agent's last state was 1, 3, or 6. The probability of the next state, given the current state and action, is independent of the history of interactions. This memoryless property of Markov Decision Processes is known as the **Markov property**: the probability of moving from one state s to another state s' in two separate occa-

sions, given the same action a, is the same regardless of all previous states or actions encountered before that point.

SHOW ME THE MATH
The Markov Property

(1) The probability of the next state

(3) Will be the same

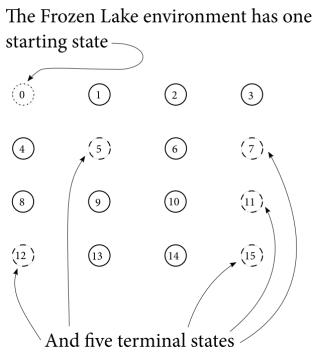
$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots)$$

(2) Given the current state and current action

(4) As if you give it the entire history of interactions

The set of all states in the MDP is $\mathcal{S}^+$. There is a subset of $\mathcal{S}^+$ called the set of **starting** or **initial states**, denoted $\mathcal{S}^i$. To start interacting with an MDP, we draw a state from $\mathcal{S}^i$ from a random probability distribution. There is another state called the **terminal** or **absorbing state**, and the set of all **nonterminal states** is denoted $\mathcal{S}$. A terminal state must have all available actions transitioning, with probability 1, to itself, and these transitions must contain a reward of 0.

In the Frozen Lake environment, there is only one starting state (state 0) and five states that transition to the terminal state. For simplicity, we will illustrate these five states, 5, 7, 11, 12 and 15, as terminal states.

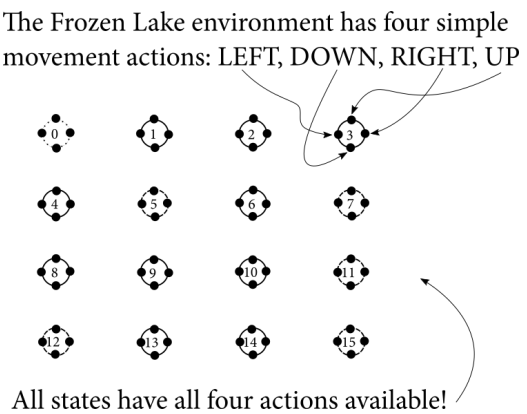


Available actions

In Frozen Lake, there are four available actions at any state: up, down, left, or right. In general, MDPs make available a set of **actions** $\mathcal{A}$, which are also **discrete** and **finite**, of size $|\mathcal{A}| = k$. The set of actions is the system's public API and the way an agent can attempt to push the system into desirable states.

The environment makes the set of all available actions known in advance. There might be some actions that are simply not allowed in a state—in fact, $\mathcal{A}$ is often a function that takes a state as an argument, that is $\mathcal{A}(s)$. This function would return the set of available actions on the given state. For simplicity, you can also define the set to be constant across the state space. And you could always set the transition probability (which I will discuss next) of a given action to zero to denote a **noop** or **no-intervene action**.

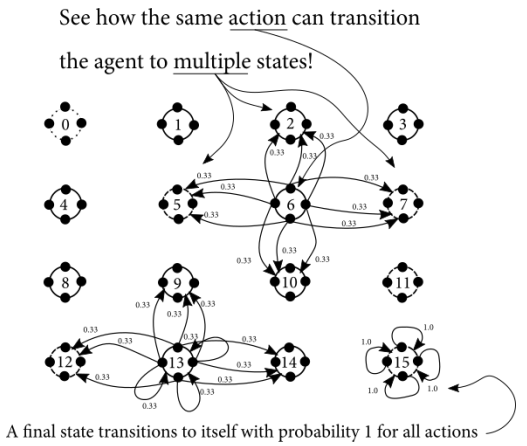
Agents can select actions either **deterministically** or **probabilistically**. This is not the same as saying the environment reacts deterministically or stochastically to agent's actions. Agents can either select actions from a collection of per-state probability distributions or a key-value pair. In the former case, the agent selects the probability distribution related to the state in question and then the actual action from the distribution; then, the action selection is stochastic. In the latter case, however, we agent selects actions deterministically, just like in the Frozen Lake environment; we simply pick an action *a* for a given state *s*.



Consequences of agent actions

In the Frozen Lake environment, we know that there is a 33.3333% chance we will transition to the intended state and a 66.6666% chance we will transition to orthogonal directions. I have added the transition function to states 6, 13, and 15 to illustrate the transition function in the Frozen Lake environment.

We can also visualize the transition function in a table format.



State	Action	Landing State	Probability of Transition
0	right	1	0.333333
0	right	0	0.333333
0	right	4	0.333333
0	down	4	0.333333
...	...	...	...

The way the environment changes is referred to as the **transition function** or **state-transition probabilities**, and denoted $\mathcal{T}(s, a)$. The transition function is often represented as a *per state* and *action* pair probability distributions and is a function of the current environment *state* and the *action* selected by the agent. The transition function $\mathcal{T}$ is a function mapping state and action pairs to states, that is you pass in the state and action pair and it returns a new state. $\mathcal{T}$ describes a probability distribution $p(.|s,a)$ determining how the system will evolve from an interaction cycle. In other words, what will the new state be, given that the agent selected action a in state s? As any probability distribution, the sum of the probabilities must equal one.

SHOW ME THE MATH
Transition Function

(1) The function p which is the probabilities of transitions

(2) Is the probability of landing on state s' on timestep t

(3) Given that while on state s on the previous timestep

$$p(s'|s,a) = P(S_t = s'|S_{t-1} = s, A_{t-1} = a)$$

(4) We select action a

(5) If we sum up the probabilities of landing on all possible next states

(6) The result must be 1

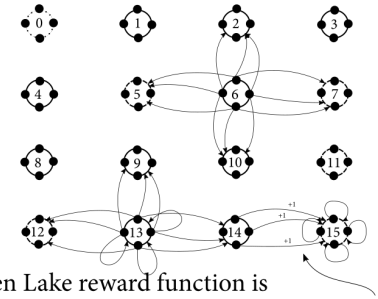
$$\sum_{s' \in S} p(s_{t+1}|s_t, a_t) = 1, \forall s \in S, \forall a \in A(s)$$

(7) For all state and action pairs in the environment

Reinforcement signal

In Frozen Lake, the reward function is simply +1 for landing in state 15, 0 otherwise. I've added the reward function only to transitions that give a reward greater than 0. Selecting the *right* action on state 14 will transition us with 33.33% chance to state 15 and give us +1 reward, but also selecting the *up* and the *down* action from state 14 will transition us with 33.33% probability each to state 15, therefore giving us +1's.

The most explicit way of defining a reward function is using the full transition (s, a, s')



Frozen Lake reward function is simple: +1 for landing in state 15, 0 elsewhere.

The **reward function** $\mathcal{R}$ maps a **scalar** to a transition tuple *s, a, s'*. The reward functions give the numeric signal of the goodness of transitions. When the signal is positive, we can think of the reward as an **income** or a **goal**. Most problems have at least one positive signal—winning a chess match or reaching the desired destination, for example. But, rewards can also be negative, and we can see these as **cost** or **penalty**. In robotics, adding cost is a common practice because we usually want to reach a goal, but within some constraints (often within a particular amount of time).

Show Me The Math

Reward Function

(1) The reward function can take a state and action

(2) And it will return the expectation over the rewards at current timestep

$$r(s,a) = E[R_t | S_{t-1} = s, A_{t-1} = a]$$

(3) Given while on state *s* we took action *a*

(4) But the reward function can also be represented by the state, action, new state triplet

(5) Which is also the expectation over the rewards at the current timestep

$$r(s,a,s') = E[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s']$$

(6) But this time given the previous state and action and the state in which we landed

(7) Notice the reward R_t comes from a set R

$$R_t \in \Re \subset \mathbb{R}$$

(8) Which is a subset of the real numbers

It is important to highlight that, while $\mathcal{R}(s,a,s')$ is more explicit, $\mathcal{R}(s,a,s')$ and $\mathcal{R}(s,a)$ are equivalent functions given that we can simply compute the expectation of rewards $\mathcal{R}(s,a,s')$ to obtain $\mathcal{R}(s,a)$.

Time

We must represent time in MDPs. The **timestep** also referred to as "**epoch**," is a global clock *syncing* all parties and *discretizing* time. Having a clock gives rise to a couple of possible types of tasks. An **episodic** task is a task in which there is a finite number of timesteps, either because the clock stops or because the agent reaches a terminal state. There is also a **continuing** task, which is a task that goes on forever. There is no terminal state, and there are an infinite number of timesteps.

Episodic and continuing tasks can also be defined from the agent's perspective. We call it the **planning horizon**. On the one hand, a **finite horizon** is a planning horizon in which we know the task will terminate in a fixed and finite number of timesteps: if we forced the agent to complete the Frozen Lake environment in fifteen steps, for example. A special case of this kind of planning horizon is called a **greedy horizon**, in which planning horizon equals one.

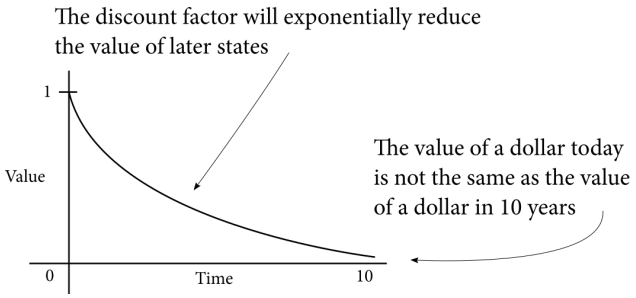
On the other hand, an **infinite horizon** is when the task doesn't have a fixed deadline. Such task may still be episodic and therefore terminate. For example, if an agent is given an infinite number of timesteps to complete a task, we do not know how many steps are left before the episode ends, but we are guaranteed to finish the task in a finite number of timesteps, because the environment has a terminal state. We refer to this type of infinite horizon task as an **indefinite horizon** task. The Frozen Lake environment is this kind of task, the most common type of reinforcement learning task.

We refer to the sequence of consecutive timesteps from the beginning to the end of an episodic task as an **episode**, **trial**, **period** or **stage**. In indefinite planning horizons, an episode is a collection containing all states visited between an initial and a terminal state.

Because of the possibility of infinite sequences of timesteps in infinite horizon tasks, we need a way to discount the value of rewards over time, telling the agent that getting +1's sooner is better. For this reason, we commonly use a positive value less than one to exponentially discount the value of future rewards. We call this a **dis-**

count factor, or **gamma**. The discount factor adjusts the importance of rewards over time. The later we receive rewards, the less attractive they become.

For the Frozen Lake example, we will use a discount factor of 0.9.



SHOW ME THE MATH
Discount Factor (Gamma)

(1) In a finite task, we can define the expected returns

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T$$

(2) As the sum of all rewards starting at the next timestep

(3) All the way to the end of the episode

(4) Though, that would not work if T is infinite; rewards would be infinite!

(5) So, we must discount later rewards by a value less than 1... Gamma

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots$$

(6) These are equivalent

$$\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

(7) This function has an interesting recursive nature which we will exploit later

$$G_t = R_{t+1} + \gamma G_{t+1}$$

(8) See why terminal states must return 0?

Extensions to MDPs

As an aside, I want to mention that there are many extensions to the MDP framework that allow us to target slightly different types of reinforcement learning problems. The following list is not comprehensive, but it should give you an idea of how large the body of research body behind MDPs is. We are currently only looking at the tip of the iceberg.

- Partially-Observable Markov Decision Process (POMDP): When the agent cannot fully observe the environment state.
- Factored Markov Decision Process (FMDP): Allows the representation of the transition and reward function more compactly so that we can represent very large MDPs.
- Continuous [Time|Action|State] Markov Decision Process: When either time, action, state or any combination of them are continuous.
- Relational Markov Decision Process (RMDP): Allows the combination of probabilistic and relational knowledge.
- Semi-Markov Decision Process (SMDP): Allows the inclusion of abstract actions that can take multiple timesteps to complete.
- Multi-Agent Markov Decision Process (MMDP): Allows the inclusion of multiple agents in the same environment.
- Decentralized Markov Decision Process (Dec-MDP): Allows for multiple agents to collaborate and maximize a common reward.

In addition to these, you can combine and create new extensions to MDPs. For example, there are Dec-POMDPs, etc. The research body is large, and I want you to be aware of it. With that out of the way, let's continue building the MDP for our Frozen Lake environment.

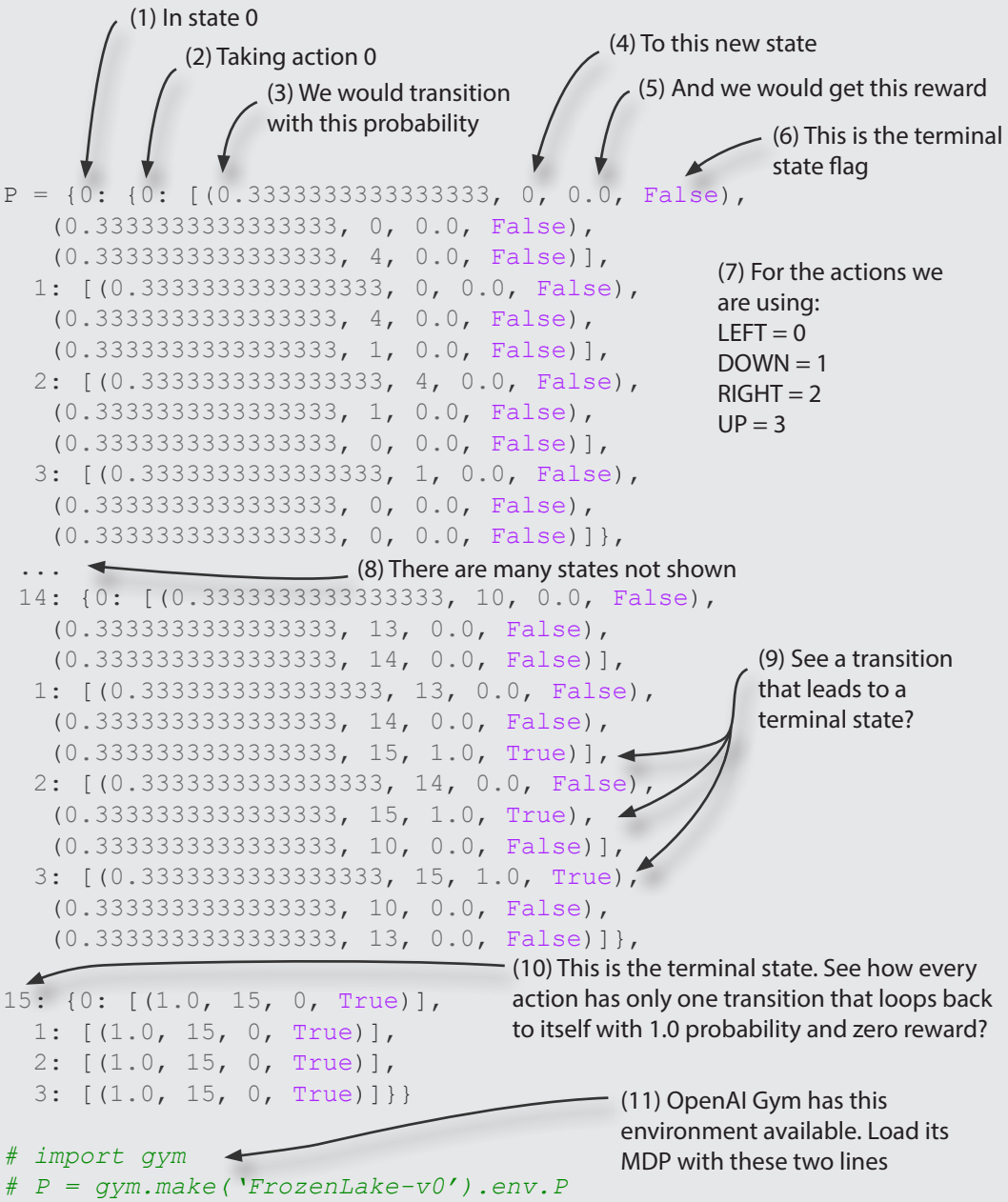
Frozen Lake MDP

As you've seen, this world has 16 states, 4 actions, and the only reward given is a +1 when the agent reaches the goal. There are 5 terminal states, 4 of which are holes that terminate the episode and give no reward, one of which is the goal state (which, again gives +1 reward). Let's look at how this MDP would look in code.

In code, we could represent this MDP as a Python dictionary of length 16. The keys of the dictionary would be the starting states of the transitions, and the values of each state would be a new dictionary. The keys of this second dictionary are the possible actions the agent can take in a given state. The values of this second dictionary would be a list containing: the possible next state, the probability of that transition, the reward for that transition and a flag determining whether the state is final or not. This last Boolean value is not required, but it simplifies the building of algorithms that solve MDPs, as you'll see later.

I Speak Python

The Frozen Lake MDP




```
P = {0: {0: [(0.3333333333333333, 0, 0.0, False),
    (0.3333333333333333, 0, 0.0, False),
    (0.3333333333333333, 4, 0.0, False)],
  1: [(0.3333333333333333, 0, 0.0, False),
    (0.3333333333333333, 4, 0.0, False),
    (0.3333333333333333, 1, 0.0, False)],
  2: [(0.3333333333333333, 4, 0.0, False),
    (0.3333333333333333, 1, 0.0, False),
    (0.3333333333333333, 0, 0.0, False)],
  3: [(0.3333333333333333, 1, 0.0, False),
    (0.3333333333333333, 0, 0.0, False),
    (0.3333333333333333, 0, 0.0, False)]},
 ...
 14: {0: [(0.3333333333333333, 10, 0.0, False),
    (0.3333333333333333, 13, 0.0, False),
    (0.3333333333333333, 14, 0.0, False)],
  1: [(0.3333333333333333, 13, 0.0, False),
    (0.3333333333333333, 14, 0.0, False),
    (0.3333333333333333, 15, 1.0, True)],
  2: [(0.3333333333333333, 14, 0.0, False),
    (0.3333333333333333, 15, 1.0, True),
    (0.3333333333333333, 10, 0.0, False)],
  3: [(0.3333333333333333, 15, 1.0, True),
    (0.3333333333333333, 10, 0.0, False),
    (0.3333333333333333, 13, 0.0, False)]},
15: {0: [(1.0, 15, 0, True)],
  1: [(1.0, 15, 0, True)],
  2: [(1.0, 15, 0, True)],
  3: [(1.0, 15, 0, True)]}}

# import gym
# P = gym.make('FrozenLake-v0').env.P
```

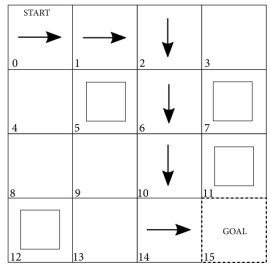
Objective of a decision-making agent

Let's start thinking about how to build an agent that can solve this environment. First, let's clearly define what it means for an agent to *solve* an environment.

The agent's goal is to find a sequence of actions that will maximize the sum of rewards (*discounted* or *undiscounted*—depending on the value of gamma) during the course of an episode or the entire life of the agent, depending on the task. The collection of rewards in a trajectory is called **expected returns**.

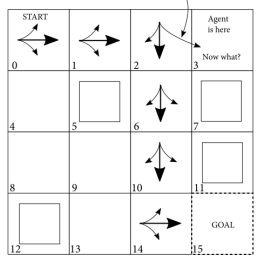
It may seem, therefore, that all the agent must find is something called a plan—that is a sequence of actions from the *START* to the *GOAL*.

A plan is a sequence of actions from an initial state to a goal state. This is only sufficient if the environment is deterministic. Not the case of the Frozen Lake!!

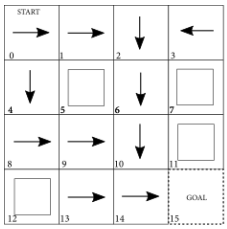


But this is not enough! Remember, as I mentioned before, the Frozen Lake environment is *stochastic*; actions taken on it will not always work the way we intend. What would happen if, due to the environment's stochasticity, our agent lands on a cell not covered by our plan?

In an stochastic environment, a plan will likely fail. In the Frozen Lake environment, unintended actions effects have even higher probability! 66.66% to 33.33%!!

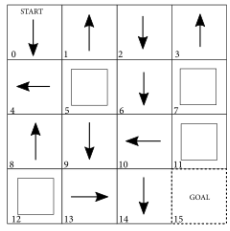


What the agent needs to come up with is called a **policy**. Policies can be *stochastic*; the policy will map a state to probabilities of selecting each possible action at that state. For the Frozen Lake example, however, we will use a *deterministic* policy, which simply maps states to actions. Let's look at all the components internal to a reinforcement learning agent that allows them to learn and find optimal policies.



In stochastic environments, we need a policy! A policy answers the question of what action to take for every possible environment state

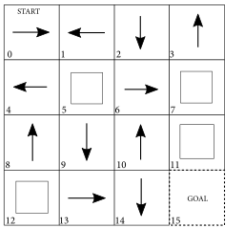
These are what policies looks like



Policies of action

Given the stochasticity in the Frozen Lake environment (and most reinforcement learning problems,) the agent needs to find a **policy**, denoted as π. A policy is a function that returns an action for any given nonterminal state (remember, though, policies can be stochastic, so this is not always true. We will expand on stochastic policies in later chapters.) Here is a sample policy:

This is a random policy.



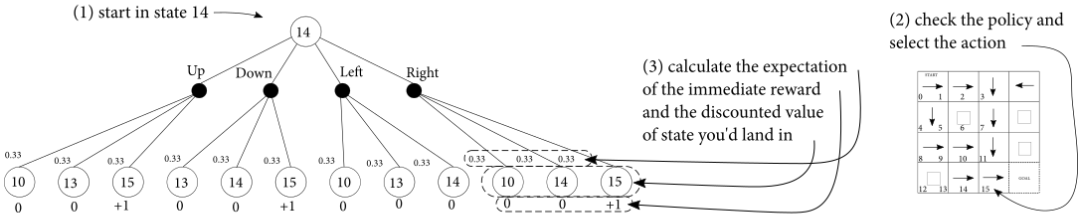
Don't let yourself get confused, this is not an optimal policy, but some of the actions that appear suboptimal are actually optimal! You'll see what I mean soon.

You can see how policies depend on neighboring states. Seen in reverse, if you start at a state next to the goal state, you can intuitively tell that going in the direction of the goal should be the optimal action to take on that state. However, we need a way to be more precise, especially given the stochasticity of the Frozen Lake environment.

Value of state

How can we numerically represent the value of being in a state? If our agent is in state 14 (to the left of the *GOAL*,) how is that better than being in state 13 (to the left of 14)? And precisely how much better is it?

The value of a state is the expectation of rewards by following a given policy



Uh-oh! You need the "value of the state you landed in," which you do not have! It's turtles all the way down. I told ya!

We defined *expected returns* as a function of future rewards that the agent is trying to maximize. We now define the value of states following a policy: the value of a state s under policy π is the expectation of returns if the agent follows policy π starting from state s. This definition is called the **state-value function** and it simply represents the return an agent can expect to receive given an environment state and following a given policy.

SHOW ME THE MATH

State-value function

(1) The value of a state s under a policy π

(2) Is the expectation

(3) of the returns at time t

(4) given we started in state s

(5) and follow policy π afterwards

$$v_\pi(s) = E_\pi[G_t|S_t = s]$$

(6) You remember the recursive definition of the returns

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$$

(7) If we expand on this, we get the Bellman equation for $V\pi$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \quad \forall s \in S$$

(8) To calculate the value of a state s under policy π, you would select the actions using your policy. For stochastic policies this could be multiple actions, for deterministic policies this would simply be a single action and this sum would be gone.

(9) Each action you take can potentially send you to multiple states. You have to get the probability of each transition so that you can weight the future.

(10) An the future is basically the immediate reward you obtain, and the discounted value of the state that you landed in.

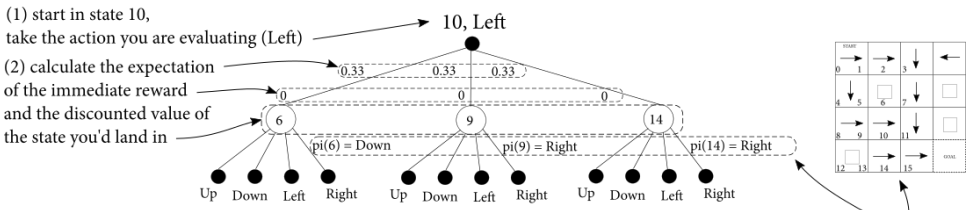
(11) This is true for all states.

Notice how the value of a state depends recursively on the value of many other states. The recursive relationship between states and successive states will come back in the next section, when we look at algorithms that can use these equations to solve the Frozen Lake MDP.

Value of taking an action

The value of a state depends on the value of taking actions in that state. Therefore, we need a way to compare the values of taking any action in any state, so that we can identify the best action and thereby come up with the best policy. The **action-value function** is the expectation of returns if the agent follows policy π starting from state s and taking action a.

The value of a state and action pair is the expectation of rewards by taking that action and by following a given policy



But, remember, the value of a state depends on following the policy. And, yeah, small detail, it is something you do not have...

Show Me The Math

Action-value function

(1) Just as before, the value of taking action *a* in state *s* under policy π

(2) Is the expectation of the returns if we were to take action *a* in state *s*

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$$

(3) And follow π thereafter

(4) This equation has the same recusive definition as before

(5) But this time we take and discount the state-value function of the next state as provided by the policy

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a]$$

(6) Do the sumation accross all possible new states and rewards

(7) weighted by their probability of occurring

(8) add the immediate reward and the discounted state-value function of the next state

(9) For all states and actions

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big], \ \forall s \in S, \ \forall a \in A$$

Advantage of taking an action

There is another type of value function derived from the previous two. The **advantage-value function** is the difference between the *action-value* function of action *a* in state *s* and the *state-value* function of state *s* all under policy π. The advantage-value function describes how much better it is to take a given action instead of following the given policy.

SHOW ME THE MATH
Advantage-value function

(1) The advantage of taking action *a* when in state *s* under policy π

(2) Is the difference between the value of taking that action in state *s* under policy π

$$a_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$$

(3) and the predicted value of the state in question

Optimality

Policies, state-value functions, action-value functions, and advantage-value functions are the components we use to describe, evaluate and improve behaviors. We call this **optimality** when these components are the best they can be. We call this optimality. An **optimal policy** is a policy with expected returns greater than or equal to the expected returns of all other policies. An **optimal state-value function** is a state-value function with the maximum value across all policies. An **optimal action-value function** is an action-value function with the maximum value across all policies. Note that although there could be *more* than one optimal policy for a given MDP, there can *only* be one optimal state-value function and optimal action-value function. You may also notice that if you had the optimal state-value function, you could simply use the MDP to do a one-step search for the optimal action-value function and then use this to build the optimal policy.

SHOW ME THE MATH
Bellman Optimality Equations

(1) The obvious but simpler definitions of optimality.

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in S$$

(2) Take the max of the respective value function for all states and actions

$$q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in S, \forall a \in A(s)$$

(3) across all policies

(4) The optimal value function of a state (see how we are not under a specific policy, optimal is accross all possible policies!)

(5) you take the max action

(6) and do a sum all possible transitions weighted by their probabilities of occurring

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

(7) the immediate reward and the discounted optimal value of the state you land with that transition

(8) the optimal value function of an action has a similar definition

(9) Just that we take the max over the choice of actions available in the next state

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$$

Planning optimal sequences of actions

So far, we have *state-value functions*, which keep track of the value of each state; *action-value functions*, which keep track of the value of taking actions in states; *advantage-value functions*, which represent the difference of action-value and state-value functions and therefore show the "advantage" of taking a particular action. We have equations for all of these to *evaluate current* policies and obtain these value functions, but we also have equations to calculate and *find optimal* value functions and optimal policies, which is really what we are going for.

Now that we have discussed the reinforcement learning problem formulation, and that we have defined the objective we are after, we can start discussing methods for finding this objective. Solving the equations defined in the previous section is one of the most common ways to solve a reinforcement learning problem and obtain optimal policies for when the dynamics of the environment, the MDP, are known. Let's look at some of these algorithms.

Evaluating policies of action

We talked about comparing policies in the previous section. We defined that policy π is better than or equal to policy π' if the expected return is better than or equal to π' for all states. Before we can use this definition, however, we must devise an algorithm for actually evaluating any arbitrary policy. Such algorithm is known as **iterative policy evaluation** or simply **policy evaluation**. The policy evaluation algo-

SHOW ME THE MATH
Policy Evaluation

(1) The policy evaluation equation iteratively approximates the true value of the policy under evaluation. As k approaches infinity, the state-value function will converge.

(2) You initialize v when $k=0$ to 0 for all states, then calculate the next approximation

(3) By taking the sum of all possible transitions weighted by their probability of occurring

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \Big[r + \gamma v_k(s') \Big]$$

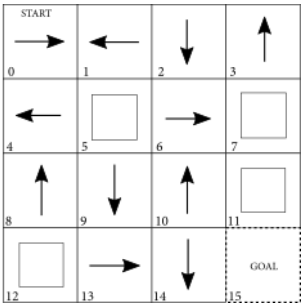
(4) Each transitions will use the immediate reward

(5) and the discounted value of the landing state as we had calculated on our previous iteration

rithm consists of calculating the state-value function for a given policy by iteratively sweeping through the state space and improving an estimate from an estimate. We refer to the type of algorithm that takes in a policy and outputs a value function as an algorithm that solves the **prediction problem**; calculating the values of a given policy.

Using the equation above, we can iteratively approximate the *true* state-value function of an arbitrary policy. The iterative policy evaluation algorithm is guaranteed to converge to the true value function of the policy given enough iterations, more concretely as we approach infinity. In practice, however, we use a small threshold to check for changes in the value function we are approximating. Once the changes in the value function are less than this threshold, we stop.

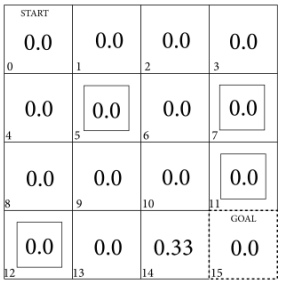
Recall this policy. Let's evaluate it.



Consider the following random policy for the Frozen Lake environment.

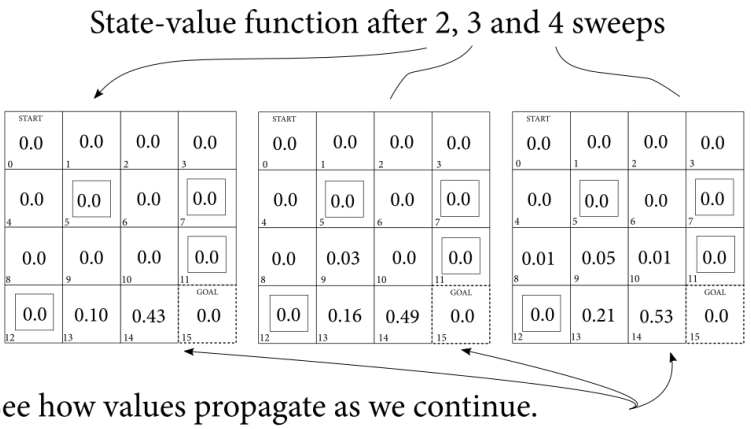
State-value function of the policy above after 1 sweep.
A single sweep updates the state-value for all states once.

Remember, you initialize the state-value function to zero, so the initial state-value function is 0 for all states.



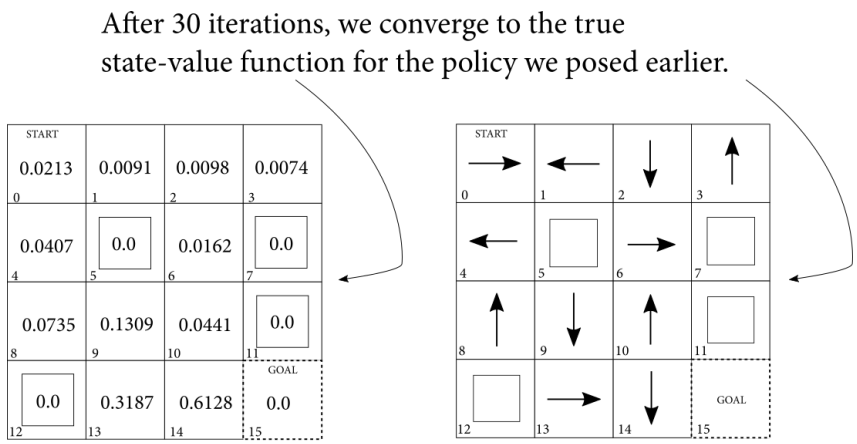
The true state-value function starts propagating!

A single policy evaluation sweep produces the following state-value function:



Iterations would continue, getting closer and closer to the true state-value function for the policy under evaluation.

Until the last two evaluations would return minimal (to no) changes for all the states of the MDP. For the policy we are evaluating, policy evaluation completes after a total of 30 iterations with the following state-value function:



Being able to evaluate policies allows us to compare, improve, and select the "best" policies for any problem.

I Speak Python

Policy Evaluation

(1) Import numpy

```
import numpy as np
```

(2) Make sure you already loaded the Frozen Lake MDP

(3) Create aliases for the actions

```
LEFT, DOWN, RIGHT, UP = range(4)
```

(4) Create the policy as in the diagrams

```
pi = {
    0:RIGHT, 1:LEFT,   2:DOWN,  3:UP,
    4:LEFT,  5:LEFT,   6:RIGHT, 7:LEFT,
    8:UP,    9:DOWN,   10:UP,   11:LEFT,
    12:LEFT, 13:RIGHT, 14:DOWN, 15:LEFT
}

def policy_evaluation(pi, P, gamma=0.9, theta=1e-10):
    V = np.zeros(len(pi))
    while True:
        max_delta = 0
        old_V = V.copy()

        for s in range(len(P)):
            V[s] = 0

            for prob, new_state, reward, done in P[s][pi[s]]:
                if done:
                    value = reward
                else:
                    value = reward + gamma * old_V[new_state]
                V[s] += prob * value

            max_delta = max(max_delta, abs(old_V[s] - V[s]))
        if max_delta < theta:
            break
    return V.copy()

V = policy_evaluation(pi, P)
V
```

(5) Initialize V for all states to zero

(6) We now loop "forever", every iteration we check for the max change in values

(7) We process all states of the policy on a single "sweep". First, hold the old state-value and set the new to zero to start the calculations

(8) See we go here through every transition for following the policy. pi[s] returns the action at s

(9) For final states assume future value is zero

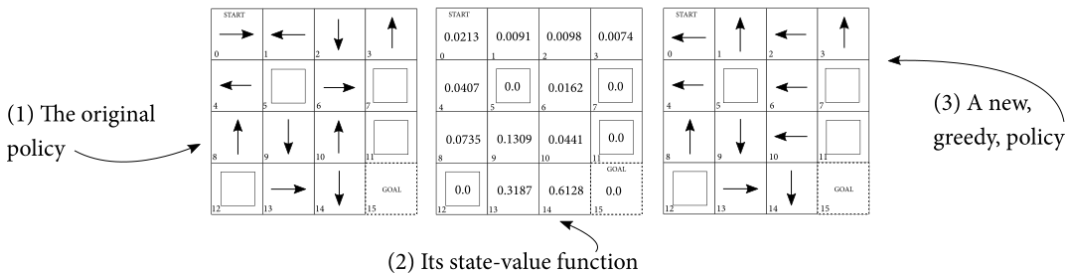
(10) We weight each transition values by their probability of occurring "prob"

(11) We check the max changes for all state-values and if the change is greater than a threashold, simply get out and return V

Improving policies of action

I hope you notice what happened in the last section. We evaluated a random policy, and the values we obtained showed that we are not, in fact, going in the direction of the highest state-value function. Let's take a closer look.

If we were to act greedily based on the state-value function that came out from evaluating our original policy, we could find a better policy than our original policy!!! And this is what we are looking for.



The step that obtains the **greedy policy** from a state-value function is the **policy improvement** algorithm. We can use the MDP we have available to look ahead and greedily improve our policy with respect to state-value function.

Think about it: we can now evaluate policies and obtain their true state-value function, and we can also improve policies using the *state-value function* and the *transition function*. Can you imagine how to combine these two and create an algorithm that goes from a random policy to an optimal policy? Let's first look at the math and code of the improvement step, and right after we will discuss such algorithm.

Show Me The Math

Policy Improvement

(1) To improve a policy we use the state-value function calculated from policy evaluation and then readjust the policy by creating a new policy that takes the action with maximum value

(2) The new policy is calculated by

(3) Taking the max action index

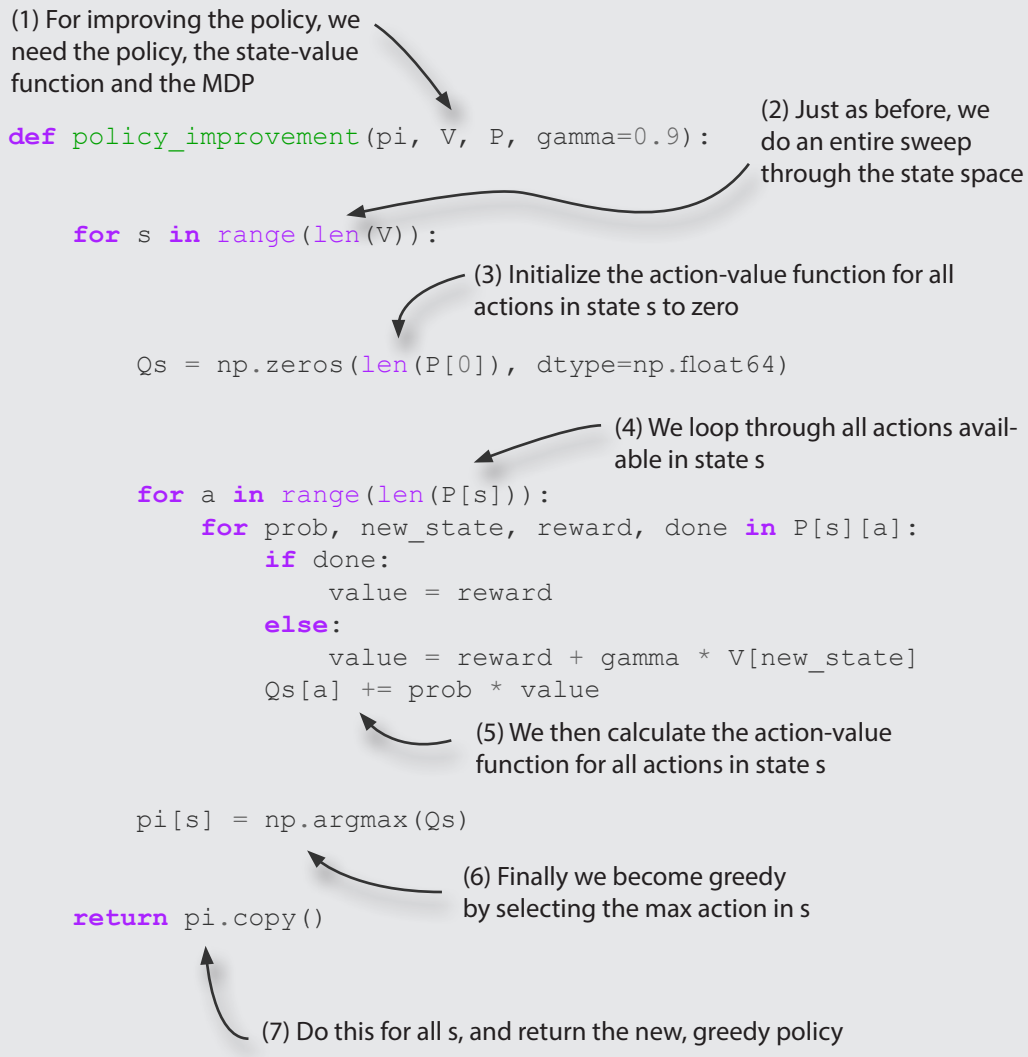
(4) Using the value function of the policy we evaluated

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

(5) So, we evaluated a policy and went from policy to state-value function. Then, created a new policy that is greedy with respect to the state-value function. Therefore, we got ourselves an improved policy!

I Speak Python

Policy Improvement

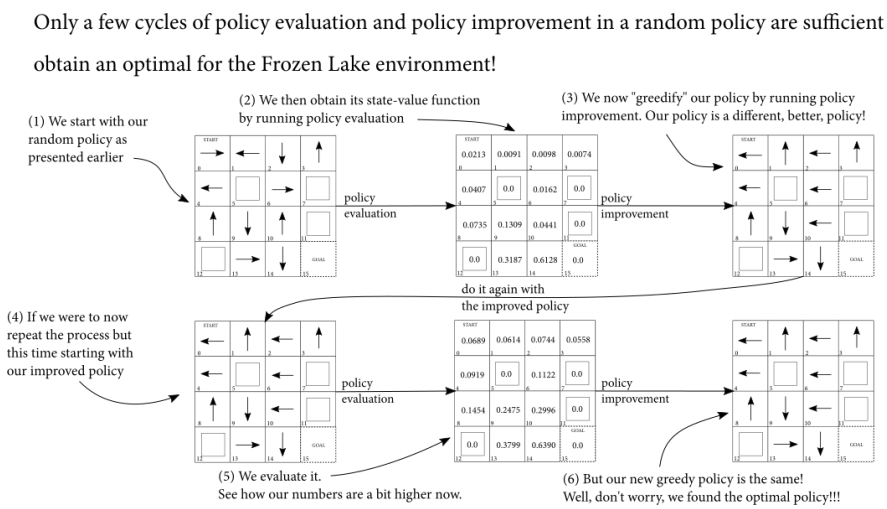



```
def policy_improvement(pi, V, P, gamma=0.9):

    for s in range(len(V)):

        Qs = np.zeros(len(P[0]), dtype=np.float64)

        for a in range(len(P[s])):
            for prob, new_state, reward, done in P[s][a]:
                if done:
                    value = reward
                else:
                    value = reward + gamma * V[new_state]
                Qs[a] += prob * value

        pi[s] = np.argmax(Qs)

    return pi.copy()
```

```
new_pi = policy_improvement(pi, V, P)
new_pi
```

Iterating over better policies

You saw how we can evaluate a policy and how can we improve it. Can you see the potential for an algorithm that will cycle between these two in order to find optimal policies? Let's see what a two-cycle *evaluation -> improvement* would look like:



Do you see how we are getting a better policy in the first iteration, and in the second we get the same policy? Do you remember when I mentioned that the new policy coming out the policy improvement algorithm is guaranteed to be at least equal to the policy evaluated? What does it mean if the new policy doesn't change? Just like in our second iteration, the policy did not change at all. The fact is that if there is no improvement then the policy and state-value function are guaranteed to be optimal.

With this knowledge, we can combine the policy evaluation and policy improvement algorithms to find better and better policies until we find the optimal state-value function and at least one optimal policy. As we saw before, an MDP can have more than one optimal policy, but it can only have a single optimal state-value function. If we iterate on a random policy between evaluation and improvement, we will monotonically reach a unique fixed point that will indicate we have arrive to the optimal state-value function and therefore an optimal policy. This algorithm is called **policy iteration**.

I Speak Python

Policy Iteration

(1) For the whole algorithm we only need the MDP. Note that gamma (the discount factor) is usually part of the MDP.

```
def policy_iteration(P, gamma=0.9):
```

(2) We create an array of random actions

```
    random_actions = np.random.choice(tuple(P[0].keys()), len(P))
```

(3) And with them, we create an initial random policy

```
    pi = {s:a for s, a in enumerate(random_actions)}
```

(4) We loop "forever"

(5) Save a copy of our 'previous' policy

```
    while True:
        old_pi = pi.copy()
```

(6) Evaluate the random policy to get the state-value function, and save a copy of the policy

```
        V = policy_evaluation(pi, P, gamma)
        pi = policy_improvement(pi, V, P, gamma)
```

(7) We now improve on the policy to get a greedy policy with respect to the evaluation

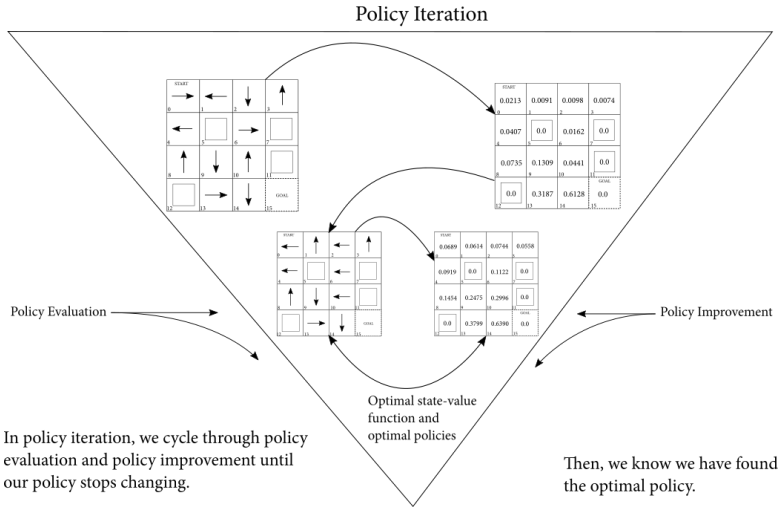
```
        if old_pi == pi:
            break
```

(8) We compare for changes in the policy. If it didn't change, it means we found an optimal policy

```
    return V, pi
```

(9) And we return the optimal state-value function and an optimal policy

```
V_star, pi_star = policy_iteration(P)
V_star, pi_star
```



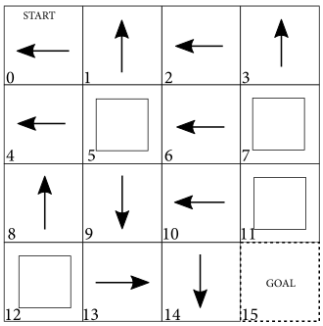
Because we have the *full dynamics* of the environment, policy iteration is guaranteed to converge and find an optimal policy. Because there could be multiple optimal policies, you need to make sure in your implementation that it doesn't get stuck going back and forth between two optimal policies. But policy iteration won't get stuck in local optima, however.

By applying the policy iteration algorithm to the Frozen Lake environment MDP, we obtain the optimal policy for that environment. Due to the stochasticity of the environment, some actions may seem to be slightly off; they are somewhat counter-intuitive. This is why it is important to have algorithms that can derive optimal policies. Sequential decision making under uncertainty of actions is a hard problem. Let's take a look at the Frozen Lake environment optimal policy.

Yes, this is Frozen Lake's optimal policy!

And I wish you at least question the validity of it. How is it possible that in state 14 the optimal action is to go DOWN? Shouldn't it be RIGHT?

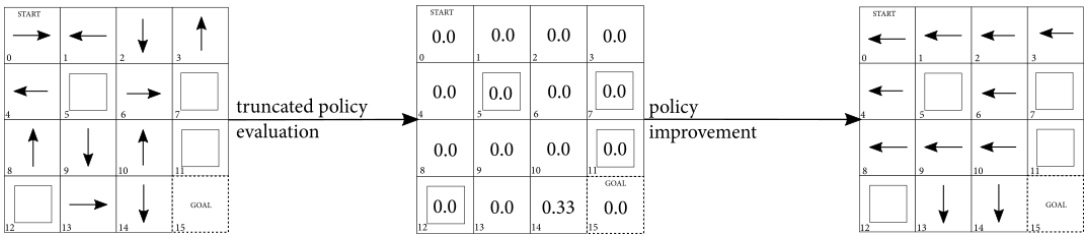
Since the Frozen Lake environment stochasticity is so large, it makes it a great example to illustrate the difficulty of sequential decision making under uncertainty of action effects. The fact is, the RIGHT action would send us to state 10, 14 or 15 with equal probability. While DOWN would send us to state 13, 14 or 15. If you think about it, state 13 is preferred to 10 because we can only get to state 15 safely going LEFT in 10, DOWN in 9, RIGHT in 13 and DOWN in 14. On the other hand, going DOWN in state 14 keeps us only a few steps away from 15 regardless of where the environment sends us.



Calculating exact values

You may have been able to notice that even after a single iteration of policy evaluation, the greedy policy we could obtain was already a better policy than the one we were evaluating.

> In value iteration, we stop policy evaluation after the first sweep and improve over that new state-value function.



Is it possible, then, to stop evaluating early and improve our policy based on that state-value function instead? Will a method like this also converge to the optimal policy? There are, in fact, multiple ways of combining evaluation and improvement steps and still guaranteeing convergence.

Let's look at an algorithm called **value iteration**, which stops after a full single sweep of policy evaluation and improves the policy in a single iteration.

Show Me The Math

Value Iteration

(1) We can merge a truncated policy evaluation step and a policy improvement into the same equation

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_k(s')\right]$$

(2) by taking the max value across the available actions

(3) And we calculate the values by weighting

(4) the sum of the immediate reward

(5) and the discounted value of the future state

(6) that we calculated in the previous iteration

(7) with the probability of landing in that future state and getting that reward given we take action a in state s

I Speak Python

Value Iteration

```
def value_iteration(P, gamma=0.9, theta = 1e-10):
    V = np.random.random(len(P))

    while True:
        max_delta = 0
        Q = np.zeros((len(P), len(P[0])), dtype=np.float64)

        for s in range(len(P)):
            v = V[s]

            for a in range(len(P[s])):
                for prob, new_state, reward, done in P[s][a]:
                    if done:
                        value = reward
                    else:
                        value = reward + gamma * V[new_state]
                    Q[s][a] += prob * value

            V[s] = np.max(Q[s])
            max_delta = max(max_delta, abs(v - V[s]))

        if max_delta < theta:
            break

    pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
    return V, pi

V_star, pi_star = value_iteration(P)
V_star, pi_star
```

(1) Initialize a state-value function arbitrarily for all s in S

(2) Initialize an action-value function for all states and all actions to zero

(3) We will do a full sweep

(4) Cache the old value for the state

(5) We check all possible transitions from s

(6) We calculate the action-value function for all state, action pairs

(7) The state-value function is going to be the max action-value function from state s

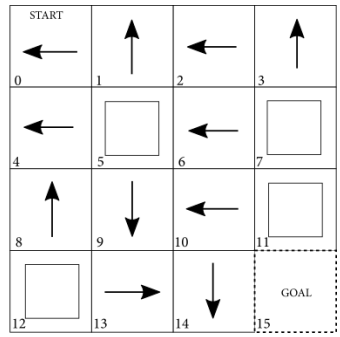
(8) If changes are minimal, stop the main loop

(9) Obtain the optimal policy

(10) Return both the optimal state-value function and an optimal policy

As expected, after over 100 iterations, value iteration comes up with the same optimal policy.

Value iteration helps us validate Frozen Lake's optimal policy

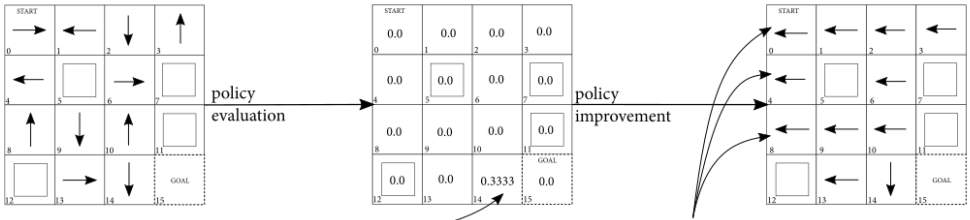


Other methods for finding optimal policies

It is important to mention that there are other methods for finding optimal policies for an MDP. The most obvious, perhaps, is solving the set of linear equations we can define using the optimal state-value function. We would have n number of equations with n unknowns in problems with n states, so sixteen in the Frozen Lake environment. You could also use some form of search on the space of policies trying to find better policies, but this process consumes lots of computer resources, and it doesn't have convergence guarantees as nice as the iterative methods we've looked at in this chapter. There are also other variants of the algorithms we looked at above that change either on the number of sweeps we evaluate policies or on the number of states that we evaluate on each sweep. The more popular ones are:

- Asynchronous value iteration
- Modified policy iteration

Asynchronous value iteration, for example, runs the value iteration update on a single state per iteration



Let's pretend that we selected randomly state 14 to be updated.

All these left actions are just the result of breaking ties from all 4 actions which are all zero valued.

I Speak Python

Frozen Lake Playground

(1) Use these code snippets to play around

```
# change transition function
prob_action, drift_right, drift_left = 0.8, 0.1, 0.1
for s in range(len(P)):
    for a in range(len(P[s])):
        for t in range(len(P[s][a])):
            if P[s][a][t][0] == 1.0:
                continue
            values = list(P[s][a][t])
            if t == 0:
                values[0] = drift_left
            elif t == 1:
                values[0] = prob_action
            elif t == 2:
                values[0] = drift_right
            P[s][a][t] = tuple(values)
```

(2) You can modify these numbers to see what would happen if the stochasticity of the environment was different. Try 80%, 10%, 10%, or even 100%, 0%, 0%.

```
# change reward function
goal, hole = 15, [5, 7, 11, 12]
reward_goal, reward_holes, reward_others = 1, -1, -0.01
for s in range(len(P)):
    for a in range(len(P[s])):
        for t in range(len(P[s][a])):
            values = list(P[s][a][t])
            if values[1] == goal:
                values[2] = 1
                values[3] = False
            elif values[1] in hole:
                values[2] = -1
                values[3] = False
            else:
                values[2] = -0.01
                values[3] = False
            if s in hole or s == goal:
                values[2] = 0
                values[3] = True
            P[s][a][t] = tuple(values)
```

(3) This snippet will change the reward function. What do you think would happen if reaching the goal was, say, +10, and falling in any of the goals, something like -100. Would this change the optimal policy? Try things out. +1, -1 and -0.01 for goal, hole and step. Or maybe +1, -1, -0.5. Go and have fun!

Summary

This chapter gave you an in-depth overview of a mathematical framework for representing sequential decision-making problems known as Markov Decision Processes. You know that MDPs are commonly composed of states, actions, transition functions, reward functions, and time-related variables such as timesteps, episodes, and a discount factor. You also know the main components of a reinforcement learning agent: policies, state-value functions, action-value functions, and advantage-value functions. You know that the objective of a reinforcement learning agent is to maximize the amount of discounted future reward it can obtain from an environment, and you know that agents can do this by finding optimal policies of actions that, under the surface, carry optimal value functions.

You also learned about general algorithms that can be applied to a variety of sequential decision-making problem to find optimal policies. You learned about policy evaluation for estimating state-value functions; policy improvement for optimizing policies by using their state-value functions; policy iteration to cycle and find optimal policies; and value iteration, a truncated version of policy iteration. We concentrated on representing a sequential decision-making problem known as the Frozen Lake environment, but you know that you can apply the same algorithms to a variety of problems.

In this chapter, you learned about balancing immediate and long-term goals. In the next chapter, we will explore another difficult tradeoff in reinforcement learning: the exploration and exploitation tradeoff. This becomes relevant when the dynamics of the environment, the MDP, are either unknown or just too large and difficult to solve using the methods we learned in this chapter. When this happens, the agent is no longer able to plan ahead of time. Instead, the agent must learn by trial and error—by interaction. We will start with a non-sequential, single-state environment, so that you can see the tradeoff in isolation; but in the second part of the chapter, we will add the sequential part of reinforcement learning back in and face, for the first time, the full reinforcement learning problem.

By now you:

- Recognize Markov Decision Processes and know how they work.
- Can represent sequential decision-making problems as MDPs.
- Understand how a reinforcement learning agent manages policies and value

functions to behave optimally.

- Know how to implement policy and value iteration, the most important algorithms for solving sequential decision-making problems.

Learning to Act Through Interaction 3

IN THIS CHAPTER

You'll solve decision-making problems when the dynamics of the environment are unknown.

You'll understand evaluative feedback by implementing different exploration strategies for reinforcement learning agents.

You'll make predictions to help reinforcement learning agents behave optimal in the face of model uncertainty.

You'll create agents that learn to behave optimally solely through interaction and experimentation.

"Our ultimate objective is to make programs that learn from their experience as effectively as humans do."

— John McCarthy
Founder of the field of Artificial Intelligence
Inventor of the Lisp programming Language

No matter how small the decision may seem, every decision you make is an intersection of two paths: information gathering and information exploitation. You go to your favorite restaurant. Should you order your favorite dish, or should you order that dish you can barely pronounce? A Silicon Valley startup offers you a job. Should you make a huge career move, or should you stay put in your current safe role? These kinds of questions are central to the *exploration-exploitation dilemma* and are at the core of the reinforcement learning problem. It boils down to deciding when to *acquire knowledge* and when to *capitalize on knowledge* previously acquired. It is a challenge to know whether the good we already have is good enough. When do we settle? When do we go for more?

The rewarding moments in life are *relative*; you need to be able to compare things to see a clear picture of their value. For example: how great did it feel when you got offered your first job? Awesome? Super awesome? But probably not better than the first time you saw your child, right? Even if we could to rank these moments, there would still be a challenge. You may know what the most rewarding moment you've encountered so far is, but you still can't know what the most rewarding moment you could experience is—maybe there is something even better out there that you have not experienced yet.

How do you balance the search for a more rewarding life and enjoying that which you already possess? Difficult problem. This same exploration-exploitation dilemma is present in reinforcement learning, and it is what makes it a unique branch of the machine learning community. Evaluative feedback, which is basically what life as we know it gives humans, gives rise to the need for exploration. As you will see later, it is not optimal to adopt the strategy of always attempting to experience the most rewarding events you know. It is not optimal to always get the same dish at your favorite restaurant! To find optimality, you are also required to try new things and discover valuable moments you didn't know existed.

In this chapter, you'll explore the next tradeoff a reinforcement learning agent faces: agents must learn to balance the value of acquiring new knowledge and the value of making use of previously acquired knowledge to maximize the amount of reward they can get over time. I'll first show you how to deal with this issue in a single-state environment. Restricting the environment to a single state will allow us to zoom in and concentrate on this second tradeoff in isolation from the one we discussed in the previous chapter (balancing immediate and long-term rewards). Later in this chap-

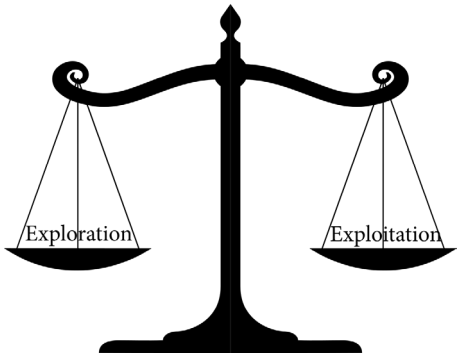
ter, I will add back that first, sequential aspect of reinforcement learning problems, combining these two tradeoffs to present to you, for the first time, the full reinforcement learning problem. You'll learn the most popular types of agents that can not only accurately evaluate policies of behavior, but can also learn to behave optimally, purely through interaction and trial and error.

The challenge of interpreting evaluative feedback

In the last chapter, when we tried to solve the Frozen Lake environment, we came to it with the knowledge of how the environment would react to any of our actions. Knowing the transition dynamics and reward function allowed us to find an optimal policy ahead of time using planning algorithms such as value iteration or policy iteration. These algorithms let us solve reinforcement learning problems without having to interact with the environment at all. But knowing the MDP in advance oversimplifies things, perhaps unrealistically. We cannot always assume we will know with precision how an environment will react to our actions—that's simply not how the world works. We need to let our agents interact and experience the environment by themselves, learning this way to behave optimally, solely from their own experience.

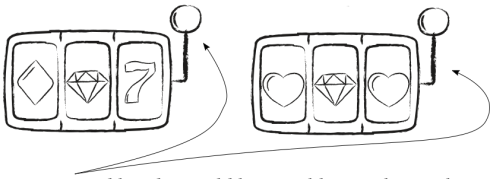
In online decision-making, which is when we are learning to behave from interaction, the environment asks the same question over and over: what do you want to do now? This question presents a fundamental challenge to a decision-making agent. What action would you like to take now? Should you, the agent, **explore** actions that you haven't tried enough? Or should you **exploit** your current knowledge by taking an action that you know pays best so far? Should you try to explore now or later? Should you explore often? The exploration-exploitation dilemma is such that you cannot pick a single action that provides you with both the maximum amount of information and the maximum amount of reward. And unfortunately, both exploratory and greedy (exploitative) action selections are of value to a decision maker. Exploration builds the knowledge that allows for effective exploitation, and maximizing exploitation is the end goal of any decision maker.

If you emphasize one, then the other one will be lacking. The trick is to balance it.



Single state decision problem

Multi-armed bandit problem



A 2-armed bandit would be a problem with two slot machines. The challenge is to maximize the amount of money you could get from them.
To do so, you need to possibly try them both. But, how do you balance exploration and exploitation?

Environments that have a single state and multiple actions are a well-studied problem known as **multi-armed bandits**. The name refers to a slot machine (a bandit) with multiple arms you can pull, but you could also see it as multiple slot machines. The challenge is that you'd like to get the most amount of money out of them, but for that, you must try all of them multiple times, unless you are ridiculously lucky. The multi-armed bandit problem is the exploration-exploitation tradeoff at its core.

Many fields use the study of multi-armed bandit problems in real life. Online advertisement needs to balance between showing you a banner you are likely to click on and showing you a different banner that you might find even more interesting. Websites that are raising money need to balance between showing the layout that has led to most contributions and new designs with more potential. Many other problems benefit from the study of the exploration-exploitation tradeoff: oil drilling, game playing, search engines, etc.

Before we unleash the full reinforcement learning problem, I would like you to dive into a bandit problem and analyze it further.

Show Me The Math

Multi-armed bandits

(1) Multi-armed bandit is a set of actions and a set of rewards $MAB(A, R)$

(2) each action gives a reward with an unknown probability $R^a(r) = \mathcal{P}[R = r|A = a]$

(3) your goal is to maximize the total cummulative reward from the first to the last timestep t (or episode) $\sum_{t=1}^{T} R_t$

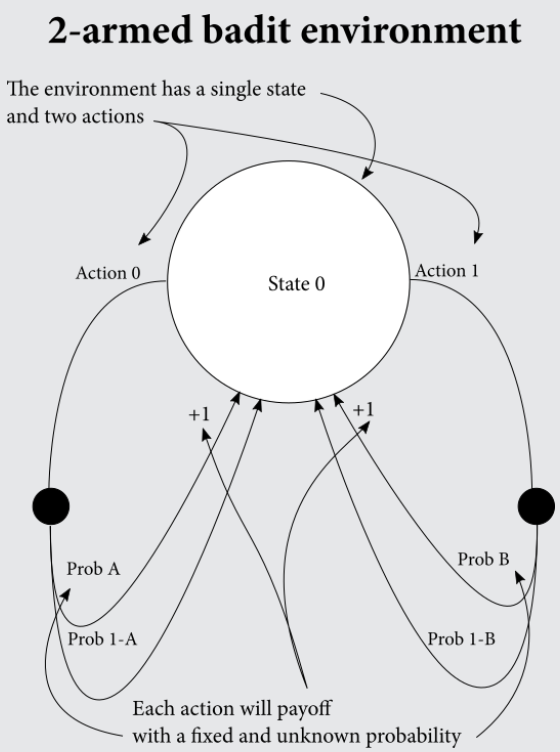
(4) The action-value function of any action is the expectation payoff $q(a) = \mathbb{E}[R|A = a]$

(5) You would really like to find the optimal action-value function, which is the same as the optimal state value function for the single state MDP $v_* = q_*(a) = \max_\pi q_\pi(a) = \max_{a \in A} q(a)$

The Challenge

2-armed Bernoulli bandit

The challenge is a problem with a single state and two actions. Having a single state makes it so that this is not a *sequential* decision-making problem, that means, taking an action will not determine what future states you will experience, because well, there is just one state.



Everytime you select one of the two available actions, the environment will payoff with some unknown probability a reward of +1. Each action has a different probability of paying off. In order to find the best action, you will have to explore. But given that your goal is to maximize the cummulative reward obtained throughout all episodes, you will also have to be as greedy as possible.

The environment is stationary, that means the transition probabilities and rewards will not change over time, every new episode is a new chance to interact with the same environment. But this time, you don't know what's under-the-hood as you did for the Frozen Lake environment. To solve this problem, you still need to find an optimal policy. The policy you are looking for, however, being a single state environment, is just a single action. The challenge is that you must find this action solely through interaction.

You won't be able to create a perfect agent, because your agent needs to interact, explore, and make mistakes in order to learn and accumulate the highest amount of reward possible.

As we saw in the previous chapter, long-term strategies involve short-term sacrifices. When discussing sequential decision-making problems the short-term vs. long-term balance involved becoming less sensitive to immediate rewards so that we could access later states in which we could obtain higher rewards. There was a *temporal* connection. In this chapter, short-term vs. long-term balance is about how to learn optimal behavior when facing the same problem over and over. The need is for *exploration*, which entails sacrificing our current beliefs, so that we can improve our estimates and better *exploit* in the long run.

In reinforcement learning, there are three major approaches to solving this problem. The most popular and simplest way of handling exploration is to act greedily most of the time, but with a small probability act randomly. In **random exploration strategies**, we draw a uniform random number every time we face a decision-making step. If the random number is less than an epsilon threshold, we pick a random action. There are multiple ways of picking this random action; we could take one of the available actions with equal probability, or perhaps we could pick actions with a high estimated value with high probability. There are also different ways of handling the epsilon threshold value. We could use a fixed epsilon value; we could also vary it over time, and so on.

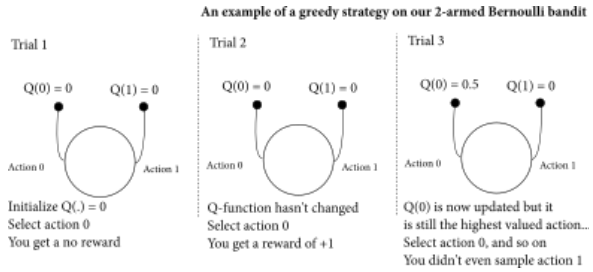
Another approach to dealing with the exploration-exploitation dilemma is to be optimistic. Yep, your mom was right. The family of **optimistic exploration strategies** is a more systematic approach that quantifies the uncertainty in the decision-making problem and puts a preference on states with the highest uncertainty. The bottom line is that being optimistic will naturally drive you toward uncertain states because you will assume that states you haven't experienced yet are the best states. This assumption will help you explore, and as you explore and come face to face with reality, your estimates will get lower and lower as they approach their true values.

The third approach to dealing with the exploration-exploitation dilemma is the family of **information state-space exploration strategies**. These strategies will model the information state of the agent as part of the environment. Encoding the uncertainty as part of the state space means that an environment state will be considered a different state when unexplored than explored. Encoding the uncertainty as part of the environment is an effective approach but can also considerably increase the number of states in a problem and, therefore, its complexity. In this book, we will explore many of these approaches. This chapter will discuss a handful of basic strategies, and in the

next chapter we'll study more complex systematic approaches. For all the strategies we will evaluate in the next sections, we will use the mean of all the payouts per action as the estimated value of taking that action. That is, the action-value function for action *a* will be equal to the mean of all payouts after selecting action *a*.

Always pick the action with the highest value

I already mentioned we need to have some exploration in our algorithms—otherwise, we will have suboptimal behavior. But for the sake of comparison, let's consider an algorithm with no exploration at all.



I Speak Python

Greedy Action Selection Strategy

```
def greedy_strategy(env, n_trials=10000):
    Q = np.zeros((env.action_space.n))
    N = np.zeros((env.action_space.n))

    returns = np.empty(n_trials)
    for t in range(n_trials):
        action = np.argmax(Q)

        _, reward, _, _ = env.step(action)

        N[action] += 1
        Q[action] = Q[action] + (reward - Q[action])/N[action]

        returns[t] = reward
    return np.cumsum(returns)/(np.arange(n_trials)+1)
```

(1) Initialize Q values and counts to zero for all actions

(2) returns just keeps track of the rewards received

(3) for every new trial, select the action with the highest value, break ties consistently

(4) Act in the environment, and observe a reward

(5) Increment the count for that action

(6) Calculate the new mean incrementally

(7) Store the reward for this trial

(8) After all trials, return the running mean

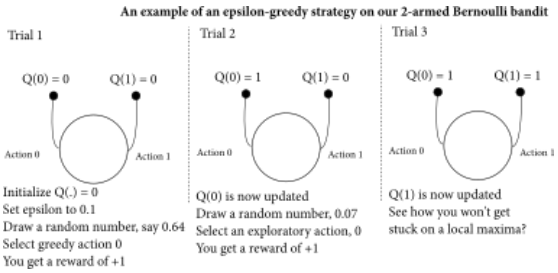
Picking the action with the highest estimated value is not a smart strategy. If we were to have a **greedy strategy**, that is an agent that would always select the action with the highest current estimate, we would quickly get stuck in a local maximum. You do not know what the *true* action values are, so being greedy can lock you onto a

suboptimal action forever, simply because your estimates can be wrong, especially early on.

A **random strategy** is obviously not a good strategy either and will also give you suboptimal results. Like being greedy all the time, you also do not want to explore all the time. We need algorithms that can do both exploration and exploitation.

Almost always pick the action with the highest value

One of the simplest ways of balancing exploration and exploitation is to act greedily most of the time and explore, every so often, by acting randomly. Let's say we draw a uniform random number, *delta*, and if delta is less than a small constant value, *epsilon*, then we pick an action randomly from the set of all available actions for a given



I Speak Python

Epsilon-Greedy Action Selection Strategy

```
def e_greedy_strategy(env, epsilon=0.1, n_trials=10000):
    Q = np.zeros((env.action_space.n))
    N = np.zeros((env.action_space.n))
    returns = np.empty(n_trials)
    for t in range(n_trials):
        exploit = np.random.random() > epsilon
        action = np.argmax(Q) if exploit else \
                 env.action_space.sample()
        _, reward, _, _ = env.step(action)
        N[action] += 1
        Q[action] = Q[action] + (reward - Q[action])/N[action]
        returns[t] = reward
    return np.cumsum(returns)/(np.arange(n_trials)+1)
```

(1) Initialize Q values and counts to zero for all actions

(2) returns just keeps track of the rewards received

(3) Draw a uniform random number, compare it to epsilon

(4) Exploit if larger than epsilon, explore otherwise.

(5) Act in the environment, and observe a reward

(6) Increment the count for that action

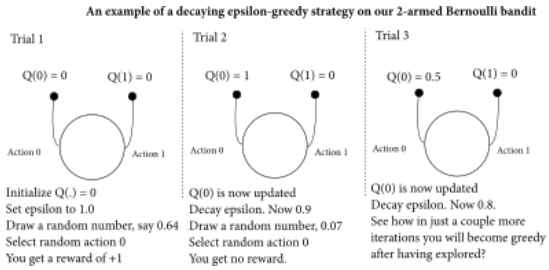
(7) Calculate the new mean incrementally

(8) Store the reward for this trial

(9) After all trials, return the running mean

state. This strategy, referred to as **epsilon-greedy strategy**, works surprisingly well. If you select the action you think is best *almost* all the time, you will get solid results because you are acting greedily, that is maximizing rewards by selecting the action believed to be best. But you are also exploring actions you haven't tried sufficiently yet. This way your action-value function has an opportunity to converge to its true value; this will, in turn, help you obtain more rewards in the long term.

First maximize exploration, then maximize exploitation



One thing we could easily improve on the previous strategy is to explore with high probability early on, when uncertainty is high and you haven't had a chance to learn. But, as time goes on, you could behave more and more greedily using the knowledge acquired. The mechanics are very simple. Start with a high epsilon less than or equal to one, and decay its value on every step. You

I Speak Python

Decaying Epsilon-Greedy Action Selection Strategy

```
def decay_e_greedy_strategy(env, init_epsilon=1.0,
                            decay_rate=1e-4, min_epsilon=0.0, n_trials=10000):
    Q = np.zeros((env.action_space.n))
    N = np.zeros((env.action_space.n))
    returns = np.empty(n_trials)
    for t in range(n_trials):
        new_epsilon = init_epsilon * np.exp(-decay_rate * t)
        epsilon = max(new_epsilon, min_epsilon)
        exploit = np.random.random() > epsilon
        action = np.argmax(Q) if exploit else \
                 env.action_space.sample()
        _, reward, _, _ = env.step(action)
        N[action] += 1
        Q[action] = Q[action] + (reward - Q[action])/N[action]
        returns[t] = reward
    return np.cumsum(returns)/(np.arange(n_trials)+1)
```

(1) Initialize Q values and counts to zero for all actions

(2) returns just keeps track of the rewards received

(3) exponentially decay epsilon that we explore a lot early on, but exploit a lot in later trials

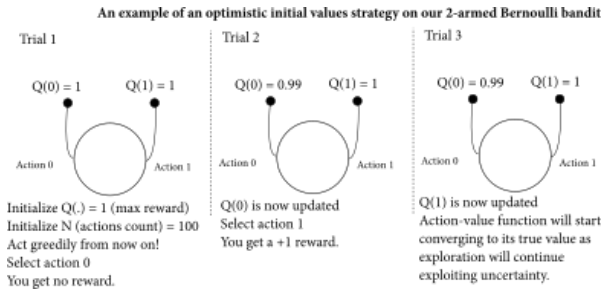
(4) we can make sure we continue exploring by clamping epsilon to a minimum desired value

(5) the rest stays the same: take the e-greedy action

can imagine different ways of approaching **decaying epsilon-greedy strategy**, from clipping the epsilon to a minimum to multiple ways of decaying the epsilon: linear decay rate, exponential decay rate, sine wave decaying, etc. The point is to explore with higher probability early and exploit with higher probability later. Early on there is a high chance that your value estimates are wrong, but as time passes and you acquire knowledge, the chance that your value estimates are close to the true values increases. This is when you should explore less frequently, so that you can exploit the knowledge you have acquired more often.

Start off believing it's a wonderful world

Another interesting approach to dealing with the exploration-exploitation dilemma is to treat actions that you haven't sufficiently explored as if they were the best possible actions—like you are truly in paradise. Optimistic initial values encourage the exploration of unknown values by giving a high prior estimate to all action values. As we interact with the environment, our mean estimate will



I Speak Python

Optimistic Initial Values Action Selection Strategy

```
def optimistic_strategy(env, optimistic_estimate=1.0,
                        n_trials=10000):
    Q = np.full((env.action_space.n), optimistic_estimate,
                dtype=np.float64)
    N = np.full((env.action_space.n), 100, dtype=np.float64)
    returns = np.empty(n_trials)
    for t in range(n_trials):
        action = np.argmax(Q)
        _, reward, _, _ = env.step(action)
        N[action] += 1
        Q[action] = Q[action] + (reward - Q[action])/N[action]
        returns[t] = reward
    return np.cumsum(returns)/(np.arange(n_trials)+1)
```

(1) This time, fill the Q table with an optimistic estimate

(2) Also fill the counts to a high enough value so the mean estimates don't drop too quickly

(3) you then act greedily and the means will start dropping towards their true values

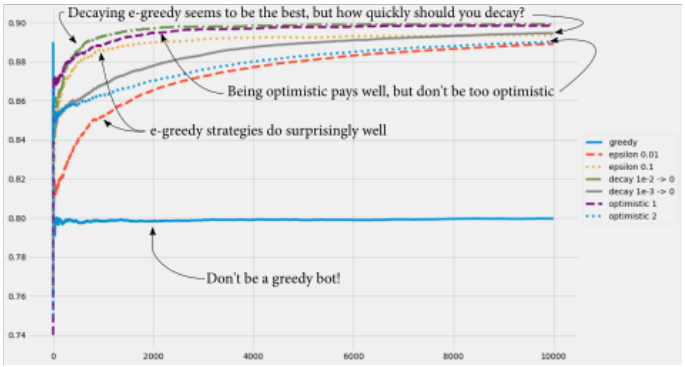
(4) everything else stays the same as before

start converging to lower, but better, estimates of the true mean allowing us to get and converge quickly to the action with the highest payoff. One of the simplest practical ways of doing this is to initialize your action-value function Q for all actions to the maximum possible reward. Like you are truly in paradise. You would then initialize the action-selection counts for each action N to a high enough value so that averages would not change too rapidly, then act greedily using these estimates. Take note: the "maximum possible reward" is something we do not have in a model-free setting, something we will address in the next chapter. But for now, this strategy, known as **optimistic initial values strategy**, will help us introduce with ease a class of strategies known as *optimism in the face of uncertainty*.

Comparing the different exploration strategies

Out of all strategies, the clear loser is the greedy action selection. Early on, it pays off well, but it quickly gets stuck in local maxima, and it performs poorly in the long-run. On the other hand, decaying e-greedy and optimistic initial value are the best-performing strategies. But, if you look closely, you will notice that very simple strategies such as epsilon-greedy also perform well. And, given enough time, these strategies may even catch up to the best ones.

Different action-selection strategies on the 2-armed Bernoulli environment

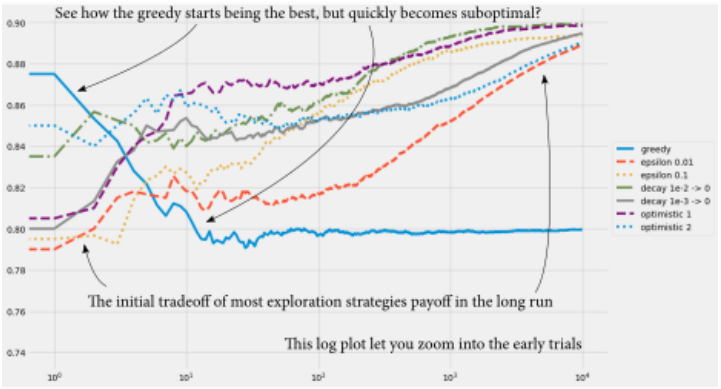


In addition to the 2-armed Bernoulli bandit environment described earlier, I ran the same action selection strategies on a 10-armed Gaussian bandit environment. In this environment, each arm will always pay. The *mean payoff* for each arm, however, is drawn from a normal distribution from 0 to 1 and variance 1 in advance. So, the *actual reward* received after selecting an action will be different on each pull. Also, the rewards we could see range approximately from -3 to 3, instead of being a fixed +1 reward. As you can imagine, this is a more challenging environment.

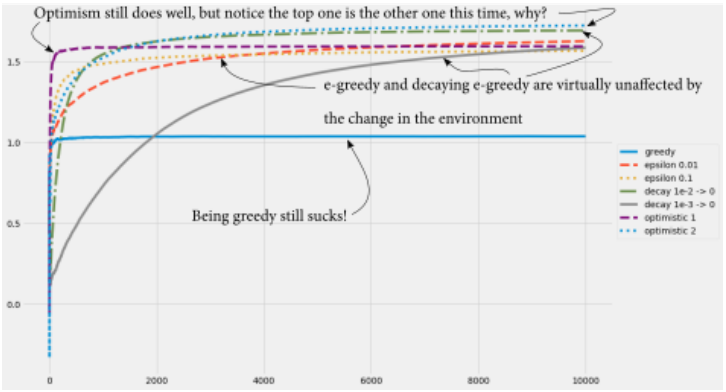
As we can see in the plots, though, we get similar results. But the optimistic initial values strategy that is on top this time is not the same strategy as in the previous example. The reason for this change is that in the previous environment the maximum possible payoff was +1, but in this environment, the maximum possible payoff is higher than that. So our "optimistic" initialization is not that optimistic anymore.

The simple strategies we have looked at so far are the most commonly used in practice. What you have learned about so far can serve you well for most reinforcement learning problems, but in the next chapter, we will look at a few more strategies that build on the strategies in this chapter and make use of more advanced techniques.

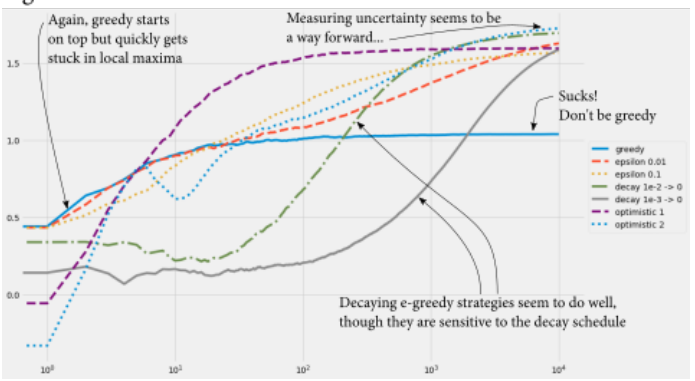
Same strategies, same environment, but this time plotted on a log scale



Different action-selection strategies on the 10-armed Gaussian environment



Same strategies, same environment, but this time plotted on a log scale



I SPEAK PYTHON

Run All Strategies

```
import gym
import gym_bandits
import numpy as np
from tqdm import tqdm


def run_exploration_strategies_experiment(env):
    ret_eg1, ret_eg2, ret_eg3 = [], [], []
    ret_egd1, ret_egd2 = [], []
    ret_os1, ret_os2 = [], []
    np.random.seed(12345)

    for i in tqdm(range(100)):
        env.reset()
        ret_eg1.append(greedy_strategy(env))
        ret_eg2.append(e_greedy_strategy(env, epsilon=0.01))
        ret_eg3.append(e_greedy_strategy(env, epsilon=0.1))
        ret_egd1.append(e_greedy_decay_strategy(
                            env, decay_rate=1e-2))
        ret_egd2.append(e_greedy_decay_strategy(
                            env, decay_rate=1e-3))
        ret_os1.append(optimistic_strategy(
                            env, optimistic_estimate=1))
        ret_os2.append(optimistic_strategy(
                            env, optimistic_estimate=2))
    return (np.array(ret_eg1), 'greedy'), \
           (np.array(ret_eg2), 'epsilon 0.01'), \
           (np.array(ret_eg3), 'epsilon 0.1'), \
           (np.array(ret_egd1), 'decay 1e-2 -> 0'), \
           (np.array(ret_egd2), 'decay 1e-3 -> 0'), \
           (np.array(ret_os1), 'optimistic 1'), \
           (np.array(ret_os2), 'optimistic 2')

env = gym.make("BanditTwoArmedHighHighFixed-v0")
results = run_exploration_strategies_experiment(env)

env = gym.make("BanditTenArmedGaussian-v0")
results = run_exploration_strategies_experiment(env)
```

(1) Bring required imports

(2) Initialize lists to hold the results

(3) We run the experiments 100 times to average out the noise

(4) You should try different initial values and see for yourself how sensitive these are to them

(5) You could also try different bandit environments

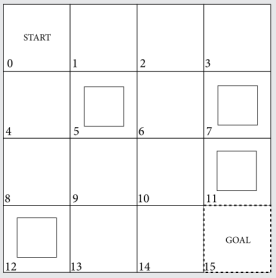
Learning to estimate policies

We are just one step away from diving into the full reinforcement learning problem. Let's bring back the sequential decision-making aspect of reinforcement learning we discussed in the previous chapter and let it mix for the first time with the model uncertainty issues that arise in the previous section. Remember that in the previous chapter we separated the policy iteration algorithm into a policy evaluation and policy improvement steps. Learning to evaluate policies, often referred to as the **prediction problem**, consists of approximating the true value functions, (V's and Q's) of any given policy. As you learned in the previous chapter, estimating policies is an important skill, because it ties easily to optimizing policies and thereby finding optimal policies. In this section, we'll also separate evaluation and improvement. We will start by creating agents that can estimate the state-value function of any given policy, and we will build on these agents in the next section to create agents that find optimal policies. We will first use the same Frozen Lake environment as before, but we will pretend we don't know the full dynamics of this environment; we won't make use of its MDP. In later sections, we will use a larger Frozen Lake environment with slightly different and unknown dynamics and create agents that find optimal policies through trial-and-error learning.

The Environment

Frozen Lake Reloaded 1/2

Recall the Frozen Lake environment introduced in chapter 2. This time, however, think of it as an unknown environment. Pretend you don't know the MDP that exists under-the-hood. We are only given in advanced the number of states in the MDP and actions available for each state. But, that's it. In fact, the agent doesn't even know that is a grid world, or what a grid world really is. All the agent is able to see is state ids (0, 4, 7) and rewards (0, 0, 1.) All the agent is able to do is pick one out of 4 available actions every timestep, look at the transitions and learn something.

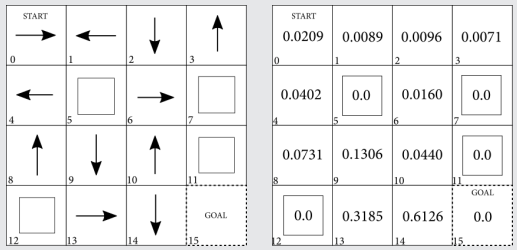


Same environment, but agent only sees state ids and rewards (agent knowns about terminal states):

Environment: You are in state 0, reward 0.
Agent: Ok, take action 2.
Environment: You are in state 4, reward 0.
Agent: Hmm, take action 1.
Environment: State 5. You're done, reward 0.

The Environment
Frozen Lake Reloaded 2/2

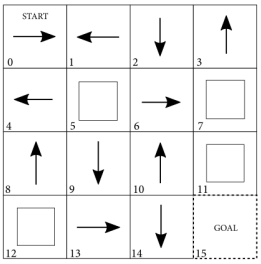
Also, recall the policy we were evaluating back then. This policy, which is not the optimal policy had a true state-value function we were able to find using the MDP and policy evaluation. This time, however, we won't be able to find the true state-value function of this policy. Instead, we will approximate this values. It's important to clarify that the methods that we will present in this chapter can converge to the true values, however, some of the requirements for garrantees of convergence to the optimal values include infinite data. Given this impractical requirement, and more so for later methods, we should be satisfied with few decimal places accuracy.



Learning to estimate using complete episodes

Given that we are being asked to estimate the state-value function of a given policy, the simplest approach that comes to mind is just to run several episodes with this policy and then calculate averages as we did with the bandit environments. This method is called **Monte-Carlo Prediction** (MC) and is based on Monte-Carlo simulations. Monte-Carlo prediction reduces the random noise of the environment with data—lots of data. Now, you probably remember the environment's dynamics and how stochastic the effects of actions are. Also, you know that the only non-zero reward of this environment is the goal state. Given these facts, propagating values around in this environment can be hard, and it will take many episodes. Unfortunately, the averages of a bunch of zeros are just zeros, so it won't be until we first hit the goal state that numbers will start changing.

Running given policy on Frozen Lake environment



Possible episodes:
(1) 0,4,0,4,8,4,5
(2) 0,1,5
(3) 0,1,0,4,4,8,9,13,13,14,15 +1
(4) 0,4,8,9,10,11
(5) 0,4,8,9,10,6,7
etc...

Values are states visited. We also have to track the reward, but given this environment only has a +1 on a single state we simplify notation by showing the +1 and not the 0s.

I Speak Python

Generate Episodes

```
def generate_episode(pi, env, max_steps):
    episodes = []
    state = env.reset()
    for t in range(max_steps):
        action = pi[state]
        new_state, reward, done, _ = env.step(action)
        episodes.append((state, action, reward, new_state, done))
        if done:
            break
        state = new_state
    return np.array(episodes, np.object)
```

(1) We are interacting now. We run policy pi on the given environment.

(2) First we reset the environment and read the initial state

(3) Simply pick the policy's action for a given state, execute and observe the full transition

(4) Store the full transition

(5) Get out when on a terminal state

(6) Store the new state for next timestep

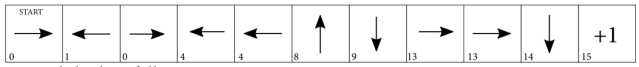
(7) Return the list of episodes as a numpy array

But let's talk about procedure now. You will first interact with the environment until you hit a terminal state. After generating each episode, you could estimate the state-value function $\mathcal{V}(s)$ on every episode simply by averaging the returns obtained from any given state. The return G, as was defined in the previous chapter, is the future discounted reward obtained from state s onwards. So, to calculate G_t, we would use the rewards at time t, $t+1$, $t+2$, up until the end of the episode $t+T$, and discount those rewards with an exponentially decaying gamma γ^0, γ^1, γ^2,..., γ^T. That means multiplying those together (rewards and discounted gammas) and then adding them up. That single value is equal to the current return estimate G_t, but we can keep track of many of these estimates and average those. So, we set $\mathcal{V}(s)$ equal to the average G's for all states.

You probably notice a potential discrepancy on how to deal with visiting states multiple times within the same episode. Should visiting the same state multiple times in an episode generate multiple G estimates or should we only use one? And if we only use one, which one should it be? For example, in episode 3, we visited state 0, 4 and 13 twice. Shouldn't this affect the return and count calculations? There are two common ways of dealing with multiple visits. Either we use the first visit and ignore subsequent visits to the same state, or we account for all visits to each state. We are currently using **first-visit Monte-Carlo prediction**, which only uses the first visit to each state for each episode when estimating the state-value function.

Taking as an example episode (3) previously shown:

(3) 0, 1, 0, 4, 4, 8, 9, 13, 13, 14, 15 +1



Rewards look as follows:

0 0 0 0 0 0 0 0 0 0 1

Discounted gamma:

1. 0.9 0.81 0.73 0.66 0.59 0.53 0.48 0.43 0.39 0.35

Multiplying and adding up would give us a return of:

0.35 for state 0

0.39 for state 1

0.48 for state 4

0.59 for state 8.... and so on. Note that this is only be a single per-state estimate.

Collecting multiple episodes would give us multiple estimates.

We would then average all estimates per state.

I ran first-visit Monte-Carlo prediction on the Frozen Lake environment to calculate the state-value function of policy π. As expected, the algorithm can approximate the value function well. As mentioned before, though, given that this is a **model-free reinforcement learning** method—meaning it doesn't make use of the environment's dynamics (MDP)—the algorithm takes much longer to estimate the policy because it needs to be able to access the non-zero reward under the policy π before values start propagating throughout.

SHOW ME THE MATH

Monte-Carlo Learning

(1) Collect episodes using policy π

(2) Calculate the return at timestep t by adding up the discounted future rewards

$$S_1, A_1, R_2, ..., S_k \sim \pi$$

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

(3) Calculate the state-value function as the expectation of all returns for state s

(4) As an implementation detail, remember we can calculate means incrementally, so we can do the following for the same update

$$mc\_target = G_t$$

$$mc\_error = mc\_target - V(S_t)$$

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(mc\_error)$$

$$V(S_t) = V(S_t) + \alpha(mc\_error)$$

(5) Additionally, we can use a learning rate for non-stationary environment

(6) Final mc update equation would look as follows

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t))$$

I Speak Python

Monte-Carlo Prediction

```
def mc_prediction(pi, env, gamma=0.9, initial_alpha=0.2,
  alpha_decay_rate=1e-3, min_alpha=0.0, n_episodes=10000,
  max_steps=500):
    nS = env.observation_space.n
    discounts = np.logspace(0, max_steps, \
                        num=max_steps, base=gamma, endpoint=False)

    V = np.zeros(nS)
    V_track = np.zeros((n_episodes, nS))

    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                np.exp(-alpha_decay_rate * t), min_alpha)
        episode = generate_episode(pi,
                                   env, max_steps)
        return_visited = np.zeros(nS, dtype=np.bool)
        for step_idx, (state, _, reward, _, _) in \
                                        enumerate(episode):
            if return_visited[state]:
                continue
            return_visited[state] = True
            seq_len = len(episode[step_idx:])
            G = np.sum(discounts[:seq_len] * \
                                episode[step_idx:, 2])
            V[state] = V[state] + alpha * (G - V[state])
        V_track[t] = V
    return V.copy(), V_track

V, V_track = mc_prediction(pi, env)
V
```

(1) This logspace function allows us to calculate an array of exponential discounts

(2) We initialize the state-value function V to zero

(3) We start interacting for n episodes

(4) We use an exponentially decaying learning rate

(5) generate a complete episode

(6) Create an array to keep track of states already visited

(7) For each timestep in the episode

(8) Keep track of the visits, and only process the first-visit to each state per episode

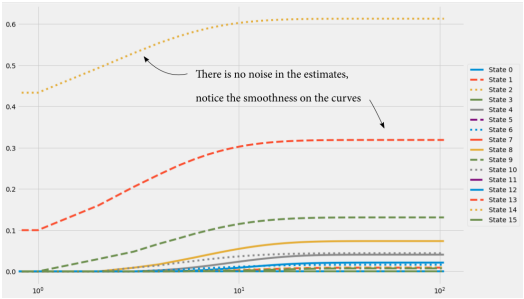
(9) This handy variable let's us access the length of the current sub-sequence of the full episode

(10) This extracts an array with the rewards of the sub-sequence

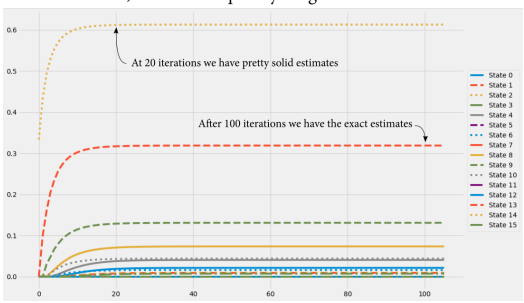
(11) We can calculate the return of the sub-sequence by multiplying the returns and the discounts arrays

(12) Adjust the state-value function a little towards the monte-carlo error

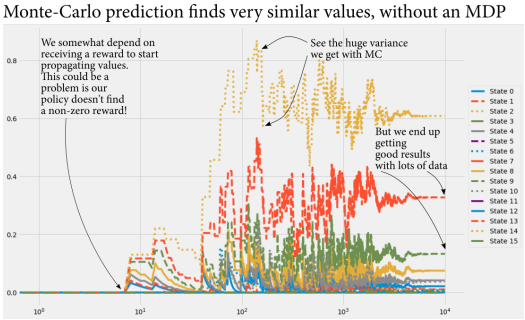
This is the true state-value function calculated using the MDP



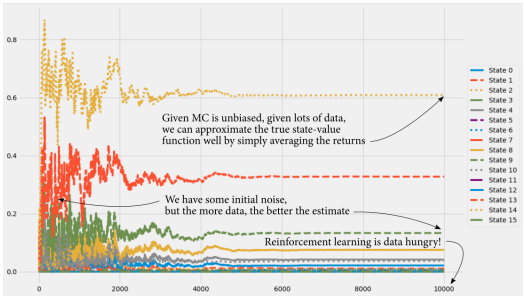
Also, notice how quickly we get to true values



Monte-Carlo prediction finds very similar values, without an MDP



MC is an unbiased estimate of the true state-value function



Learning to estimate one timestep at a time

Monte-Carlo prediction does the job well. It is an algorithm without *bias* because it uses the true rewards received during complete episodes and then averages those true values. But there is a way we could integrate the reward we receive at each timestep without having to wait until the end of each episode before we calculate true values. It might not be too obvious with the Frozen Lake environment we are currently exploring, because the only non-zero reward in the environment is at the goal state and that reward can only occur at the end of the episode. But imagine that same environment with a different reward function. Let's say we get a reward of +1 for terminating an episode at the goal state, -1 for terminating on a hole, and -0.01 for every timestep, for every move. In this case, MC will first interact and wait until the end of the episode has been reached before the rewards get integrated into the current state-value function estimates. Why not use each -0.01 as soon as we receive and improve our estimates?

Well, there is another class of model-free reinforcement learning methods, called **Temporal-Difference** (TD) methods. These methods, unlike MC, learn from incomplete episodes by using the real return of one timestep, that is the immediate reward, and an estimate of the returns we would get from that moment onwards, which is the state-value function of the next state after seeing a full transition. Remember, a transition is a tuple *(s, a, r, s')* that represents the initial state, the action taken, the reward received, and the state we transitioned to. At first, the state-value function estimate of state *s'* is a random value, a guess. But as time progresses and we integrate more and more experienced rewards, the values of all states will get better and better, and so will this guess. Using the estimated state-value function of the next state as part of our current state calculations is called bootstrapping and is a way of updating a guess towards an improved guess, because we are adding one step of the real returns every time.

Show Me The Math
Temporal-Difference Learning

(1) Monte-Carlo target is the true, complete return *G*, while the td target uses a bias sample reward and a estimate of the next state, which by the way, is a guess.

$$td\_target = R_{t+1} + \gamma V(S_{t+1})$$

$$td\_error = td\_target - V(S_t)$$

(2) The td error is the difference between the current estimate and the td target

(3) The new state-value function estimate is then moved a little towards the error

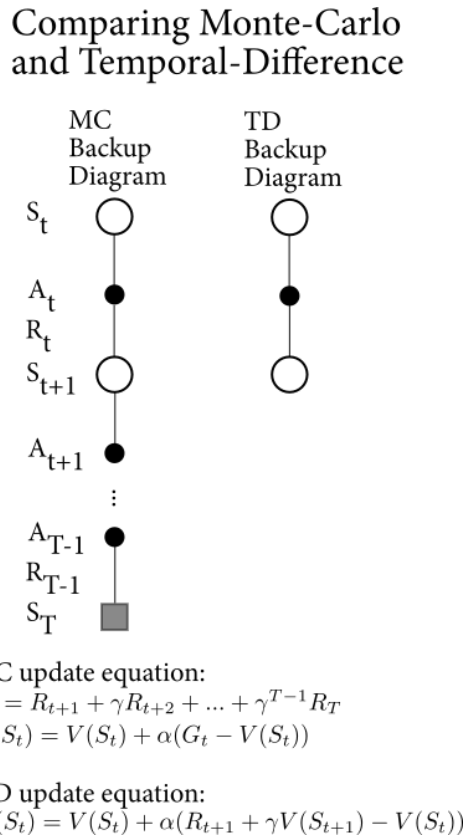
$$V(S_t) = V(S_t) + \alpha(td\_error)$$

(4) The full equation looks as follows

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

Still, this way of updating is *biased*, because we use a single transition, which carries noise, as the basis for our updates. But it works well, too. This is something like what we did with dynamic programming methods. We estimated the state-value function of a state by bootstrapping on the estimated state-value function of future states. In dynamic programming methods, we used expectations across all transitions, so there was no bias. But in TD learning, we only use a single sampled reward and transition at a time, with an estimate that is somewhat incorrect. For this reason, the TD target

is said to be a biased estimate of the true state-value function. On the other hand, the TD target is much lower *variance* than the return *G* we use in Monte-Carlo updates. The return *G* depends on many random variables within a single episode: you take multiple actions, transition multiple times, and receive multiple rewards. That adds high variance because the randomness accumulates within the same episode and estimates could wildly differ from one episode to another. The TD target depends only on a single action, a single transition, a single reward, and a single next-state estimate.



TD methods can be more efficient than MC methods on Markov environments like the Frozen Lake environment. But MC methods can be more effective than TD in non-Markovian environments, which are often the case in real-world problems. Temporal-difference methods bootstrap, just like Dynamic Programming methods do, so they can run in environments that don't have terminal states or environments in which terminal states are hard to reach. Our Monte-Carlo method does not, unfortunately, bootstrap. This means it can only use updates after reaching a final state, and it is virtually useless otherwise. MC and TD methods both learn from interaction by sampling the environment, so you don't need to know the full dynamics of the environment ahead of time. DP methods can't do this.

This is all to show that the MC vs. TD debate is mostly a matter of the engineering tradeoff you'll have to face between *bias* and *variance*, *episodic* or *continuous* tasks. Both methods have similar convergence properties, both are used in practice, and both have their place in the reinforcement learning landscape. I ran TD learning on the Frozen Lake environment to obtain an estimate of the state-value function for policy π, just as I did with MC. TD learning also finds a solid approximate state-value function, despite the bias.

I Speak Python

Temporal-Difference Prediction

```
def td(pi, env, gamma=0.9, initial_alpha=0.2,
       alpha_decay_rate=1e-3, min_alpha=0.0, n_episodes=10000):

    nS = env.observation_space.n
    V = np.zeros(nS)
    V_track = np.zeros((n_episodes, nS))

    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                np.exp(-alpha_decay_rate * t), min_alpha)

        state, done = env.reset(), False
        while not done:
            action = pi[state]
            new_state, reward, done, _ = env.step(action)
            td_error = reward + gamma * V[new_state] - V[state]
            V[state] = V[state] + alpha * td_error
            state = new_state
        V_track[t] = V
    return V.copy(), V_track

V, V_track = td(pi, env)
V
```

(1) Initialize all variables as before

(2) We also run for n episodes

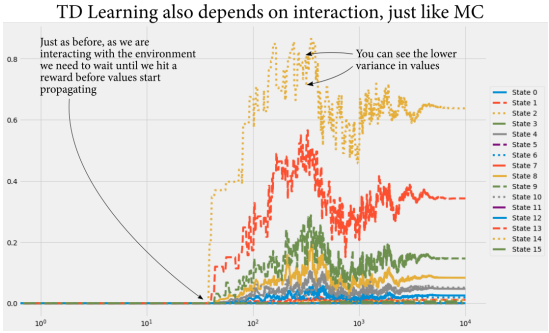
(3) But this time we will update the state-value function online.

(4) Calculate the td error as in the update equations

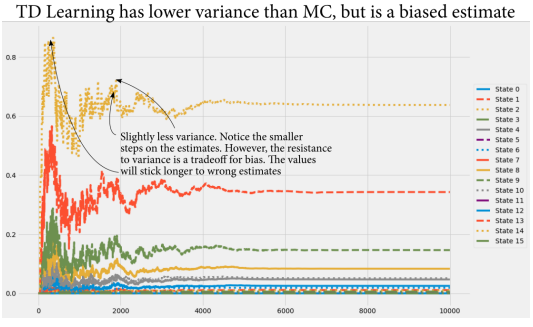
(5) Calculate the state-value function

(6) Make sure you update your state variable so we update the right state inthe next iterations

TD Learning also depends on interaction, just like MC



TD Learning has lower variance than MC, but is a biased estimate



Learning to optimize behavior

Now that you know how to estimate the state-value function of any given policy, let's think of ways our agents can optimize policies so that they could learn, from a random policy, to behave optimally through trial-and-error. Learning to behave optimally through interaction in a sequential decision-making problem is, finally, the full reinforcement learning problem: how can an agent learn to behave optimally purely from interaction with the environment. The agent doesn't know anything about the dynamics of the environment, and we have no prior policy that we are trying to evaluate. The agent is dropped in an environment and starts interacting—taking actions and observing states and rewards. How can this agent improve and ultimately learn optimal behavior?

In the last section you learned how to *estimate* the value function of a policy for an unknown environment (no MDP). Now we're going to look at how to *optimize* the policy of a value function for an unknown environment. Optimizing policies is referred to as the **control problem**. I will use the same idea you learned in the pre-

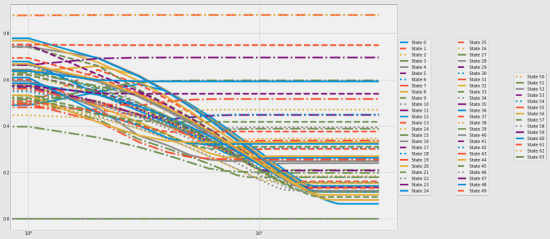
The Environment

Frozen Lake Revolutions

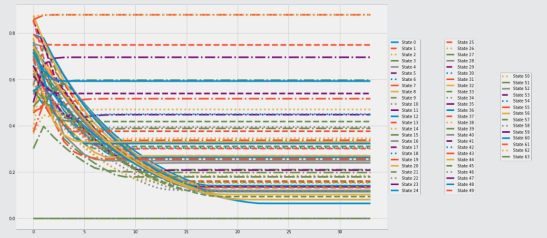
For the following experiments we will use a different version of the Frozen Lake environment. The environment has gotten bigger, its transition probability and its reward function are unknow. We still have 4 action available, but this time we have 64 states. We know need to create agents that learn to behavior by trial-and-error learning.

I ran Policy Iteration on the MDP and I'm showing the optimal state-value function in the graphs on the right. This as a reference of the difference between planning and learning through interaction.

Frozen Lake environment with 64 states

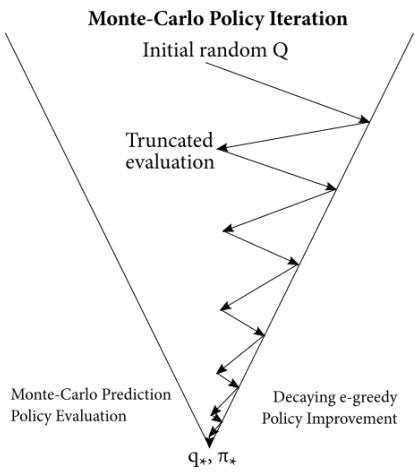


These are the optimal state-value function plots for reference



vious chapter when we discussed policy iteration: First, we'll evaluate a policy and obtain its state-value function, though this time we will obtain only approximations. We will then improve the evaluated policy, but this time we won't be able to simply act greedily because we can't look ahead and know the maximum values. Instead, we will have to use some of the exploration techniques discussed earlier to balance the exploration-exploitation tradeoff. By iterating through those two processes, *evaluation*, and *improvement*, we will get closer and closer to the optimal value function and optimal policy.

Learning to optimize using complete episodes



The policy evaluation algorithm allows us to obtain a state-value function from a given policy by using an MDP. In the previous section, you learned how to calculate the state-value function of a given policy without the need for an MDP. Let's now consider simply replacing the policy evaluation step (of policy iteration) with the Monte-Carlo prediction method you just learned. You have two algorithms that return the same thing: a state-value function. Would this make an algorithm that helps us find optimal policies solely through interaction? No. There are two issues with this approach. First, remember that the policy improvement method, which is the second step of policy iteration, used the state-value function *and the transition dynamics* of the MDP to do a lookahead of the transitions and act greedily using the current estimates. We no longer have the MDP, and we can no longer peek into the future. So, from now on, we will be using the *action-value function* instead of the state-value function for value estimation. The action-value function allows us to do control when we don't have the dynamics of the environment because of the cache values and separate transition information on a per-action basis. So instead of acting greedily over the state-value function, we can simply select the action with the highest value.

The second issue with replacing the policy evaluation step as mentioned before prevents us from just acting greedily when doing our policy improvement step. Again,

in dynamic programming methods, we used the transition function to calculate the expectation across all possible next states. But, we can no longer act greedily because our one-step values are no longer perfect. We need to explore, so that our values get better over time. Luckily, you learned multiple exploration strategies in the bandit environments from the first section of this chapter. You could choose epsilon-greedy or decaying epsilon-greedy, and that's it—you would have a complete, **model-free reinforcement learning** algorithm in which we evaluate a given policy by using *Monte-Carlo prediction* and we improve it by using *decaying e-greedy action selection strategy*. Also, just as with value iteration, which is an instance of truncated policy evaluation, we can truncate the Monte-Carlo prediction step. So, instead of waiting for it to terminate, you can just iterate between a single MC prediction step and a decaying e-greedy action selection improvement step.

On-policy Monte-Carlo control provides a good opportunity to introduce a practical example of a stochastic policy in action. Recall that stochastic policies hold probabilities of selecting an action per-state. We'll store a table of 2-dimensions (states by actions) and, on each entry, set the probability of taking that action, given that state. Obviously, we must ensure the probabilities across the actions for every state, that is each row, add up to one. We are going to use this kind of policy as a way of doing the decaying e-greedy strategy since e-soft policies simply move the exploration strategy into policy space. But, they work in a similar way and give the similar results.

I Speak Python
E-Soft Policy 1/2

```
class ESoftPolicy:
    def __init__(self, nS, nA, epsilon=0.2):
        self.nA, self.nS = nA, nS
        self.epsilon = epsilon

        self.probs = np.random.dirichlet(np.ones(self.nA), \
                                         size=self.nS)
        for state in range(self.nS):
            action = np.random.randint(self.nA)
            self.readjust_probs(state, action, epsilon)

    def __getitem__(self, state):
        return np.random.choice(self.nA, p=self.probs[state])
```

(1) This is a handy function that creates an array of dimensions (nstate, naction) and adds up to 1 across the actions axis

(2) We then adjust the probabilities to make the e-soft policy

I Speak Python

E-Soft Policy 2/2

```
    def readjust_probs(self, state, best_action, epsilon):
        self.epsilon = epsilon

        prob_best = 1 - self.epsilon + self.epsilon/self.nA
        prob_other = self.epsilon/self.nA
        new_probs = [prob_other,] * (self.nA - 1)
        new_probs.insert(best_action, prob_best)
        self.probs[state] = new_probs
```

(3) The probability of selecting the best action is as follows

(4) Update the probabilities with the new epsilon

I Speak Python

Monte-Carlo Control 1/2

```
def mc_control(env, gamma=0.9, initial_epsilon=1.0,
   epsilon_decay_rate=1e-5, min_epsilon=0.01,
   initial_alpha=0.5, alpha_decay_rate=1e-3, min_alpha=0.001,
   max_steps=500, n_episodes=100000):

    nS, nA = env.observation_space.n, env.action_space.n
    discounts = np.logspace(0, max_steps, \
                        num=max_steps, base=gamma, endpoint=False)
    Q = np.zeros((nS, nA))
    Q_track = np.zeros((n_episodes, nS, nA))

    pi = ESoftPolicy(nS, nA, initial_epsilon)

    for t in tqdm(range(n_episodes)):
        epsilon = max(initial_epsilon * \
                      np.exp(-epsilon_decay_rate * t), min_epsilon)
        alpha = max(initial_alpha * \
                        np.exp(-alpha_decay_rate * t), min_alpha)
        episode = generate_episode(pi, env, max_steps)
        return_visited = np.zeros((nS, nA), dtype=bool)
```

(1) We initialize all the same variables as in Monte-Carlo prediction

(2) We create an e-soft policy to garrantee exploration of all state-action pairs

(3) We iterate for n episodes, note that this could be improve for faster convergence

(4) Both epsilon and alpha have exponential decaying values

(5) We generate a full episode... This is Monte-Carlo

(6) Algorithm continues in the next page

I Speak Python

Monte-Carlo Control 2/2

(7) Then iterate over all states in the episode

(8) We keep track of the states visited in the current episode, First-Visit MC

(9) Handy variable to let us access the discounts

(10) Return is the same as before, a simple array sumation of the product of discounts and rewards

(11) We calculate the mean incrementally, but notice how we are using action-value functions instead of the state-value function in MC prediction

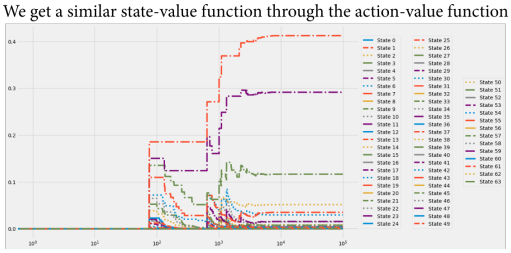
(12) We now recalculate the probabilities of the e-soft policy to ensure a decaying e-greedy exploration strategy

(13) Here we readjust the probabilities of the stochastic policy given the best action for any given state and the decayed epsilon

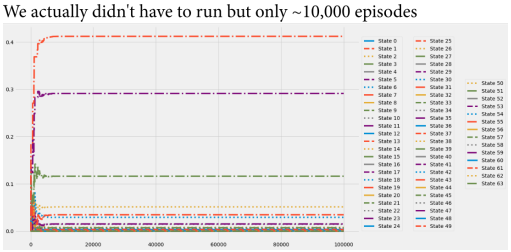
(14) At the end of the process, extract a deterministic policy and a state-value function from the Q values we terminated with, this just for comparison

```
        for step_idx, (state, action, reward, _, _) \
                                                in enumerate(episode):
            if return_visited[state][action]:
                continue
            return_visited[state][action] = True

            seq_len = len(episode[step_idx:])
            G = np.sum(discounts[:seq_len] * \
                                                episode[step_idx:, 2])
            Q[state][action] = Q[state][action] + alpha * \
                                                (G - Q[state][action])
        Q_track[t] = Q

        _, uidxs = np.unique(episode[:,0], return_index=True)
        for idx in uidxs:
            state, action, reward, _, _ = episode[idx]
            best_action = np.argmax(Q[state])
            pi.readjust_probs(state, best_action, epsilon)
    pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
    V = np.max(Q, axis=1)
    return Q, V, pi, Q_track


Q_best, V_best, pi_best, Q_track = mc_control(env)
Q_best, V_best, pi_best
```

We get a similar state-value function through the action-value function



We actually didn't have to run but only ~10,000 episodes



Learning to optimize one timestep at a time

Using action selection strategies like decaying e-greedy inside the learning algorithm is a more common approach to dealing with the exploration-exploitation tradeoff than moving exploration into policy space like in e-soft policies, so we will leave e-soft policies behind for now and use a decaying e-greedy action selection strategy. We can go from Monte-Carlo control to Temporal-Difference control by replacing the MC prediction method with TD prediction—this creates the popular **SARSA** algorithm. Using action selection strategies inside the learning algorithm is a more common approach to dealing with the exploration-exploitation tradeoff than using e-soft policies.

I Speak Python
SARSA 1/2

```
def sarsa(env, gamma = 0.9, initial_alpha = 1.0,
      alpha_decay_rate = 5e-4, min_alpha = 1e-4,
      initial_epsilon = 1., epsilon_decay_rate = 1e-5,
      min_epsilon = 0.01, n_episodes=100000):
    nS, nA = env.observation_space.n, env.action_space.n
    Q = np.zeros((nS, nA))
    Q_track = np.zeros((n_episodes, nS, nA))

    select_action = lambda state, Q, epsilon: \
        np.argmax(Q[state]) \
        if np.random.random() > epsilon \
        else np.random.randint(nA)
```

(1) We initialize all variable just as before

(2) Now, we are using a decaying epsilon-greedy action selection strategy. Here we have the e-greedy in a lambda function, we decay epsilon below

(7) Algorithm continues...

I SPEAK PYTHON

SARSA 2/2

```
    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                np.exp(-alpha_decay_rate * t), min_alpha)
        epsilon = max(initial_epsilon * \
                  np.exp(-epsilon_decay_rate * t), min_epsilon)
        state, done = env.reset(), False
        action = select_action(state, Q, epsilon)
        while not done:
            new_state, reward, done, _ = env.step(action)
            new_action = select_action(new_state, Q, epsilon)
            if done:
                Q[new_state] = 0.
                Q_est = reward
            else:
                Q_est = reward + gamma * Q[new_state][new_action]
            td_error = (Q_est - Q[state][action])
            Q[state][action] += alpha * td_error
            state, action = new_state, new_action
        Q_track[t] = Q
    pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
    V = np.max(Q, axis=1)
    return Q, V, pi, Q_track

Q_best, V_best, pi_best, Q_track = sarsa(env)
Q_best, V_best, pi_best
```

(3) Iterate over n episodes

(4) Here we decay both alpha and epsilon exponential. At a different schedule, but configurable too

(5) We initialize the environment and get the initial state

(6) We select the best action according to the decaying e-greedy strategy

(8) We send the action to the environment and select the action to be taken next

(9) Terminal state Q functions should always be zero. Here we check if we landed on a terminal state and zero out the Q function for the next state

(10) Calculate the td target using the next Q value

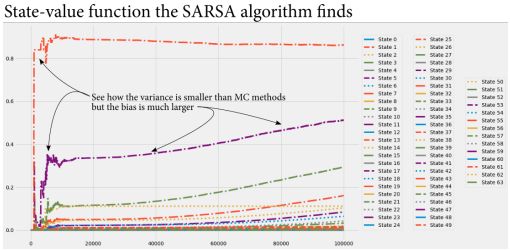
(11) Calculate the td error

(12) Adjust the Q value for that state, action pair a bit toward the error

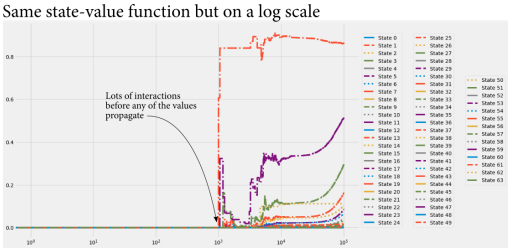
(13) Update the state and action for next iteration

(14) Extract a deterministic policy and state-value function only for comparison

State-value function the SARSA algorithm finds



Same state-value function but on a log scale



Decoupling behavior from learning

The SARSA algorithm was a sort of "learning on the job," meaning we optimized the policy we were using as we used it. This is called **on-policy learning**. On-policy learning is great—we learn from our own mistakes. But in on-policy learning, we only learn from our own current mistakes. What if we want to learn from experiences collected with older policies—our previous mistakes? What if we want to learn from the mistakes of others? That would amplify our learning capabilities as we would be increasing the amount of sources we could learn from. **Off-policy learning** decouples the behavior from the learning so that our agents can learn from many sources. For this, we use two policies: a *behavior policy*, used to sample experience, and a *target policy*, used to optimize behavior.

I Speak Python

Q-Learning 1/2

```
def q_learning(env, gamma = 0.9, initial_alpha = 0.5,
       alpha_decay_rate = 1e-3, min_alpha = 0.001,
       initial_epsilon = 1., epsilon_decay_rate = 1e-5,
       min_epsilon = 0.01, n_episodes=100000):
    nS, nA = env.observation_space.n, env.action_space.n
    Q = np.zeros((nS, nA))
    Q_track = np.zeros((n_episodes, nS, nA))
    select_action = lambda state, Q, epsilon: \
        np.argmax(Q[state]) \
        if np.random.random() > epsilon \
        else np.random.randint(nA)
```

(1) We initialize all variable just as before, in fact most of the algorithm is like SARSA

(2) Same action selection strategy

(3) Same way of going through the all episodes

(4) Algorithm continues...

I Speak Python

Q-Learning 2/2

```
    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                np.exp(-alpha_decay_rate * t), min_alpha)
        epsilon = max(initial_epsilon * \
                np.exp(-epsilon_decay_rate * t), min_epsilon)
        state, done = env.reset(), False
```

(5) But now, we don't select the action just yet, we will update towards the max value and decouple behavior from learning

```
        while not done:
            action = select_action(state, Q, epsilon)
            new_state, reward, done, _ = env.step(action)
```

(6) Select the action and observe a single transition

```
            if done:
                Q[new_state] = 0.
                Q_est = reward
            else:
                Q_est = reward + gamma * Q[new_state].max()
```

(7) We estimate the q target the same way we did before, carying about the final state. This is important for convergence, especially is we don't initialize Q to zero

(8) See how we take a max to calculate our target. This is the key off-policy line. In SARSA we used the Q value of the state we actually took. We won't necessarily take the max Q next iteration because of the decaying e-greedy action select, it could be a random action!

```
            td_error = Q_est - Q[state][action]
            Q[state][action] += alpha * (td_error)

            state = new_state
        Q_track[t] = Q
```

(9) Calculate the error

(10) Adjust the Q value a bit closer to the error

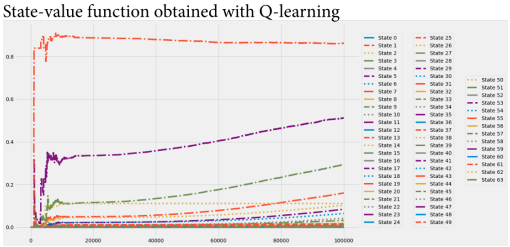
(11) Update the state variable for next timestep

```
    pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
    V = np.max(Q, axis=1)

    return Q, V, pi, Q_track
```

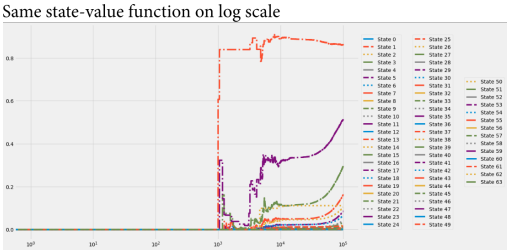
(12) Extract the greedy policy and state-value function obtained from the full algorithm

```
Q_best, V_best, pi_best, Q_track = q_learning(env)
Q_best, V_best, pi_best
```

State-value function obtained with Q-learning



Same state-value function on log scale



The off-policy temporal-difference control algorithm is called **Q-learning**, and it is probably the most popular version of "tabular" reinforcement learning. It is also the base of perhaps the most popular deep reinforcement learning algorithm you will learn about in this book, deep Q-networks (DQN). It is important to note that the MC control algorithm we studied before is an on-policy method. There is also an off-policy version of this MC control algorithm, just like there are Q-learning and SARSA, but the off-policy version of MC control is just very ineffective, and its convergence properties are terrible making the algorithm impractical.

Stabilizing learning

Q-learning often over-estimates the value function. At every iteration, we are taking the *maximum over the estimates* of the value function in an attempt to *estimate the maximum value finction*. This is not only an inaccurate way of estimate the maximum value function, but also a bigger problem given the estimates are often biased. So, the use of *maximum of biased estimates* as the *estimate of maximum value* is a problem known as **maximization bias**. One way of dealing with this is to keep two different Q tables. At each timestep, we pick one of them to determine the maximizing action, and we use the other one to calculate the estimate. We would use both to select actions by using the average of both *Q(s)* estimates. This algorithm is called **double Q-learning**, and it is the basis of another very popular deep reinforcement learning algorithm you'll learn about later called double deep Q-networks (DDQN).

I Speak Python

Double Q-Learning 1/3

```
def double_q_learning(env, gamma = 0.9, initial_alpha = 0.5,
      alpha_decay_rate = 1e-3, min_alpha = 0.001,
      initial_epsilon = 1., epsilon_decay_rate = 1e-5,
      min_epsilon = 0.01, n_episodes=200000):
```

(1) Algorithm continues...

I Speak Python

Double Q-Learning 2/3

```
nS, nA = env.observation_space.n, env.action_space.n
(2) Setup two Q tables for double learning
Q1 = np.zeros((nS, nA))
Q2 = np.zeros((nS, nA))

Q_track1 = np.zeros((n_episodes, nS, nA))
Q_track2 = np.zeros((n_episodes, nS, nA))
(3) Same action selection as in previous algorithms
select_action = lambda state, Q, epsilon: \
    np.argmax(Q[state]) \
    if np.random.random() > epsilon \
    else np.random.randint(nA)
                                  (4) Iterate over n episodes
for t in tqdm(range(n_episodes)):

    alpha = max(initial_alpha * \
            np.exp(-alpha_decay_rate * t), min_alpha)
    (5) Both, alpha and epsilon, use an exponential decaying schedule
    epsilon = max(initial_epsilon * \
            np.exp(-epsilon_decay_rate * t), min_epsilon)
    (6) We only get the initial state, just like in Q-learning
    state, done = env.reset(), False

    while not done:
        action = select_action(state, Q1+Q2, epsilon)
        new_state, reward, done, _ = env.step(action)
(7) Select the action and observe a full transition from the environment
        if done:
            Q1[new_state] = 0.
            Q2[new_state] = 0.
                                  (8) Clear out the Q values of
                                  terminal states
    (9) Flip a coin and update one or the other Q table only
        if np.random.randint(2):
            argmax_Q1 = np.argmax(Q1[new_state])
            Q2_est = reward if done else \
                reward + gamma * Q2[new_state][argmax_Q1]
            Q1[state][action] += alpha * \
                 (Q2_est - Q1[state][action])
(10) We update the
current estimate
towards the best
action of the other
Q table.
                                  (11) Algorithm continues...
```

I Speak Python

Double Q-Learning 3/3

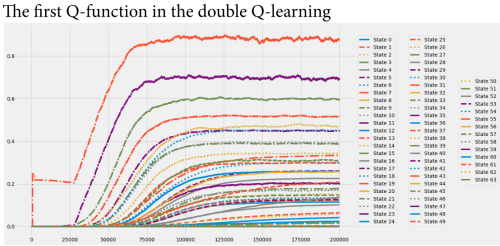
```
            else:
                argmax_Q2 = np.argmax(Q2[new_state])
                Q1_est = reward if done else \
                     reward + gamma * Q1[new_state][argmax_Q2]
                Q2[state][action] += alpha * \
                     (Q1_est - Q2[state][action])
            state = new_state
        Q_track1[t] = Q1
        Q_track2[t] = Q2

    Q = (Q1 + Q2)/2.
    pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
    V = np.max(Q, axis=1)

    return Q, V, pi, Q_track1, Q_track2

Q_best, V_best, pi_best, Q_track1, Q_track2 = double_q_learn-
ing(env)
Q_best, V_best, pi_best
```

(12) Update the state for next iteration

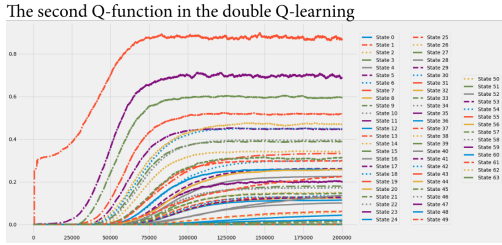
(13) Calculate an average of the action-value function

(14) And use that to get the policy and state-value function

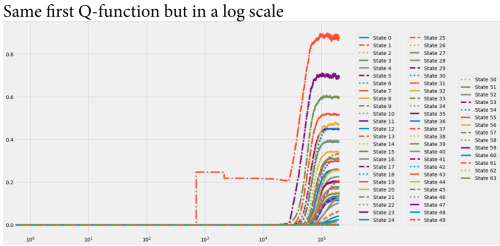
The first Q-function in the double Q-learning



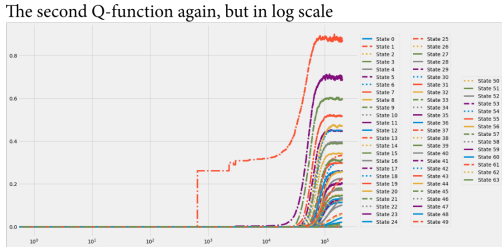
The second Q-function in the double Q-learning



Same first Q-function but in a log scale



The second Q-function again, but in log scale



Summary

In this chapter, you learned the importance of balancing exploration and exploitation by solving a couple of bandit environments, and you learned that you need both to have an optimal action selection strategy. You learned there are three major families of action selection strategies: random exploration, optimism in the face of uncertainty, and information state space. You learned two ways to solve the prediction problem when the MDP is unknown: the Monte-Carlo prediction method, which waits until all rewards are available at the end of the episode to estimate the state-value function, and the Temporal-Difference prediction method, which can bootstrap a guess from a guess.

You then applied that knowledge to the full reinforcement learning problem on a new and unknown Frozen Lake environment. You learned that you couldn't simply replace prediction methods such as MC and TD with the policy evaluation method in policy iteration, because you must first swap state-value functions for action-value functions and then use exploration strategies instead of greedy policy improvement.

You learned to find optimal policies, either by using the full returns in Monte-Carlo methods or by using the differences in estimates one timestep at a time, as in Temporal-Difference methods. You learned about both prediction and control problems, which are estimation and optimization, respectively. You learned that on-policy learning is learning to optimize a policy as we used it and it couples behavior and optimization into the same policy, while off-policy learning decouples learning from behavior, allowing your agent to learn from different sources. Finally, you learned about ways of improving Q-learning and its maximization bias, through the use of double learning.

In the next chapter, we will extend and unify all the methods that you have learned so far. We will try to get the best of both the TD learning and MC learning worlds, and we will also try to unify model-free reinforcement learning methods and dynamic programming methods to create more effective and more efficient reinforcement learning algorithms: this is referred to as model-based reinforcement learning.

By now you:

- Can create agents that learn to behave optimally solely through interaction in any discrete (or discretized), unknown environment.
- Can solve most tabular reinforcement learning problems.

- Know how to implement Monte-Carlo and Temporal-Difference methods and recognize their advantages and disadvantages and their different tradeoffs.
- Know how to improve fundamental methods such as Q-learning using a technique known as double learning.

More Effective and Efficient Reinforcement Learning

4

IN THIS CHAPTER

You'll learn how to program into your agents more effective exploration strategies by collecting targeted samples from the environment.

You'll learn how to make your agents more efficient at learning from the samples they collect.

You'll learn to integrate algorithms with extreme backup depths in a unifying algorithm that takes the best of both worlds.

You'll learn to integrate planning and learning creating architectures that often have less sample complexity and are more robust to non-stationary environments.

"Efficiency is doing things right; effectiveness is doing the right things."

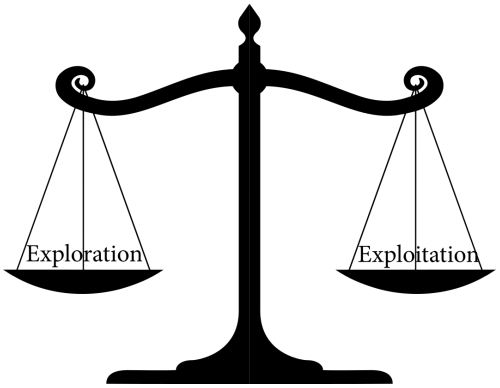
— Peter Drucker
Founder of modern Management and
Presidential Medal of Freedom recepient

You've learned the two fundamental tradeoffs in reinforcement learning: balancing the need for immediate rewards with the sacrifice that long-term goals require for maximum profit, and balancing the need for gathering the knowledge required to be effective and the practical use of this knowledge to achieve goals. What you've learned so far in this book can go a long way in creating intelligent agents. With what you already know, you can virtually solve any problem in reinforcement learning, that is *tabular* reinforcement learning as opposed to deep reinforcement learning.

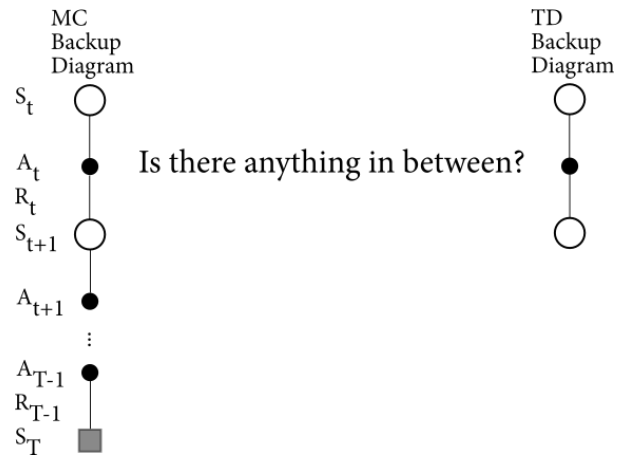
But, there is still room for improvement. There always is. And I don't want you to go on thinking that this is all there is about reinforcement learning principles. There is a lot more to cover. So, I will dedicate one more chapter to reinforcement learning principles to let you see the potential and open the door of your imagination which will become particularly useful in later chapter.

There are three areas of improvements I will touch on before moving on to the world of deep reinforcement learning. First, recall the beginning of the last chapter when I introduced the exploration vs. exploitation tradeoff on the bandit's problem. I presented a handful of methods that solve this environment. However, I hope you were unsatisfied with the strategies we implemented. I don't mean to give the wrong impression here, those strategies are not only good enough, but they also are the most commonly used and most effective for their complexity. Recall the epsilon-greedy strategy, for example. You "flip a coin" and based on that outcome you decide either to select what you consider the best or some randomly selected action. We, as humans, don't explore options randomly. At least not the most intelligent humans. We don't go about flipping coins and leaving our destiny into the random-chance gods' hands. It just doesn't work that way. We have more systematic approaches to dealing with this problem. In the first part of the current chapter, we will explore more complex ways of balancing exploration vs. exploitation which is such an important tradeoff in reinforcement learning.

You will learn more effective ways for dealing with the exploration vs. exploitation tradeoff



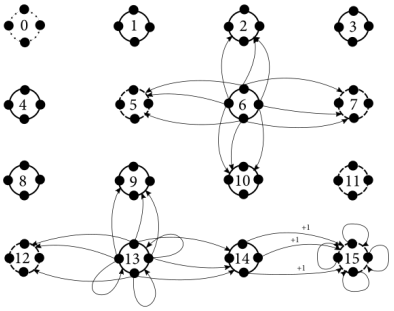
Second, recall the last two parts of the previous chapter in which I showed you the two extremes in the backup depth spectrum. In the one hand, you have Monte-Carlo prediction and control methods which have deep backups. That is, these algorithms wait until the end of an episode and then use the true rewards obtained along the path to calculate the returns and value functions estimates. In the other hand, you have Temporal-Difference prediction and control methods which use shallow backups. TD methods change value functions one step at a time. And instead of using the true rewards obtained along the full trajectory, they use a single immediate reward and a guess of what the value function of the landing state is. It may sound scary that these methods make their estimates by leveraging or *bootstrapping* on their current estimates of states. After all, the current estimates are *guesses*. So, they make guesses from guesses each time they encounter a state. But, the magic happens by injecting one sample of the true reward on each transition. As the time goes on, you will inject much more "trueness" into your estimates, and most of the time, the guesses get better with each timestep. So, which of two backup styles is better? Should it be one-step estimates, like TD methods, or the full trajectory, like MC methods? Recall that the comparison in the last chapter showed advantages and disadvantages of both approaches. But, why not use some other number, say 2, 3 or 7 steps? Would any of those be better? Or, can we combine and weight multiple steps on a single update? In the second part of this chapter, we will unify TD and MC and show a family of algorithms that use a weighted combination of the many timesteps your agents experience in a single episode. These algorithms retain most of the advantages of the previous approaches and often perform better than either of them. It's the best of both worlds.



Lastly, recall how in the second chapter we used Markov decision processes (MDPs) to solve environments without the need for interaction. Such *planning* methods are very useful when you have a full description of the environment you are trying to solve, but those methods don't work in problems in which you don't know how the

environment will react to your agents' actions, which is most of the real-world problem you'll face. However, if we keep track of the states we visit, the actions we take, the states we are sent to and the rewards we receive, we can try to model the environment as we interact with it. We can, then, interleave planning methods with model-free methods to improve our estimates without having to interact with the environment. Although more computationally expensive than the algorithms you have learned so far, these methods can be very useful when interacting with the environment is costly or challenging. Think about robotics, for example, losing hardware due to a bad estimate is arguably worse than letting a computer run for longer. In the last part of this chapter, we will explore these kinds of methods.

Can an agent build an MDP with data from interactions?



Would that help us at all?

Strategic exploration

Alright, imagine you write a reinforcement learning agent to learn driving a car. You decide to implement an epsilon-greedy exploration strategy. You flash your agent into the car's computer, you start the car, and your car starts exploring; it will flip a coin and decide to drive on the other side of the road to see what rewards are over there. I hope this example helps illustrate the need for better exploration strategies.

In the previous chapter, I showed you different approaches to solving the exploration vs. exploitation dilemma. However, you surely felt that we could do better than some of those techniques. In this section, I want to present a few more exploration strategies that are an iteration of improvements over the ones we previously discussed. You'll notice how, even though still not perfect, the action-selection strategies presented in this chapter are more *directed* and *strategic*, which is already an improvement.

Now, before I proceed, however, I want to reiterate that epsilon-greedy is still the most popular exploration strategy in use today, perhaps because it performs well, perhaps because of its simplicity. Maybe because most reinforcement learning applications today live inside a computer and there are very few safety concerns with the virtual world. It's important for you to think hard about this problem. Balancing

the exploration vs. exploitation tradeoff is central to human intelligence and reinforcement learning, and I'm certain work in this area will have a big impact in the field of reinforcement learning and many other fields interested in this fundamental tradeoff.

Select actions in proportion to their estimates

An improvement over an epsilon-greedy strategy, which is also part of the family of random exploration strategies, is to use a SoftMax distribution over the action-value functions so that the probability of selecting a given action is proportional to its current action-value estimate.

SHOW ME THE MATH

SoftMax Function

(1) To calculate the probability of selecting an action *a*

(2) Use the action-value function for that action divided by the temperature parameter as the preference

(3) Raise e to that value

$$\pi(a) = \frac{e^{\frac{Q(a)}{\tau}}}{\sum_{b=1}^{k} e^{\frac{Q(b)}{\tau}}}$$

(4) Then calculate the same for all possible action and add them up

(5) Finally, divide the value by the sum to normalize the probabilities

By using the **SoftMax strategy**, we are effectively making the action-value estimates an indicator of *preference*. So, it doesn't matter how high or low the values are; if you add a constant to all of them the distribution will stay the same. But, the difference between action-value estimates will create a tendency to select the highest estimates more often. Also, we can add a parameter to tune the sensitivity of the differences between action-value estimates. That way we could choose to make the preferences more pronounced at times or ignore the differences altogether at other times. One good way of using this parameter, called the *temperature*, is to initialize it so that

during the first few episodes, when the probability that the action-value estimates are inaccurate is higher, we treat the difference in the value estimates as less important. This way we would still be selecting actions randomly and in proportion to their values, but more uniformly. Meaning, the probabilities of selecting actions will be close. As episodes go by, we decay this parameter thus making the difference in the action-value estimates more relevant, so in this way, the probability of selecting actions show a meaningful difference. We would be selecting actions in proportion

I Speak Python

SoftMax Action Selection Strategy

```
def softmax(env,
            initial_temp=1.0,
            decay_rate=1e-3,
            min_temp=1e-2,
            n_trials=10000):
    Q = np.zeros((env.action_space.n))
    N = np.zeros((env.action_space.n))

    returns = np.empty(n_trials)
    for t in range(n_trials):
        temp = max(initial_temp * np.exp(-decay_rate * t),
                   min_temp)

        probs = np.exp(Q/temp) / np.sum(np.exp(Q/temp))

        actions = np.arange(len(Q))
        action = np.random.choice(actions, size=1, p=probs)[0]

        _, reward, _, _ = env.step(action)
        N[action] += 1
        Q[action] = Q[action] + (reward - Q[action])/N[action]
        returns[t] = reward

    return np.cumsum(returns)/(np.arange(n_trials)+1)
```

(1) Initialize Q values and counts to zero for all actions

(2) The temperature will be decayed exponentially from `initial_temp` to `min_temp` by the `decay_rate` variables

(3) Calculate the SoftMax distribution as defined earlier

(4) We select an action by the SoftMax probability distribution

(5) Calculating the Monte-Carlo estimates as usual

to their value estimate, but still exploring randomly.

It's not just about optimism; it's about realistic optimism

In the last chapter, you learned about a clever (and perhaps philosophical) approach to dealing with the exploration vs. exploitation tradeoff. I described the optimistic initial values strategy as one of the simplest methods in the optimism in the face of uncertainty family of strategies. Optimism in the face of uncertainty is not only a well-known strategy but also a good idea. But, there are two major inconveniences with the specific algorithm we looked at: First, we do not always know the maximum reward that we can obtain from an environment. If you initialize an optimistic algorithm to a value much higher than the true R_{MAX}, unfortunately, the algorithm will perform sub-optimally because the agent will take much longer to bring down the values near the true values. The other challenge with optimistic initial values is that we used a fake initial number of trials as a proxy for the uncertainty measure. Setting a fake number of initial trials is an incorrect way to model uncertainty, and your agent performance will unfortunately depend on the value of this new hyper-parameter.

We need to find a strategy that, instead of believing everything is roses from the beginning and arbitrarily setting certainty measure values, follows the same principles as optimistic initial values while using statistical techniques to calculate the value estimates and uncertainty, which is what the **upper confidence bound (UCB) strategy** does.

Show Me The Math

Upper Confidence Bound Function

(1) To calculate the action at time t

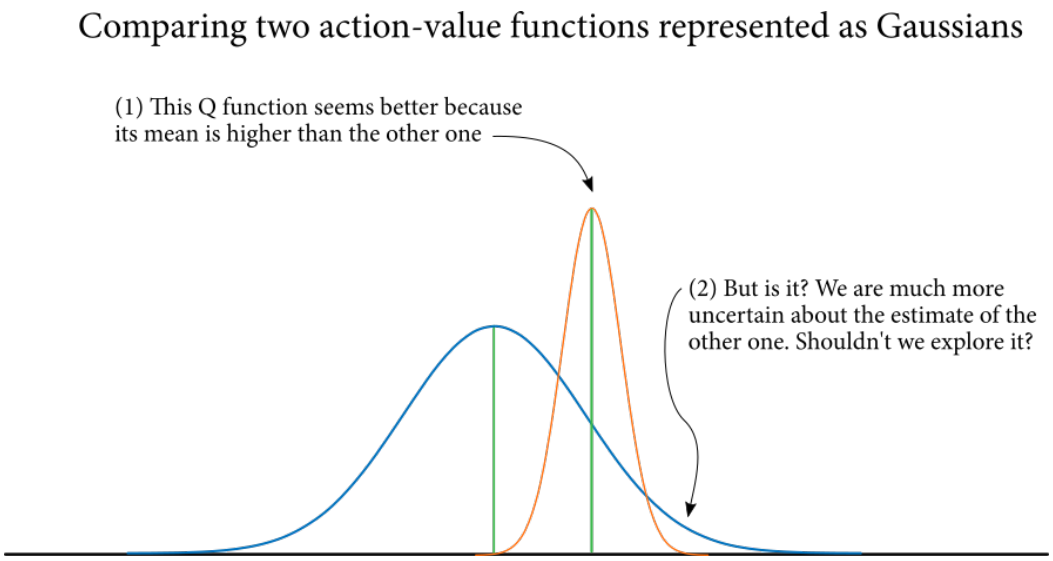
(2) We will select the action with the highest sum

(3) Between the current action-value function estimates

$$A_t = \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

(4) And a measure of uncertainty given by the UCB bonus here

In UCB, we are still optimistic, but it is a more *realistic* optimism; instead of blindly hoping for the best, we look at the uncertainty of value estimates. The more uncertain we are about an action-value estimate, the more important it is to explore it. Note that it is not about believing the value will be the "max," but it might be! The new metric that we care about here is *uncertainty*; we want to give uncertainty the benefit of the doubt.



To implement this strategy, we represent the action-value function as Gaussian distributions with the means representing the action-value estimates, and the variances indicating how confident we are in the estimates. We are going to add a bonus to our action-value estimates, an upper confidence bound $U_t(a)$, such that given we attempt an action only a few times, the U bonus is large so that we encourage exploring this action. While if the number of attempts is large, we add only a small U bonus value to the action-value estimates, because our action-value estimates should be more accurate. Then we select the argmax of the sum of the action-value function Q and the uncertainty bonus U. The c term is a hyper-parameter controling the degree of exploration and confidence level. Using the UCB equation defined above, we are putting a preference on actions that have either high estimates or on actions that have not been sufficiently explored.

I SPEAK PYTHON

Upper Confidence Bound Action Selection Strategy

```
def upper_confidence_bound(env, c=1, n_trials=10000):

    Q = np.zeros((env.action_space.n))
    N = np.zeros((env.action_space.n))

    returns = np.empty(n_trials)
    for t in range(n_trials):

        if t < len(Q):
            action = t
        else:
            U = c * np.sqrt(np.log(t)/N)
            action = np.argmax(Q + U)

        _, reward, _, _ = env.step(action)
        N[action] += 1
        Q[action] = Q[action] + (reward - Q[action])/N[action]

        returns[t] = reward
    return np.cumsum(returns)/(np.arange(n_trials)+1)
```

(1) Initialize Q values and counts to zero for all actions

(2) Before we do any calculation be sure to select each action once. We don't want to be dealing with division by zero

(3) Calculate the *U* bonus as we defined earlier

(4) Action is based on the sum of Q's and U's

(5) Everything else proceeds as usual

Balancing reward and risk

The UCB algorithm is a frequentist approach to the bandits because it makes minimal assumptions about the distributions. There are also many other techniques, Bayesian strategies, that can use priors to make good assumptions and exploit prior knowledge. The **Thompson sampling strategy** is a sample-based probability matching strategy that allows us to use *Bayesian* techniques to balance exploration and exploitation. To implement this strategy, you keep track of each action-value function as a Gaussian, just as we did for the UCB strategy; the Gaussians' means are the action-value estimates and the variance measures the uncertainty of these values. This time, however, we will use these Gaussians to sample the action-value function distribution for each of the Gaussians and select the argmax of these sampled values. We select that action, receive a reward and then recalculate the means and shrink the variance of the Gaussians by applying Bayes rule.

I Speak Python

Bayesian Action Selection Strategy

```
def bayesian(env, tau=1, n_trials=10000):

    R = np.zeros((env.action_space.n))
    N = np.zeros((env.action_space.n))
    means = np.zeros((env.action_space.n))

    returns = np.empty(n_trials)
    for t in range(n_trials):

        if t < len(N):
            action = t
        else:
            samples = np.random.randn() / np.sqrt(N) + means

            action = np.argmax(samples)

        _, reward, _, _ = env.step(action)

        R[action] += reward
        N[action] += tau

        means[action] = tau * R[action] / N[action]

        returns[t] = reward
    return np.cumsum(returns)/(np.arange(n_trials)+1)
```

(1) Initialize Q values and counts to zero for all actions

(2) Before we do any calculation be sure to select each action once. We don't want to be dealing with division by zero

(3) We sample from all the Gaussians

(4) Then select the action with the highest sample

(5) Act

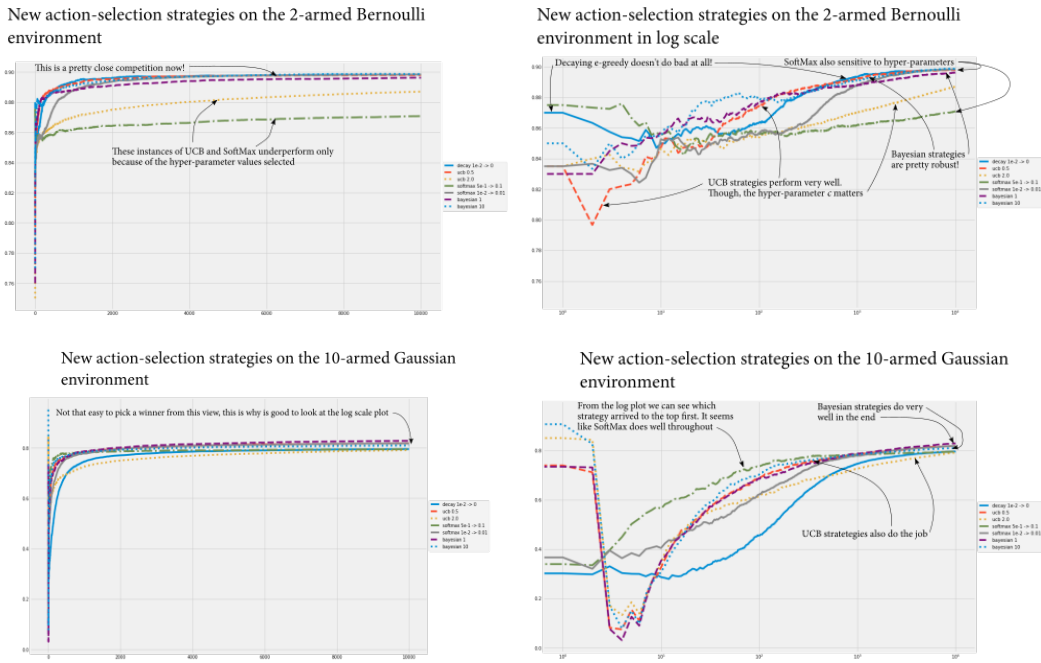
(6) Keep track of the rewards and counts

(7) Recalculate the means of the Gaussians

(8) Everything else proceeds as usual

Comparing the different exploration strategies

Just as in the previous chapter, I ran all strategies described above in both, the 2-armed Bernoulli bandit environment and the 10-armed Gaussian bandit. I included the decaying e-greedy strategy that performed best last time for comparison. You can see that at this point many algorithms perform very well, and I think that by adjusting the hyper-parameters, the action-selecting strategies that are not contending for first place, would. SoftMax, Bayesian, and UCB all do well overall. One of the UCB runs doesn't do as well on the 2-armed Bernoulli, but it is on the best performing on the 10-armed Gaussian bandit.

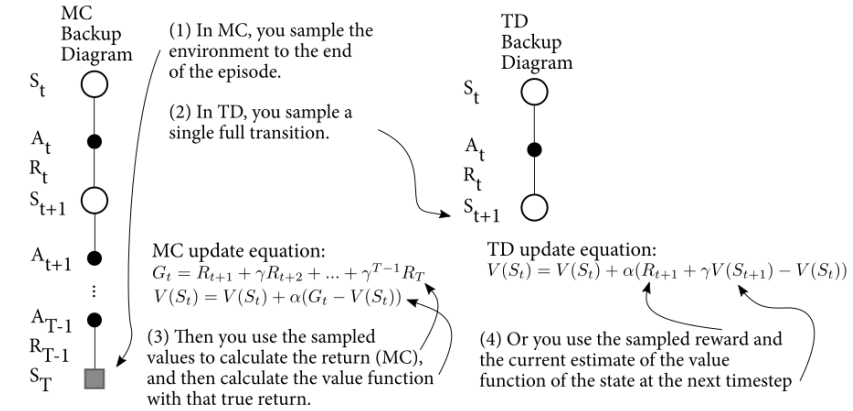


After all this work, the new, strategic exploration methods are still competing with the decaying e-greedy strategy, though. Such a simple strategy performing so well should be surprising or even disappointing. But this is, in fact, one of the reasons why using an e-greedy or decaying e-greedy strategy is often the better way to go.

Agents that that learn often and accurately

In the previous chapter, we looked at the two major algorithms for finding optimal policies through interaction. In MC methods, we sample the environment all the way through the end of the episode before we calculate or *backup* the value functions. We bring all the rewards encountered all the way through the end of the episode back to all value functions that we are calculating: often all visited states or state action pairs. On the other hand, in TD methods, we sample the environment only once before backing up the reward and value from the next state into the current state. You have seen that TD methods *bootstrap*. What that means is that, instead of waiting until the end of an episode to get the true rewards along the path like MC methods do, TD methods use a single reward and then estimate the value function of the next state.

Recall the Monte-Carlo and Temporal-Difference backup trees and equations

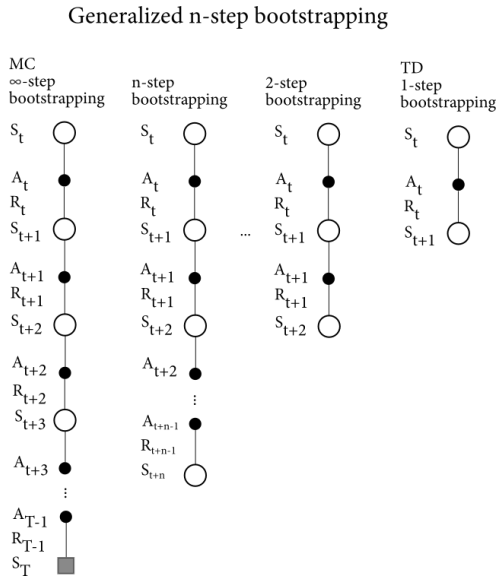


We already discussed in the previous chapter some of the advantages and disadvantages of using either backup style, but I never committed to one. The reason for this is that it is usually a combination of the two styles that perform the best. There is a family of algorithms known as **TD(λ)** that combines MC and TD style backups. In this section, we will explore this family of methods. We will start by discussing a solution to the prediction problem, and we will then go over the control problem.

Learning to predict from multiple timesteps every timestep

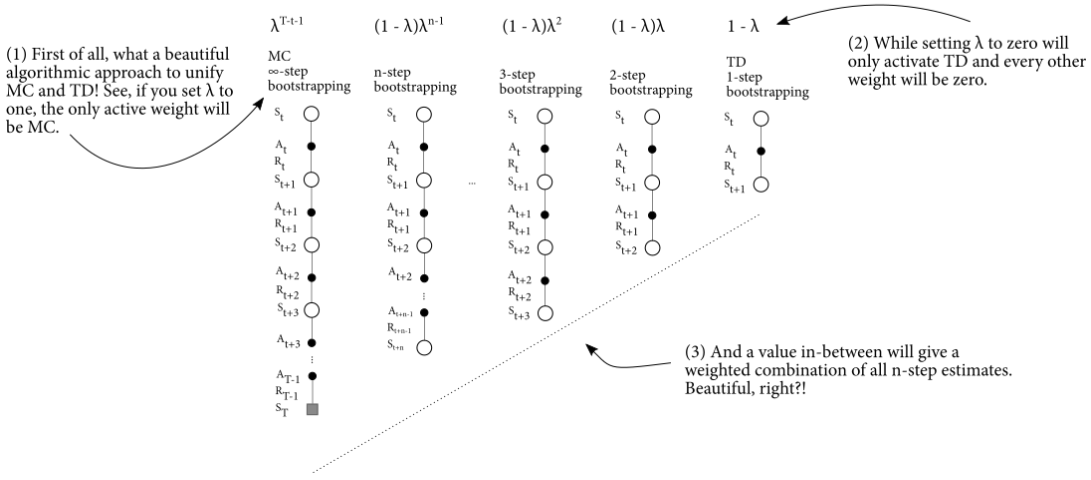
The motivation should be clear; we have two extremes, Monte-Carlo methods, and Temporal-Difference methods. One can perform better than the other depending on the circumstances. MC is an *infinite*-step method because it goes all the way until the end of the episode. I know, "infinite" may sound confusing, but recall we defined a terminal state in the first chapter as a state with all actions and all transitions coming from all actions looping back to itself. And all rewards for all of these transitions set to zero. This way, you can think of an agent "getting stuck" in this loop forever and therefore doing an infinite number of steps without accumulating reward, or changing value functions. TD, on the other hand, is a *one*-step method because it samples a single step.

You can generalize these two methods into a *n*-step method. Instead of doing a single step, like TD, or the full episode like MC, why not use *n*-steps to calculate value functions? This technique is called **n-step bootstrapping**. Some methods use *n*-step bootstrapping, and interestingly an intermediate *n* value often performs the better than either extreme.



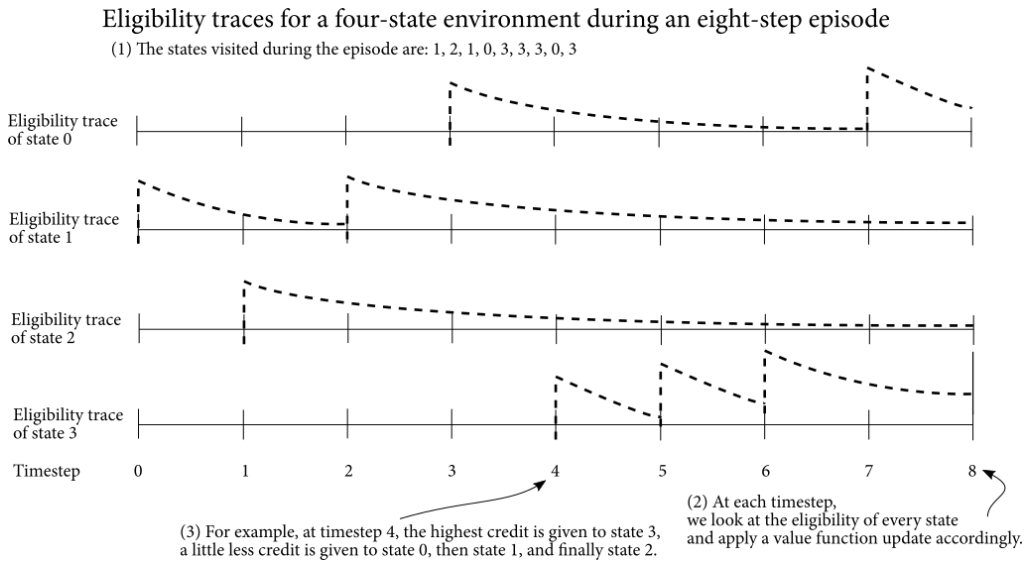
However, with *n*-step bootstrapping, you are still under "the curse of the timestep" because you still have to wait until *n* interactions with the environment have passed before you can make an update to the value estimates. You are basically playing wait and catch-up. For example, in a five-step bootstrapping method you will have to wait until you've seen five states, five rewards before you can make any calculations, a little bit like MC methods. Also, the question remains: what is a good *n* value? Should you use a two-step, three-step or something else. Why not just use a weighted combination of multiple timesteps as our return? This is exactly what the method called TD(λ) does. TD(λ) is a model-free reinforcement learning prediction method that combines multiple *n*-steps into a single update, and instead of having to wait until the end of an episode before applying the updates, it can split these updates into partial updates and apply to the value estimates on every time step. Promising, right? Let me show you.

TD(λ): a model-free reinforcement learning prediction method that uses a weighted combination of all n-step bootstrapping methods to calculate value functions



As you can see in the figure above, in TD(λ), the λ refers to the weighted combination of multiple *n*-step returns. The TD(λ) algorithm unifies MC and TD methods: given a λ value of one, and you get updates equivalent to that of MC, given a λ value of zero, and you get an update equivalent to that of TD. A λ value in between gives you an update which is a combination of multiple *n*-steps. That is multiple steps are weighted according to the λ value.

In addition to generalizing and unifying MC and TD methods, TD(λ) retains the ability to apply updates everytime step just like TD. The mechanism that allows TD(λ) to retain this advantage is known as **eligibility traces**. An eligibility trace is a memory vector that keeps track of recently visited states. The basic idea is to track states that are eligible for an update at any time step. We keep track, not only whether a state is eligible or not, but also by how much. For example, all eligibility traces are initialized to zero, and when you encounter a state, you add a one to its trace. Each time step, you calculate an update to the value function for all states and multiply it by the eligibility trace vector. This way only eligible states will get the update. After the update, the eligibility trace vector is decayed by the λ (weight factor) and γ (discount factor) so that future reinforcing events have less impact on ealier states. By doing this, the most recent states get a larger credit for a reward encountered in a recent transition than those states visited earlier in the episode. Of course, given that λ is not set to one, otherwise this is just a MC update which gives equal credit to all states visited during the episode.



I Speak Python

TD(λ)

```
def td_lambda(pi,
              env,
              gamma=0.9,
              lambda_=0.5,
              initial_alpha=0.2,
              alpha_decay_rate=1e-3,
              min_alpha=0.0,
              n_episodes=10000):

    nS = env.observation_space.n
    V = np.zeros(nS)
    E = np.zeros(nS)
    V_track = np.zeros((n_episodes, nS))

    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                    np.exp(-alpha_decay_rate * t), min_alpha)
        E.fill(0)

        state, done = env.reset(), False
        while not done:
            action = pi[state]
            new_state, reward, done, _ = env.step(action)
            td_error = reward + gamma * V[new_state] - V[state]
            E[state] = E[state] + 1
            V = V + alpha * td_error * E
            E = gamma * lambda_ * E
            state = new_state
        V_track[t] = V
    return V.copy(), V_track

V, V_track = td_lambda(pi, env)
V
```

(1) Initialize V values and counts to zero for all actions

(2) E is the eligibility trace vector, one value per state

(3) On every episode, set all eligibility traces to zero

(4) Calculate TD error just as we do in vanilla TD.

(5) Increment the eligibility of the state

(6) Apply the update to all states relative to their eligibility

(7) Reduce the eligibility of all states as decribed before

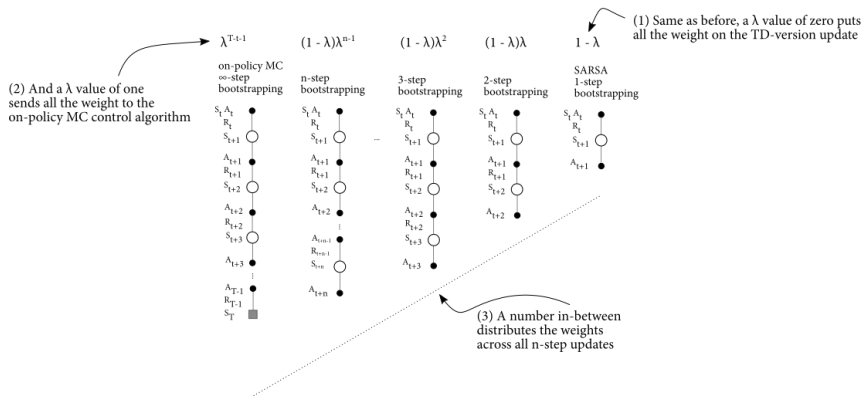
(8) Everything proceeds as normal

One of the main advantages of TD(λ) over *n*-steps methods is that you no longer must wait until *n* interactions with the environment. *n*-step methods are known as *"forward view"* because you must look forward and wait until *n* interactions have passed before you can complete an update. By keeping track of the eligibility of all states for an update, you can apply partial updates on each time step, which brings a couple of other advantages: first, the value functions will always include the most recent information for all states which can be important for decision making. Also, you don't have to wait and calculate massive updates every *n* step. Instead, you will be splitting the computational requirements of the algorithm. Each step you calculate similarly complex updates. Methods that use eligibility traces, such as TD(λ) are known as a *"backward view"* because they look backward by keeping track of states eligible for updates using eligibility traces.

Learning to act from multiple timesteps every timestep

We can apply the concept of eligibility traces to control problems as well. It is very simple to adapt what you learned in the previous section to the algorithms you learned in the previous chapter; for example, SARSA or Q-Learning. The first thing to do is to replace the use of state-value functions in TD(λ) with action-value functions, like we did in SARSA and Q-Learning. Also, we need to track the eligibility of state-action pairs instead of only states because we are building a control method. Finally, we select an improvement step strategy such as the epsilon-decaying greedy we've been using in all of our model-free reinforcement learning algorithms so far. By doing these things, we have ourselves a couple new agents. Let's first look at the on-policy control version of TD(λ), **SARSA(λ)**.

SARSA(λ): a model-free on-policy reinforcement learning control method



I Speak Python

SARSA(λ) 1/2

```
def sarsa_lambda(env,
                 gamma = 0.9,
                 lambda_ = 0.5,
                 initial_alpha = 1.0,
                 alpha_decay_rate = 5e-4,
                 min_alpha = 1e-4,
                 initial_epsilon = 1.,
                 epsilon_decay_rate = 1e-5,
                 min_epsilon = 0.0,
                 n_episodes=100000):

    nS, nA = env.observation_space.n, env.action_space.n
    Q = np.zeros((nS, nA))
    Q_track = np.zeros((n_episodes, nS, nA))

    E = np.zeros((nS, nA))

    select_action = lambda state, Q, epsilon: \
        np.argmax(Q[state]) \
        if np.random.random() > epsilon \
        else np.random.randint(nA)

    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                    np.exp(-alpha_decay_rate * t), min_alpha)
        epsilon = max(initial_epsilon * \
                    np.exp(-epsilon_decay_rate * t), min_epsilon)
        E.fill(0)
        state, done = env.reset(), False
        action = select_action(state, Q, epsilon)
```

(1) Initialize the action-value function to zero

(2) Eligibility traces are now a 2-d matrix indexed by the state and action

(3) Same decaying epsilon-greedy strategy as in all previous agents

(4) For each episode, we decay alpha exponentially

(5) Same with epsilon

(6) We set all eligibility traces to zero at the beginning of each episode

(7) We proceed just like in vanilla SARSA. Get the initial state, and action

(8) Algorithm continues...

I Speak Python

SARSA(λ) 2/2

(9) We enter the main loop and repeat the following on every timestep

```
while not done:
```

(10) First, interact with the environment and observe the transition

```
    new_state, reward, done, _ = env.step(action)
```

```
    new_action = select_action(new_state, Q, epsilon)
```

(11) Select the new action for the new state

```
    if done:
        Q[new_state] = 0.
        Q_est = reward
```

(12) Same as before, a terminal state should have a special kind of update to avoid the value going to infinity.

(13) Non-terminal states have the same SARSA, TD on-policy control update.

```
    else:
        Q_est = reward + gamma * Q[new_state][new_action]
```

(14) Increase the eligibility of the current state by one

```
    E[state][action] = E[state][action] + 1
```

(15) Apply a similar update as we did for TD(λ), but to action-value functions. See the eligibility trace.

```
    Q = Q + alpha * (Q_est - Q[state][action]) * E
```

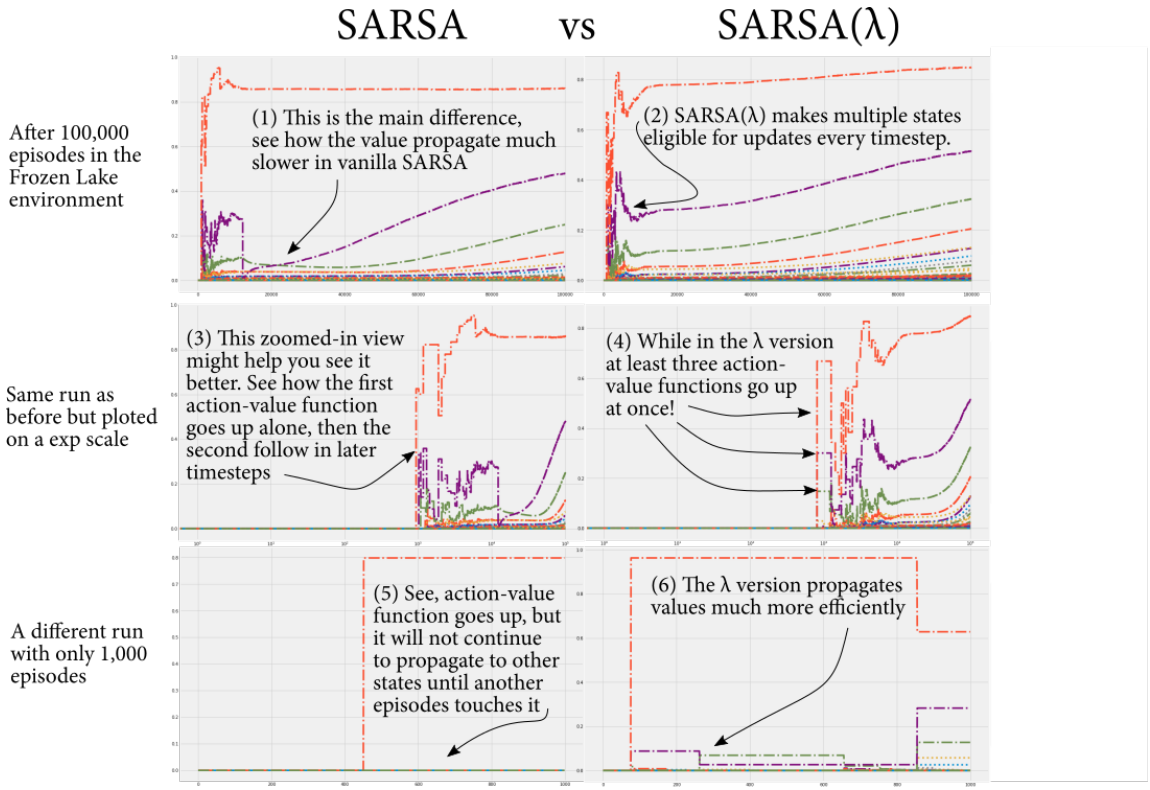
(16) Decaying the eligibility traces

```
    E = gamma * lambda_ * E
```

(17) Algorithm proceeds as vanilla SARSA, also known SARSA(0)

```
    state, action = new_state, new_action

Q_track[t] = Q

pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}

V = np.max(Q, axis=1)
return Q, V, pi, Q_track


Q_best, V_best, pi_best, Q_track = sarsa_lambda(env)
Q_best, V_best, pi_best
```

I run both types of agents, SARSA and SARSA(λ), on the 8x8 Frozen Lake environment I described in the previous chapter. The results favor the use of eligibility traces. Action-value function propagate much faster and the SARSA(λ) algorithm seems more efficient than SARSA.

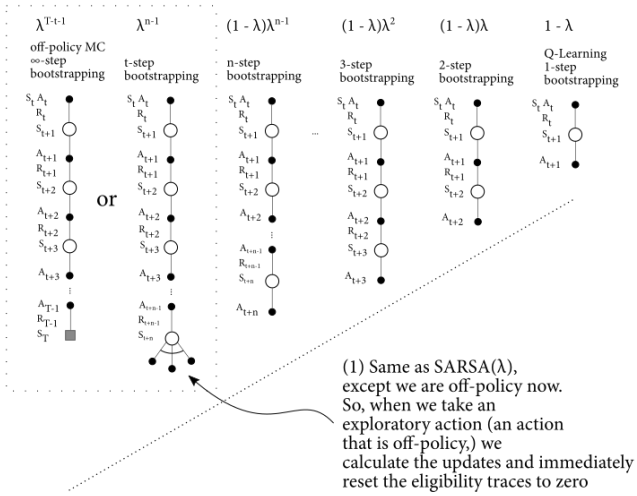


The benefit of using eligibility traces is evident, SARSA(λ) is more efficient at propagating updates than SARSA. In the first episode that ends at the goal state in this environment, SARSA, which is a *one*-step bootstrapping method, will only increment the action-value function one-step away from the goal state. *n*-step bootstrapping methods will increment *n* action-value functions before the goal state at equal values. However, SARSA(λ), an eligibility trace method, will update all action-value functions experienced during that episode to different degrees, weighted by recency. This difference is important as it compounds quickly. In the next episode, SARSA will only propagate back one step the action-value functions that either result in an immediate reward (in this environment the only reward is at the goal state) or after

experiencing an action-value function with a value other than 0, which at this point should only be the value function one step away from the goal. That is likely a total of only two action-value updates in two episodes, that is, if we are lucky to hit (or get close to) the goal state twice. On the other hand, SARSA(λ) will update several action-value functions the first time it hits the goal. Then, it will propagate the values even more because the action-value functions with non-zero values will be many more than SARSA after the first episode. So, SARSA(λ) will have many non-zero action-value functions the episode after the first episode ending in a goal state. This will help SARSA(λ) propagate values throughout multiple states, not just the final or one step away from the final state. Given that the non-zero action-value functions are many more and spread throughout the environment, we are likely to experience multiple updates to many action-value functions on the same episode.

There is also an off-policy control version of the λ algorithms. **Q(λ)** is an extension of Q-Learning to use λ returns. In fact, there are different ways to extend Q-Learning to eligibility traces; I will only explain the original version, commonly referred to as **Watkin's Q(λ)**. Q-Learning is an off-policy, one-step bootstrapping, model-free reinforcement learning method. Off-policy, again, means that the policy learned about, the *target* policy, is not necessarily the same as the policy used for action selection, the *behavior* policy. Q-Learning learns about the greedy policy while following an exploratory policy such as one with decaying epsilon-greedy action selection strategy. Being an off-policy learning method presents a challenge when extending

Q(λ): a model-free off-policy reinforcement learning control method



Q-Learning to eligibility traces because in Q-Learning we use a *max* operator to calculate our target update instead of the action-value function of the state-action pair we will be actually experiencing. For example, we act *non-greedily* when we select an exploratory action. The eligibility trace that comes after taking an exploratory action is likely no longer related to the greedy policy. For this reason, instead of resetting the eligibility traces vector to zero at the *end* of the episode as we did in TD(λ) and SARSA(λ), in Q(λ), we must reset it after taking an exploratory action, or at the end of the episode if no exploratory actions were taken.

I SPEAK PYTHON

Watkin's Q(λ) 1/2

```
def q_lambda(env,
             gamma = 0.9,
             lambda_ = 0.5,
             initial_alpha = 0.5,
             alpha_decay_rate = 1e-3,
             min_alpha = 0.001,
             initial_epsilon = 1.,
             epsilon_decay_rate = 1e-5,
             min_epsilon = 0.0,
             n_episodes=100000):
    nS, nA = env.observation_space.n, env.action_space.n
    Q = np.zeros((nS, nA))
    E = np.zeros((nS, nA))
    Q_track = np.zeros((n_episodes, nS, nA))
    select_action = lambda state, Q, epsilon: \
        np.argmax(Q[state]) \
        if np.random.random() > epsilon \
        else np.random.randint(nA)
    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                    np.exp(-alpha_decay_rate * t), min_alpha)
        epsilon = max(initial_epsilon * \
                      np.exp(-epsilon_decay_rate * t), min_epsilon)
        E.fill(0)
        state, done = env.reset(), False
```

(1) Initialize the action-value function and eligibility traces to zero

(2) Set action selection strategy as in all previous algorithms

(3) At the beginning of every episode calculate the new alpha and epsilon

(4) Reset the eligibility traces to 0

(5) Algorithm continues...

I Speak Python

Watkin's Q(λ) 2/2

(6) Now, in every timestep for that episode

```
    while not done:
```

(7) Select action for current state and act

```
        action = select_action(state, Q, epsilon)
        new_state, reward, done, _ = env.step(action)
```

(8) Select the action for next state and calculate what's the greedy action. We need to do this to find out if we are taking an exploratory or greedy action next timestep

```
        new_action = select_action(new_state, Q, epsilon)
        best_new_action = np.argmax(Q[new_state])
```

(9) Make sure we calculate the max consistently with the argmax

```
        if np.allclose(Q[new_state][new_action], \
                       Q[new_state][best_new_action]):
            best_new_action = new_action
```

(10) Just as before, calculate the Q target handling terminal states adequately

```
        if done:
            Q[new_state] = 0.
            Q_est = reward
        else:
            Q_est = reward + gamma * Q[new_state].max()
```

(11) Keep track of eligibilities and calculate new values just as before

```
        E[state][action] = E[state][action] + 1
        Q = Q + alpha * (Q_est - Q[state][action]) * E
```

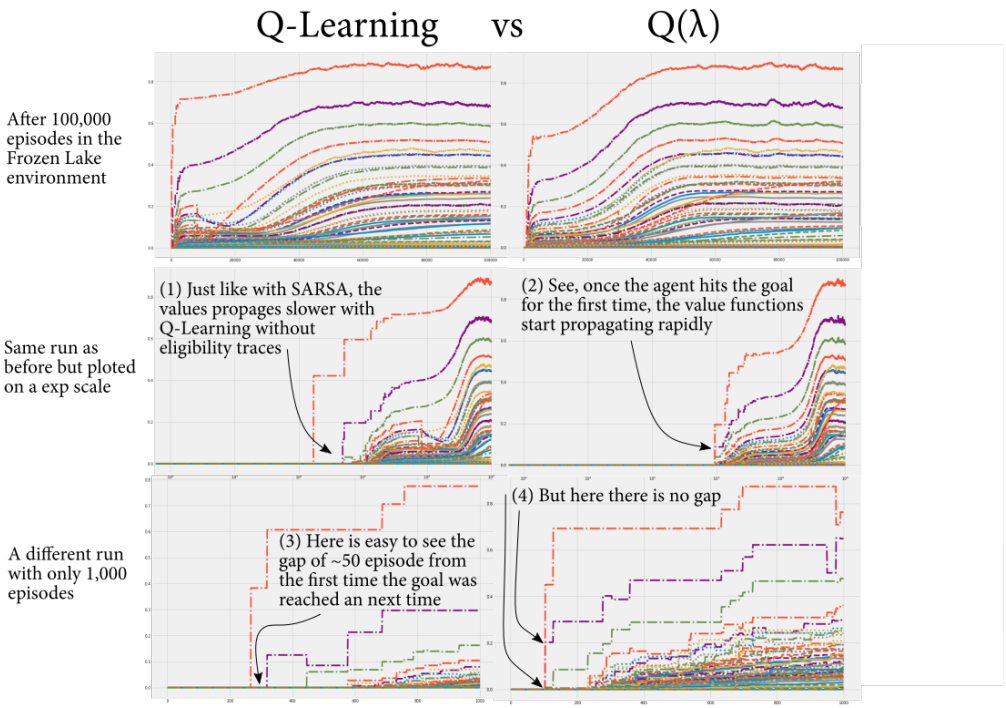
(12) If the action was a greedy action, then use the eligibility traces as expected. Otherwise, reset all traces to zero!!!

```
        if new_action == best_new_action:
            E = gamma * lambda_ * E
        else:
            E.fill(0)
```

(13) After that the algorithm proceeds as expected

```
        state, action = new_state, new_action
    Q_track[t] = Q
pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
V = np.max(Q, axis=1)
return Q, V, pi, Q_track

Q_best, V_best, pi_best, Q_track = q_lambda(env)
Q_best, V_best, pi_best
```

I ran Q-Learning and Q(λ) for a side-by-side comparison. As you can see in the graphs below, Q(λ) is still more efficient than Q-Learning at propagating the action-value functions. However, it is arguably not as efficient as SARSA(λ). As you can imagine, having to reset the eligibility trace on every exploratory action will dilute some of the advantages we had gained by using eligibility traces in the first place.



Still, Q(λ) and the idea of eligibility traces, in general, unify and generalize the MC and TD update style. This more general approach not only provides an easy way of shifting between the two extremes but also allows us to do incremental and efficient updates to our value functions. MC updates have a low bias because they do not bootstrap. Low bias is advantageous when dealing with non-Markovian tasks (recall chapter 2, Markov property: the probability of a future is only dependent in the current state. The past doesn't matter.) But TD methods have advantages over MC; particularly, value functions can be updated every time step when using TD methods. With eligibility traces, we get the best of both worlds: we can have the low bias that MC methods have, but we can also propagate values every time step.

There is more work on eligibility traces than what I presented here, but one final issue I'd like to mention is regarding the value for λ. What is a good λ value? Unfortunately, there is no straightforward answer to this question yet, and you may have to tune this parameter for each problem you're trying to solve. There is also no theory suggesting that the use of eligibility traces is strictly better than the other methods learned in the previous chapter. However, an intuition that can help you decide whether you should use a λ method is whether the rewards are delayed multiple steps. As you saw, λ methods propagate values much more efficiently, yet they require more computation. So, if you are using a simulation or solving an off-line task in which there is no online interaction with the environment, consider sticking to the methods learned in the previous chapter. But, if data is scarce and you must extract the most information possible from each episode, then consider using a λ method.

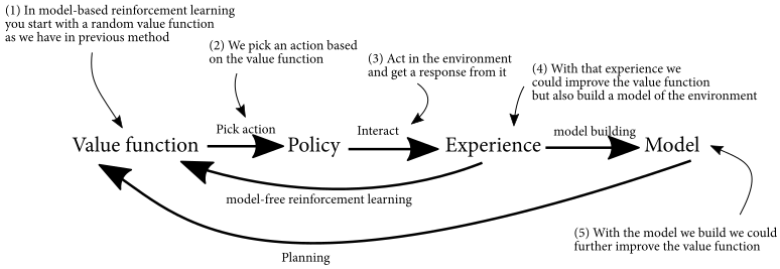
Agents that interact, learn and plan

In chapter 2, we discussed *planning* algorithms such as value iteration (VI) and policy iteration (PI). These are planning algorithms because they don't require interaction with the environment. They can calculate optimal policies offline. However, they do require a map of the environment, an markov decision process (MDP). With that map, they can look at it and compute optimal policies without having to do any trial-and-error learning. On the other hand, in chapter 3, I presented model-free or *learning* methods suggesting that they were an improvement over planning methods. The advantage of model-free methods over planning methods is that the former does not require the MDP. Often MDPs are difficult to obtain in advance. Not requiring an MDP in advanced is a practical benefit. But, what if we do not require and MDP in advance, but perhaps *build* one as we interact with the environment? Think about it, as you walk around a new area, you start building a map in your head. You turn right, walk a couple of blocks, find a coffee shop. You know that to get back to the original place, you walk a couple of block in the opposite direction and then turn left. This should be pretty intuitive to you.

In this section, we explore agents that interact with the environment, just like the model-free methods you learned in the previous chapter and section, they learn value functions and *models* of the environment from these interactions, and then use these models to improve the value function estimates. These are **model-based rein-**

forcement learning methods. Note that in the literature, you will often see VI and PI referred to as planning methods, but you may also see them referred to as model-based methods. I prefer to call them planning methods. SARSA and Q-Learning algorithms are model-free because they do not require and *do not learn* an MDP. The methods that I will teach you in this section are model-based reinforcement learning methods because the do not require but learn an MDP (or an approximation.)

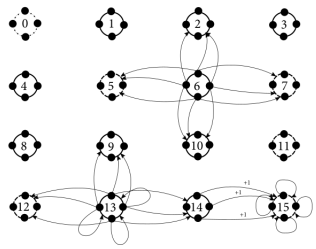
Model-based reinforcement learning



Learning models

Model is an overloaded word in the machine learning community. Model in supervised learning is a very different thing than what I'm referring to here. In reinforcement learning, a model is an MDP or an MDP approximation; something an agent can use to predict the transition and reward function. In later chapters, we will a supervised learning model (you see how it is an overloaded term?) to predict these, so into the model goes the current state and action, out from the model we get predicted next states and rewards. But in this chapter, when I talk about models, I'm referring to two tensors for that, one for the transition function, and one for the reward function.

The 4x4 Frozen Lake environment MDP. A model of the environment.



There is two types of models you could build: a *distribution* model or a *sample* model. The former calculates entire probability distributions of next state and rewards, while the latter calculates a single possibility sampled from their probabilities of occurring. There are also two ways you can use the real experience generated by interacting with the environment: you could do *model-learning* or *direct reinforcement*

learning. The former allows you to build a more accurate model of the environment which can then allow you to plan better; the latter is simply the improvement of value functions as we know it. *Indirect reinforcement learning* would be the improvement of value functions using experience generated by a learned model, as in model-learning.

Interacting, learning, and planning

One of the most well-known architectures for unifying planning and model-free methods is the **Dyna-Q** architecture. What we are trying to accomplish here is to have some of the benefits of planning while retaining the freedom of model-free reinforcement learning. When we plan, we improve the value functions and possibly the policy without any interaction. It is often the case that only a few iterations of planning interleaved with a vanilla model-free method can make our algorithms effective more quickly.

What we are going in Dyna-Q is keep track of both the transition and reward function as a 3-dimensional tensor indexed by the initial state, the action and the next state. The transition tensor will keep count of the number of times we've seen the 3-tuple *(s, a, s')* indicating how many times we arrived at state *s'* when we were at state *s* and selected action *a*. The reward tensor will hold the average reward we received on the 3-tuple *(s, a, s')* indicating the expected reward when we selected action *a* on state *s* and transition to state *s'*. Keeping track of counts and averages this way is not the only way to model the environment, of course. You can see that we are assuming the environment is the deterministic because of how we are modeling the reward. It could be that with 50% chance we a very high reward, say 100, but with other 50% we get a very low reward, say -100. We would still be modeling zero reward for that transition, but that could turn to be misleading information.

In Dyna-Q we learn the model as previously described, adjust action-value functions as we do in vanilla Q-Learning, and then run a few planning iterations at the end of the algorithm. If we were to remove the model-learning and planning lines from the code, we would be left with the same exact Q-Learning algorithm as we had in the previous chapter. When we are in the planning phase, only state-action pairs that have been used are queried so to not waste resources with state-action pairs the model has no information of. Out of the list of those visited state-action pairs, we randomly sample a state-action pair and then sample a next state and reward. We

apply a one-step tabular Q-learning update to the 4-tuple *(s, a, r, s')* obtained from the simulations. This planning strategy is called **Q-planning**. Take a look at the full algorithm.

I Speak Python

Dyna-Q 1/3

```
def dyna_q(env,
           gamma = 0.9,
           initial_alpha = 0.5,
           alpha_decay_rate = 1e-3,
           min_alpha = 0.01,
           initial_epsilon = 1.,
           epsilon_decay_rate = 1e-4,
           min_epsilon = 0.01,
           n_planning=20,
           n_episodes=5000):

    nS, nA = env.observation_space.n, env.action_space.n
    Q = np.zeros((nS, nA))
    T_count = np.zeros((nS, nA, nS))
    R_model = np.zeros((nS, nA, nS))

    Q_track = np.zeros((n_episodes, nS, nA))
    select_action = lambda state, Q, epsilon: \
        np.argmax(Q[state]) \
        if np.random.random() > epsilon \
        else np.random.randint(nA)
    for t in tqdm(range(n_episodes)):
        alpha = max(initial_alpha * \
                    np.exp(-alpha_decay_rate * t), min_alpha)
        epsilon = max(initial_epsilon * \
                      np.exp(-epsilon_decay_rate * t), min_epsilon)
        state, done = env.reset(), False
```

(1) Initialize the usual suspect

(2) Initialize two tensors, one to keep track of the transition and another to approximate the reward function

(3) Setup the action selection strategy function as we've done before

(4) Start the episode as we have in all previous algorithms

(5) At the beginning of every episode, calculate the new alpha and epsilon

(6) Reset the environment and get the initial state

(7) Algorithm continues...

I Speak Python

Dyna-Q 2/3

(8) In the main loop of each episode

```
while not done:
    action = select_action(state, Q, epsilon)
    new_state, reward, done, _ = env.step(action)
```

(9) Select action and act

```
    T_count[state][action][new_state] += 1
    r_diff = reward - R_model[state][action][new_state]
    R_model[state][action][new_state] += \
                r_diff / T_count[state][action][new_state]
```

(10) We use update the T_count tensor to approximate the transition function

(11) We also calculate the average reward we get for getting on that transition

```
    if done:
        Q[new_state] = 0.
        Q_est = reward
    else:
        Q_est = reward + gamma * Q[new_state].max()
```

(12) We treat terminal and non-terminal states as usual

(13) Update the action-value function as always

```
    Q[state][action] = Q[state][action] + alpha * \
                                (Q_est - Q[state][action])
    backup_new_state = new_state
    for _ in range(n_planning):
        if Q.sum() == 0:
            break
```

(14) Backup the new state before getting into planning steps

(15) If no value function has been updated don't plan

(16) Select a random state from the list of all visited states

```
        visited_states = np.where(\
                    np.sum(T_count, axis=(1,2)) > 0)[0]
        state = np.random.choice(visited_states)
```

(17) Select a random action from the list of all action taken in that state

```
        actions_taken = np.where(\
                    np.sum(T_count[state], axis=1) > 0)[0]
        action = np.random.choice(actions_taken)
```

(18) From that state and action, sample the model for a next state, and a next reward

```
        T_sum = T_count[state][action].sum()
        probs = T_count[state][action]/ T_sum
        new_state = np.random.choice(\
              np.arange(len(T_count[state][action])),
              size=1, p=probs)[0]
        reward = R_model[state][action][new_state]
```

(19) Algorithm continues...

I Speak Python

Dyna-Q 3/3

(20) The planning updates, called Q-planning, look just like a regular Q-Learning update

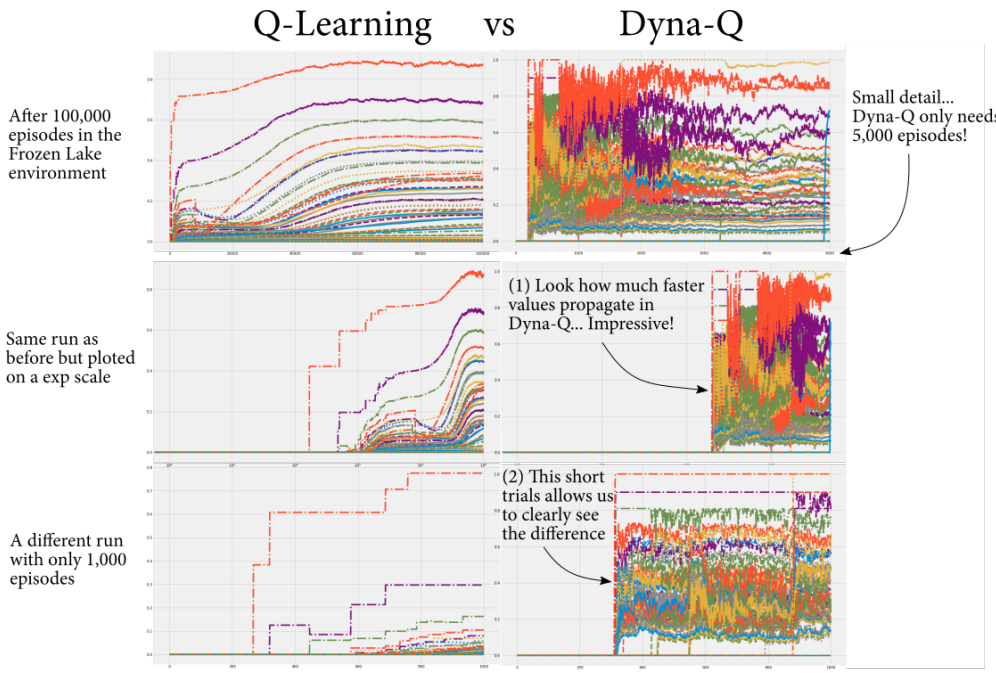
```
            Q_est = reward + gamma * Q[new_state].max()
            Q[state][action] = Q[state][action] + alpha * \
                                        (Q_est - Q[state][action])
        state = backup_new_state
    Q_track[t] = Q
```

(21) Restore the state after planning

(22) Proceed as usual

```
    pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
    V = np.max(Q, axis=1)
    return Q, V, pi, Q_track
Q_best, V_best, pi_best, Q_track = dyna_q(env)
Q_best, V_best, pi_best
```

Just like for all other methods, I ran Dyna-Q on the 8x8 Frozen Lake environment and the results are very interesting. In only 5,000 episodes Dyna-Q is able to reach the same performance as Q-Learning in 100,000 episodes.



In fact, the biggest challenge was to have enough episodes for the agent to touch the goal state for the first time. Given that the 8x8 Frozen Lake environment only carries a non-zero reward at the final, goal state, there are no values to propagate unless you touch the final state at least once. After that, the agent had already learned a model strong enough to improve action-value functions quickly without any interaction.

Dyna-Q is a great method for introducing model-based reinforcement learning algorithms but is not the only method that integrates model-free, model-learning and planning. Algorithms that use planning have been hitting the news hard recently. For example, the DeepMind guys use a method called **Monte-Carlo Tree Search (MCTS)** for decision-time planning. The core idea of MCTS is to rollout several simulations from every new state to evaluate and select the best actions. This search method rolls out full episodes to a terminal state, like in Monte-Carlo methods for episodic tasks, but not from the starting state, instead of from the current state. The idea is to focus the search and planning on states that look promising so to improve action selection based on previously acquired experience. As you can see, we could use search methods for planning, and we would still get many improvements on our reinforcement learning agents.

Summary

In this chapter, you learned to improve on many of the algorithms previously learned. You started by improving on some of the action-selection strategies learned in the previous chapter. You learned that SoftMax action-selection strategy is a strategy under the random exploration strategies family. You learned about a couple of optimistic strategies that don't assume you know maximum reward values in advanced but instead keep track of uncertainty. You learned about UCB as a frequentist approach and about Thompson sampling as a Bayesian approach to the same "optimism in the face of uncertainty" principle.

You then unify many of the principles and methods you learned in all previous chapters. First, you unified Monte-Carlo methods and Temporal-Differences methods using eligibility traces. You learned that eligibility traces are a mechanistic approach to propagate updates backward to all states previously visited. You learned that the eligibility trace is not just a Boolean tracking whether the state should be updated but a float instead that keeps track of the corresponding weight to be applied to the current update. You learned how these λ methods, such as SARSA(λ) or Q(λ) are

more efficient because they make more use of the reinforcing events encountered. Additionally, you learned that these methods apply these updates partially on every time step and do not have to wait until the end of an episode or even after *n* interactions before they can update the value functions.

Lastly, you unify planning methods and model-free reinforcement learning methods to give rise to a new family of methods known as model-based reinforcement learning. You learned about the Dyna architecture and more specifically, the Dyna-Q agent. You learned that Dyna-Q does model-learning by assuming a deterministic model, does planning through the Q-planning method which consists of generating a random sample from this model and applying a Q-Learning update to this simulated experience. You learned that beyond the Q-planning steps, Dyna-Q is simply a Q-Learning agent.

You ought to be excited now. In the next chapter, we enter the world of deep reinforcement learning. You are now at a point in which tabular reinforcement learning is not enough. When the state-space of the environment your agent is interacting with is too large, you need to leverage the power of function approximation. This is not only required when the state-space doesn't fit in a table on computer memory, but it is also a good idea in order to speed-up learning by using generalization across different states or state-action pairs. We will harness the power of a popular type of function approximators known as multi-layer neural networks (deep learning) to generalize and be able to solve complex reinforcement learning problems.

By now you:

- Can solve difficult reinforcement learning problems where rewards are sparse, and/or episodes are very long on average.
- Can solve reinforcement learning problems more efficiently by combining MC methods and TD methods.
- Can solve reinforcement learning problems effectively more quickly by combining planning and model-free learning principles.
- Are ready to dive into the world of deep reinforcement learning.