Master's Degree in Mathematics

Final project

DEEP REINFORCEMENT LEARNING AND ITS APPLICATION TO GAMES

Alberto Torrejón Valenzuela
Faculty of Mathematics
Sevilla, August 2021

Tutors:
José Luis Pino Mejías
Rafael Pino Mejías

To my brother. Keep on working, time will come.

Index

Greetings

"And the child grew and became strong; he was filled with wisdom,
and the grace of God was on him". Lucas 2:40.

Thanks to mom and dad.

Thanks to my tutors Jose Luis Pino Mejías, for being always available and helpful, and to Rafael Pino Mejías, who, despite he could not be present this year, taught me to love computational statistics.

Alberto Torrejón Valenzuela

Abstract

The following project aims is to review the main concepts of Reinforcement Learning and combine them with the tools of Deep Learning, studying in depth the application of these methodologies, the **Deep Reinforcement Learning** algorithms, that are having such an impact today being applied to numerous fields such as autonomous driving, robot control, gaming and many more. In order to do this, first, in *chapter 1*, we will give a general overview of Deep Reinforcement Learning as a introduction, as well as which is motivation to study this topic. Then, in *chapter 2*, since it will be fundamental to achieve our goal, we give a brief review of Deep Learning. We get into details with *chapter 3*, where we define Reinforcement Learning mathematically, formalizing the concepts in order to build the classic solution algorithms in *chapter 4*. As an application of these techniques, the implementation of the algorithms for the game of Blackjack is presented in *chapter 5*. Finally, in *chapter 6*, we reach our initial objective by building the algorithm that hides behind the Deep Q-Networks and we apply it to the Gridworld games in *chapter 7*. A *conclusions and improvements* section for the project culminates the text.

Resumen

El siguiente proyecto tiene como objetivo revisar los principales conceptos del Aprendizaje con Refuerzo y combinarlo con las herramientas del Aprendizaje Profundo, estudiando con detalle la aplicación de estas metodologías, **Aprendizaje con Refuerzo Profundo**, que están teniendo tanto impacto en la actualidad siendo aplicados a numerosos campos como la conducción autónoma, el control de robots, juegos y muchos más. Para ello, en primer lugar, en el *capítulo 1*, situaremos al Aprendizaje con Refuerzo Profundo a modo de introducción, motivando el estudio de este campo. Acto seguido, en el *capítulo 2*, ya que será fundamental para lograr nuestro objetivo, se realiza una breve revisión del Aprendizaje Profundo. Entraremos en detalles con el *capítulo 3*, donde definiremos matemáticamente que se entiende Aprendizaje con Refuerzo, formalizando los conceptos con el fin de construir los algoritmos de solución clásicos en el *capítulo 4*. Como aplicación de estas técnicas, en el *capítulo 5* se presenta la implementación de los algoritmos para el juego del Blackjack. Finalmente, en el *capítulo 5*, alcanzaremos nuestro objetivo inicial construyendo el algoritmo detrás de las Deep Q-Networks y lo aplicamos a los juegos Gridworld en *capítulo 7*. Una sección de *conclusiones y mejoras* para el proyecto culmina el texto.

Figure index

Table index

Only intelligence examines itself.

Jaime Luciano Balmes

1 Introduction

1.1 First name, *Reinforcement*, last name, *Deep*

You sit at night to study, turn on the lamp light and take off your slippers to make yourself more comfortable. Before getting up again, you try to put them on, but you can not find the slippers in the same place you thought you left them at the beginning, the light does not illuminate enough to see where they are and also, very often, we are too lazy to turn our body down and look for them. What would you do?

You are tired and you decide to go to the bedroom, but you only have one switch at the beginning of the bedroom, many things in between and at the end of a long journey, the bed. What would you do?

In the morning the alarm rings, you get up and almost automatically you go to the bathroom to wash up, get dressed and put first gear towards a new day, known as routine...

All of these issues require an intelligent response to achieve the desired goal. In the first of the previous cases, we usually use the foot, either by tapping the ground or as a scanner, until it hits the shoes. Bingo! In the second one, an option is to observe the path that best suits you in advance, try to memorize it and execute it once the light is off so you do not have to return back to the switch. In the last case, the automation of the processes saves us time and energy since our brain does not need to rethink the solution again, routine is helpful for us.

Intelligence is what makes the human race different, it is the engine that governs life and on which the evolution process is sustained (although evolution might be a staggered process and even with periods of involution due to the different degrees of intelligence of people). Therefore, it is not surprising that one of the main research goals in the world is to understand human intelligence and replicate it. *Doubt*, *deception*, *imagination* or *creativity*, the ability to *dream* or *make decisions* based on our own experiences, are the greatest exponents of human intelligence and what we have always tried to understand in order to know ourselves. Psychology or philosophy try to do it from a humanistic point of view; neuroscience, from a chemical and biological point of view; and mathematics tries to formalize it in order to being able to replicate it.

The branch of mathematics in charge of this task, shared with other sciences such as physics or engineering, is known as **Artificial Intelligence**. By Artificial Intelligence (AI) we understand the intelligence carried out by machines. An ideal “smart” machine is a flexible agent who perceives its environment and takes actions that maximize its chances of success. For example, the first chess programs involved a series of codified rules. It was believed that AI would advance to the human level by crafting a large enough set of explicit rules of knowledge. This is what is known as symbolic AI, which was the object of study between the 1950s and 1980s.

The search, almost Faustian, for a **general** or **strong AI**, capable of imitating human behavior in its image and likeness, then became the main and most ambitious objective of researchers. Experiments then arise to find out if an AI is capable of replicating the human being, such as the *Turing Test* (although widely questioned, for example by the *Chinese room experiment*), in which an interrogator is asked to identify whether the entity that is interviewing is a person or a machine through successive questions. In this way, the construction of a broad set of rules proved to be adequate for well-defined logical problems, such as playing chess, but not for, due to the difficulty of elaborating explicit rules regarding more complex tasks such as perceiving, reasoning or learning. Whereupon the idea of **machine learning** and **weak AI**, capable of performing an intellectual task with total perfection, emerged as a new approach.

There are numerous disciplines involved in AI and the classification of all these disciplines is not yet clear mainly because AI is constantly under renovation. However, machine learning is considered one of the main branches of this field. Depending on the way of learning, the *learning rule* followed, there are mainly three different types of learning:

- **Supervised learning** (SL): A training set of examples is provided with the values of the target variable and based on this training set the algorithm is generalized to respond correctly to all possible inputs.
- **Unsupervised learning** (USL): The values of the target variable are not known; is the algorithm that tries to identify similarities between the inputs.
- **Reinforcement learning** (RL): The algorithm is told if the answer is wrong but not how to correct it. You have to explore and try different possibilities and learn from experience until you figure out how to get the correct answer. We will study RL in detail in chapters 3 and 4.

When machine learning focuses on one or two layers of data representations it is called superficial learning. In **deep learning** (DL), the referred representations in layers are almost always achieved through models called *Artificial Neuronal Networks*, combining some layers of neurons on top of others. DL can then be defined either as a more advanced and efficient approach to learning or as a tool that can be used in combination with other learning algorithms to boost the learning process. Is the second of these two the one we will explore along this project. We formally introduce DL in chapter 2.

Neither reinforcement learning nor deep learning algorithms are new. The basis of RL have their origin in the *Control Theory* proposed by Richard Bellman and others in the mid-1950s through extending a nineteenth century theory of Hamilton and Jacobi (see [1]). On the other hand, some sources point out that Frank Rosenblatt developed and explored all of the basic ingredients of the deep learning systems of today in his book *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms* [2] in the 1960s (see [3]). Is the combination of both methodologies, **Deep Reinforcement Learning** (DRL), the novelest and which arises important breakthroughs in the present.

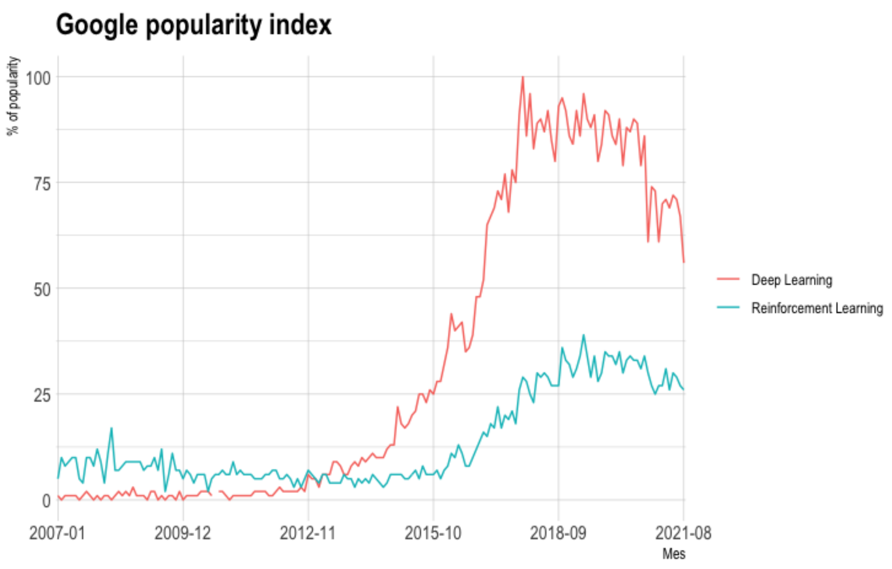
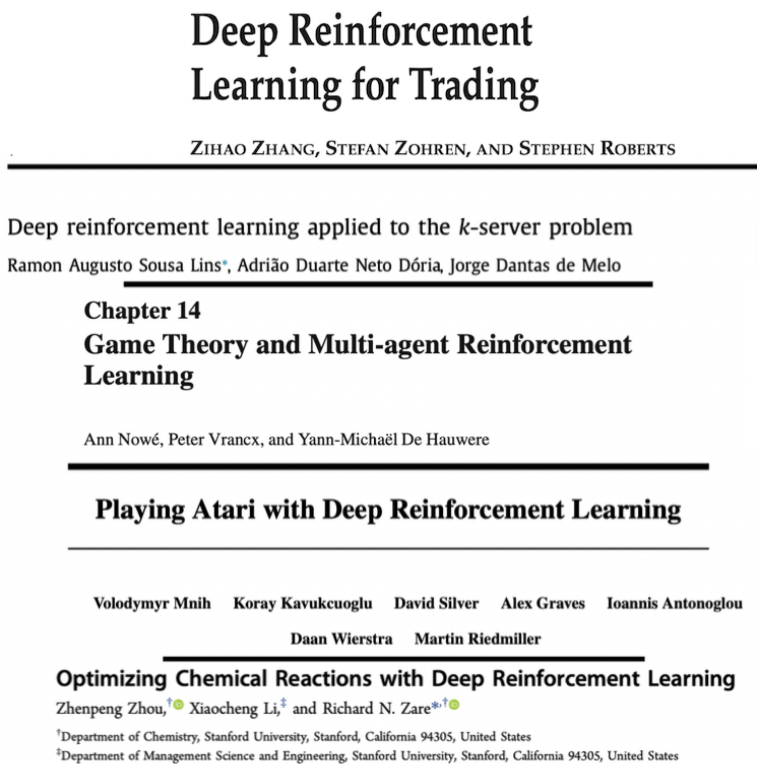


Figure 1.1: Popularity of the terms *reinforcement learning* and *deep learning* according to Google Trend index of popularity

In the Figure 1.1, one can advert that these two concepts has grown in popularity over the past years, using the Google Trend index of popularity. We can spot a considerable increment in the popularity index of both terms in the year 2015, which corresponds to the fame of the AlphaGo algorithm developed by the company, already acquired by Google, *DeepMind*. In October 2015, AlphaGo became the first Go machine to beat a professional Go player. The AlphaGo algorithm uses a combination of RL and DL techniques. The neural networks of the system were initially bootstrapped from the human gaming experience. AlphaGo was initially trained to mimic human play, trying to match the moves of expert players from recorded historical games, using a database of around 30 million moves. Once he had reached a certain degree of skill, he was further trained by being called upon to play a large number of matches against other instances of himself (see [4]). We will build the basic algorithm under this phenomena in chapter 5, that although simple, has been applied successfully to play ATARI games at a human mind level.

The outstanding results obtained by AlphaGo supposed a new resurgence of reinforcement learning and opened the doors to the application of these methodologies in many other fields. To cite some of them:

- Gaming: Atari games [5], go [6], poker [7] [8], . . .
- Autonomous driving [9] [10]
- Robotics [11]
- Natural language processing [12]
- Trading and finance [13] [14]
- Optimizing Chemical Reactions [15]
- Smart grids [16]
- News and products recommendation [17]


Deep Reinforcement Learning for Trading

ZIHAO ZHANG, STEFAN ZOHREN, AND STEPHEN ROBERTS

Deep reinforcement learning applied to the k-server problem

Ramon Augusto Sousa Lins*, Adrião Duarte Neto Dória, Jorge Dantas de Melo

Chapter 14

Game Theory and Multi-agent Reinforcement Learning

Ann Nowé, Peter Vrancx, and Yann-Michaël De Hauwere

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller

Optimizing Chemical Reactions with Deep Reinforcement Learning

Zhenpeng Zhou,[†] Xiaocheng Li,[‡] and Richard N. Zare*[†]

[†]Department of Chemistry, Stanford University, Stanford, California 94305, United States

[‡]Department of Management Science and Engineering, Stanford University, Stanford, California 94305, United States

Figure 1.2: Some articles of the current applications of Deep Reinforcement Learning

1.2 Motivation

As seen in the previous section, DRL plays an important role in the today field of machine learning, and many other applications, thanks to technological developments in the last era and the ability of computers to perform increasingly complex tasks. This is mainly due, as will be discussed in the respective chapter, for two reasons: is not necessary to know the dynamics of the environment (how it is structured) in order to apply RL algorithms and thanks to the deep approach we can face large-scale problems (in number of possible actions and states).

Personally, Reinforcement Learning Theory and its applications has a double interest for me:

- **Decision making and Game Theory**

It is evident that decision-making forms, perhaps not the "*whole*" of the human being, but the central axis of its development, of its maturity. We need to make decisions to move forward in our life. Maturing involves choosing, and therefore assuming, for more complex decisions that are interrelated with ourselves, with our experiences and our knowledge on the environment that surrounds us.

A first approach to the formalization of decision-making can be carried out by reviewing **Game Theory** (GT), so I did in my final degree project [18]. I studied the main models of GT and some of the solution algorithms related to those models, focusing on non-cooperative games/conflicts and its application to classic games as Poker. GT solutions, like *Nash equilibrium* or *backwards induction*, have the handicap that they cannot escalate easily to problems with many players or with a set of actions of large dimensions, although they are algorithms that seek an optimal strategy that provides the greatest utility. They also lie too much on the assumption that all players might behave *rationally* when facing a conflict.

Using Game Theory as a starting point, one can study the behavior of people when facing a conflict. This is achieved by allowing a game to be played repeatedly, learning from your own experience, *repeated games*. This is where Game Theory and Reinforcement Learning meet. Learning algorithms such as *Fictitious Player* or *Regret Matching* can be combined with the RL algorithms we will review in this project to obtain improvements in the results. In this case, DRL algorithms have already been applied (see [19]) to find an optimal policy to follow when playing a repeated game, even with multiple players or games of imperfect information.

- **Philosophical implications of the exploration vs exploitation conflict**

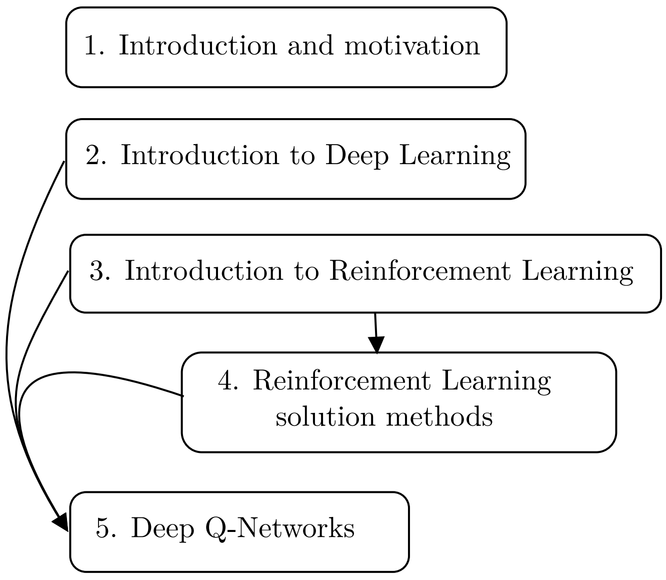
Inspired by behavioral psychology, reinforcement learning has many philosophical implications. Besides the fact that it is itself one of the branches of psychology, called *behaviorism*, there is an interesting struggle between two widely studied concepts in RL: **exploration** and **exploitation**.

Given the circumstance in which you have to make a decision, you can exploit the knowledge you have about the environment or you can explore it in search of options that will provide you with greater satisfaction. If you choose the best option, the action that gives you a better utility (measuring utility as you desire: money, satisfaction, power, etc.), you are exploiting your knowledge. If you choose an option that does not give you the best utility in the moment, but allows you to look for actions that would give you a greater cumulative reward in the future than the best option so far, you are exploring your environment.

Achieving this balance is not an easy task. If you tend to explore too much, you may get less than you expected or you may never find the best action at all. Also there are some scenarios where you can lose opportunity to choose the best action if you do not go for it from the very beginning. On the other hand, if you always exploit your best action at the end of the process you might get much less reward that you could have and you might regret it. One of the main problem in RL is to compute the optimal balance between exploration and exploitation, and of course, getting a optimal balance between this two concepts is very useful for life.

1.3 Project structure

The following chart summarizes the structure of the work, showing the itinerary to follow for the correct understanding of it.


1. Introduction and motivation
2. Introduction to Deep Learning
3. Introduction to Reinforcement Learning
4. Reinforcement Learning solution methods
5. Deep Q-Networks

I never think of the future - it comes soon enough

Albert Einstein

2 Introduction to Deep Learning

We will briefly introduce Deep Learning, from the point of view that will interest us in order to understand Deep Q-Networks, considering it as a form of supervised learning. We will give a general overview and expose the simplest neural network model. Neural networks have many possible topologies and can be used in a wide range of problems, a more detailed description of deep learning would imply reviewing concepts that go far beyond the objective of this project.

2.1 Supervised learning, bias and overfitting

In its most abstract form, supervised learning consists in finding a function $f : \mathcal{X} \to \mathcal{Y}$ based on a *training set* $(x_1, y_1), \dots, (x_n, y_n)$ for the purpose of approximating y at future observations of an input x. We assume that there is a function with noise $y = f(x) + \varepsilon$, where the noise (ε) has zero mean and variance σ^2.

A supervised learning algorithm can be viewed as a function that maps a dataset $\mathcal{D} = \{(x_1, y_1), \dots, (x_n, y_n)\}$ of learning samples $(x_i, y_i) \overset{\text{i.i.d.}}{\sim} (X, Y)$ into a model. The prediction of such a model at $x \in \mathcal{X}$ of the input space is denoted by $f(x; \mathcal{D})$. Given $\mathcal{D}$ and given a particular x, a natural measure of the effectiveness of f as a predictor of y is

$$E_{\mathcal{D}}[L(y - f(x; \mathcal{D}))],$$

where $L(\cdot, \cdot)$ is the loss function and $E[\cdot]$ means expectation with respect to the corresponding probability distribution. If $L(y, \hat{y}) = (y - \hat{y})^2$ is it called the mean-squared error. We want $(y - f(x; \mathcal{D}))^2$ to be minimal, both for $x_1, \dots, x_n$ (*training set*) and for points outside of our sample (*test set*). Of course, we cannot hope to do so perfectly, since the y_i contain noise ε_i; this means we must be prepared to accept an irreducible error in any function we come up with. For mean-squared loss function the error decomposes naturally into a sum of a **bias term** and a **variance term**.

$$\mathrm{E}_D\left[(y - f(x; D))^2\right] = (\mathrm{Bias}_D[f(x; D)])^2 + \mathrm{Var}_D[f(x; D)] + \sigma^2,$$

where

- $\mathrm{Bias}_D[f(x;D)] = \mathrm{E}_D[f(x;D)] - f(x)$.
- $\mathrm{Vax}_D[f(x;D)] = E_D\left[(\mathrm{E}_D[f(x;D)] - f(x;D))^2\right]$.
- σ^2 is the the irreducible error.

Proof. The derivation of the bias–variance decomposition for squared error proceeds as follows. First, recall that, by definition, for any random variable X, we have

$$\mathrm{Var}[X] = \mathrm{E}[X^2] - \mathrm{E}[X]^2.$$

Rearranging, we get

$$\mathrm{E}[X^2] = \mathrm{Var}[X] + \mathrm{E}[X^2]$$

and since f is deterministic, i.e. independent of D, $\mathrm{E}[f] = f$.

Thus, given $y = f + \varepsilon$ and $\mathrm{E}[\varepsilon] = 0$, implies $\mathrm{E}[y] = \mathrm{E}[f + \varepsilon] = \mathrm{E}[f] = f$.

Also, since $\mathrm{Var}[\varepsilon] = \sigma^2$,

$$\mathrm{Var}[y] = \mathrm{E}\left[(y - \mathrm{E}[y])^2\right] = \mathrm{E}\left[(y - f)^2\right] = \mathrm{E}\left[(f + \varepsilon - f)^2\right] = \mathrm{E}\left[\varepsilon^2\right] = \mathrm{Var}[\varepsilon] + \mathrm{E}[\varepsilon]^2 = \sigma^2 + 0^2 = \sigma^2.$$

Thus, since ε and $\hat{f} = f(x;D)$ are independent, we can write

$$\begin{aligned}
E\left[(y - \hat{f})^2\right] = E\left[(f + \varepsilon - \hat{f})^2\right] = \\
&= E\left[(f + \varepsilon - \hat{f} + E[\hat{f}] - E[\hat{f}])^2\right] = \\
&= E\left[(f - E[\hat{f}])^2\right] + E\left[\varepsilon^2\right] + E\left[(E[\hat{f}] - \hat{f})^2\right] + 2E[(f - E[\hat{f}])\varepsilon] + \\
&\quad + 2E[\varepsilon(E[\hat{f}] - \hat{f})] + 2\mathrm{E}[(E[\hat{f}] - \hat{f})(f - E[\hat{f}]) = \\
&= (f - E[\hat{f}])^2 + E\left[\varepsilon^2\right] + E\left[(E[f] - \hat{f})^2\right] + 2(f - E[\hat{f}])E[\varepsilon] + \\
&\quad + 2E[\varepsilon]E[E[\hat{f}] - \hat{f}] + 2E[E[\hat{f}] - \hat{f}](f - E[\hat{f}]) = \\
&= (f - E[\hat{f}])^2 + E\left[\varepsilon^2\right] + E\left[(E[\hat{f}] - \hat{f})^2\right] = \\
&= (f - E[\hat{f}])^2 + \mathrm{Var}[\varepsilon] + \mathrm{Var}[\hat{f}] = \\
&= \mathrm{Bias}[\hat{f}]^2 + \mathrm{Var}[\varepsilon] + \mathrm{Var}[\hat{f}] = \\
&= \mathrm{Bias}[\hat{f}]^2 + \sigma^2 + \mathrm{Var}[\hat{f}].
\end{aligned}$$

■

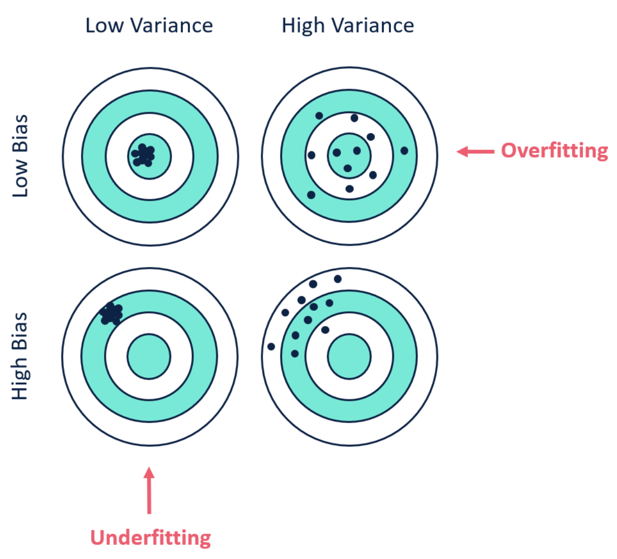


Figure 2.1: Bias vs variance

This bias-variance decomposition can be useful because it highlights a trade-off between an error due to erroneous assumptions in the model selection/learning algorithm (the bias) and an error due to the fact that only a finite set of data is available to learn that model (the parametric variance). The parametric variance is also called the overfitting error. The bias-variance trade-off is a central problem in supervised learning. Ideally, one wants to choose a model that both accurately captures the regularities in its training data, but also generalizes well to unseen data. Unfortunately, it is typically impossible to do both simultaneously. High-variance learning methods may be able to represent their training set well but are at risk of overfitting to noisy or unrepresentative training data. In contrast, algorithms with high bias typically produce simpler models that may fail to capture important regularities in the data.

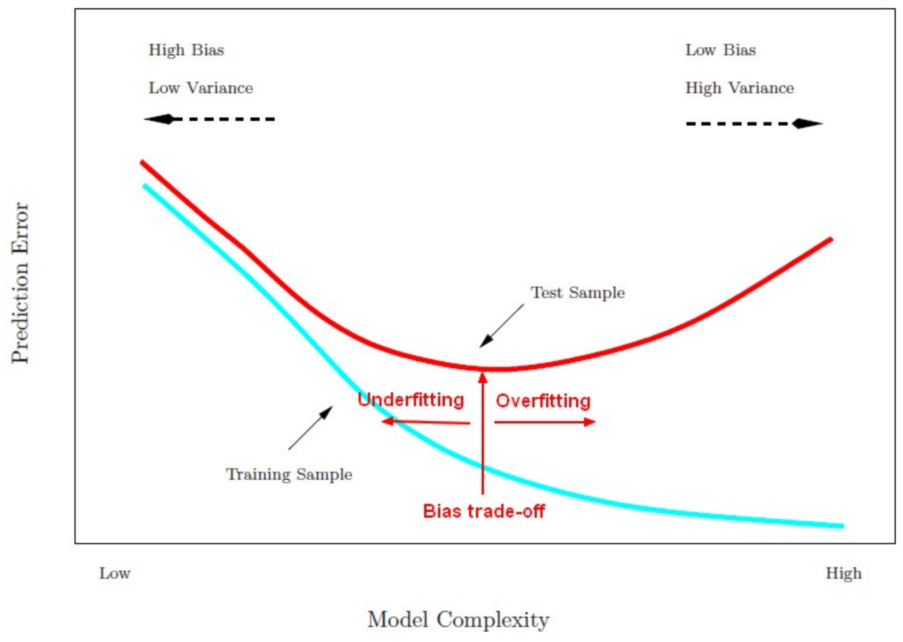


Figure 2.2: Bias-variance trade off

Even though there is no such direct decomposition for other loss functions, there is always a trade-off between a sufficiently rich model (to reduce the model bias, which is present even when the amount of data would be unlimited) and a model not too complex (so as to avoid overfitting to the limited amount of data).

2.2 Deep learning approach

Deep learning relies on a function $f : \mathcal{X} \to \mathcal{Y}$ parameterized with $\theta \in \mathbb{R}^{n_\theta}$ ($n_\theta \in \mathbb{N}$):

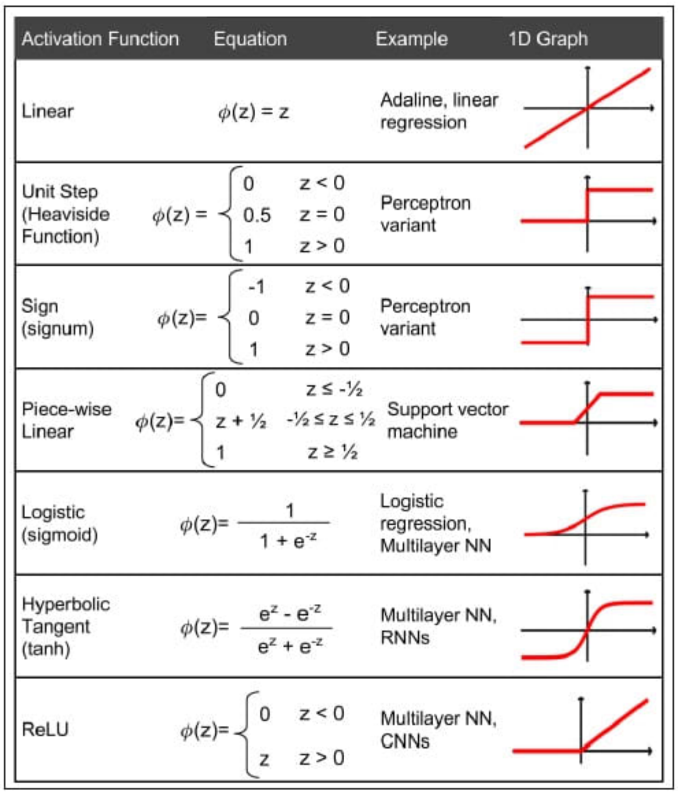
$$y = f(x; \theta).$$

A deep neural network is characterized by a succession of multiple processing layers. Each layer consists in a non-linear transformation and the sequence of these transformations leads to learning different levels of abstraction.

Lets describe the simplest neural network, the multilayer perceptron, with one fully connected hidden layer. In this case, the first layer is given the input values (i.e. the input features) x in the form of a column vector of size n_x ($n_x \in \mathbb{N}$). The values of the next hidden layer are a transformation of these values by a non-linear parametric function, which is a matrix multiplication by W_1 of size $n_h \times n_x$ ($n_h \in \mathbb{N}$), plus a bias term b_1 of size n_h, followed by a non-linear transformation:

$$h = \phi\left(W_1 \cdot x + b_1\right)$$

where ϕ is the activation function. The principal activation functions are shown in Figure 2.3.



Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z) = \begin{cases} 0 & z \le -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \le z \le \frac{1}{2} \\ 1 & z \ge \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	
Hyperbolic Tangent (tanh)	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

Figure 2.3: Activation functions

This non-linear activation function is what makes the transformation at each layer non-linear, which ultimately provides the expressivity of the neural network. The hidden layer h of size n_h can in turn be transformed to other sets of values up to the last transformation that provides the output values y. In this case:

$$y = (W_2 \cdot h + b_2),$$

where W_2 is of size $n_y \times n_h$ and b_2 is of size n_y ($n_y \in \mathbb{N}$). All these layers are trained to minimize the empirical error $I_S[f]$[1]. The most common method for optimizing the parameters of a neural network is based on gradient descent via the *backpropagation algorithm.* In the simplest case, at every iteration, the algorithm changes its internal parameters θ so as to fit the desired function:

$$\theta \leftarrow \theta - \alpha \nabla_\theta I_S[f],$$

where α is the learning rate. We will expand the explanation of gradient descent algorithm adjusting it to our interests in chapter 5, studying the case of *stochastic gradient descent.*

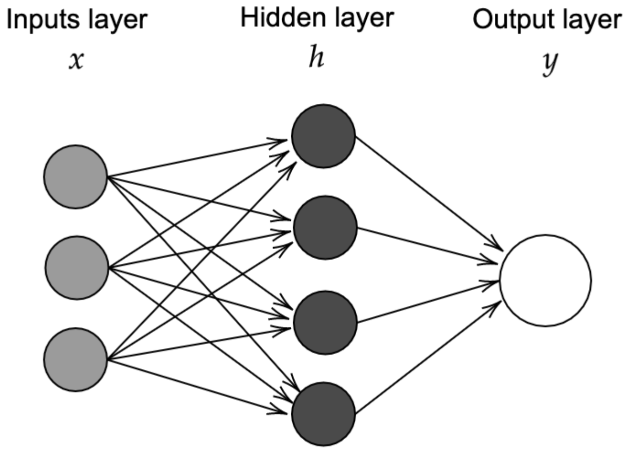


Figure 2.4: Neural network with one hidden layer

[1]Given n data points (x_i, y_i), the empirical error is $I_S[f] = \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i))$.

2.3 Convolutional Neural Networks

As we will understand in the last chapter, learning in Atari games would be performed using pixels as primary element. Due to this, the neural network model that suits our purpose is the **convolutional neural network** (CNN)[2], which is widely used in computer vision and image pattern recognition.

CNN inputs are images and use the convolution operation on at least one of their layers. A convolution is a mathematical operation on two functions that expresses how the shape of one is modified by the other.

Definition 2.3.1 — Convolution.

The convolution of f and g is written $f * g$. It is defined as the integral of the product of the two functions after one is reversed and shifted.

$$(f * g)(t) \doteq \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\, d\tau$$

or equivalently shifting f and g role in the definition because of commutativity. The convolution formula can be described as the area under the function $f(x)$ weighted by the function $g(-x)$ shifted by amount t. If f and g are defined on the set $\mathbb{Z}$ of integers, the discrete convolution of f and g is given by:

$$(f * g)(n) \doteq \sum_{m=-\infty}^{\infty} f(m)g(n-m).$$

Although not essential, before feeding the network, it is appropriate to normalize the input values. The colors of the pixels have values that go from 0 to 255, we can make a transformation of each pixel $value/255$ and we will always have a value between 0 and 1.

Basically, it can be said that CNNs are composed of three types of layers:

- **Convolutional layers**, which play an extremely important role performing the convolution operations.
- **Pooling layers**, which is responsible for reducing, among others, the dimensionality of the network, the number of parameters and the complexity of the model.
- **Fully connected layers**, which contain neurons that are directly connected to the two immediately adjacent layers.

Convolutional layer

It requires a few components, which are **input data**, a **filter**, and a **feature map**.

The input data is given by a dimensional matrix. If for example the input is a color image, the the input matrix representing pixels would have 3 dimensions (three colors of the RGB color representation). The *feature detector*, *kernel* or *filter*, will move across the image, proceeding with the convolution, which can be interpreted as checking if the feature is present. The filter is then applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterwards, the filter shifts

[2]This is not the only way to use deep learning in RL, but is so far the basis we will need to understand the Deep Q-Networks and to apply it to our case example.

by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products from the input and the filter is known as a feature map.

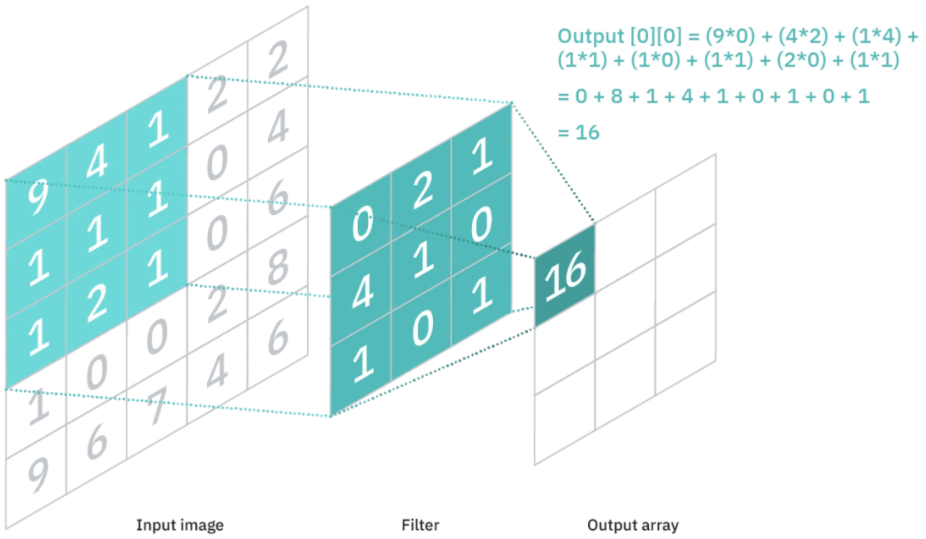

Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
Input image
Filter
Output array

Figure 2.5: Example of filter application to the input in a CNN.

In reality, we will not apply only a kernel, but we will have many kernels (their set is the one called filters). Therefore, there are some parameters that must be tuned during the process: the *number of filters*, the *stride* (distance, or number of pixels, that the kernel moves over the input matrix) and the *padding*. Zero-padding is usually used when the filters do not fit the input image. This sets all elements that fall outside of the input matrix to zero, producing a larger or equally sized output. There are three types of padding:

- *Valid padding* or *no padding*. In this case, the last convolution is dropped if dimensions do not align.
- *Same padding*. This padding ensures that the output layer has same size as the input layer.
- *Full padding*. This padding increases the size of the output by adding zeros to the border of the input.

After each convolution operation, a CNN applies transformation to the feature map, introducing non-linearity to the model through an activation function (see Figure 2.3). The most used activation function for this type of neural network is the *ReLu*.

Pooling layer

Pooling is an operation that reduces the number of network parameters. While convolutional layers can be followed by additional convolutional layers, if we made a new convolution directly from the feature map, the number of neurons in the next layer would highly increase and this implies more processing.

To reduce the size of the next layer of neurons, we will do a subsampling process in which we will reduce the size of our filtered images, but where the most important characteristics detected by each filter should prevail. Pooling layers, also known as *downsampling*, conducts dimensionality reduction of the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field.

There are different types of pooling layers:

- **Max pooling**. As the filter moves across the input, it selects the pixel with the maximum value to send to the output array.
- **Average pooling**: As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

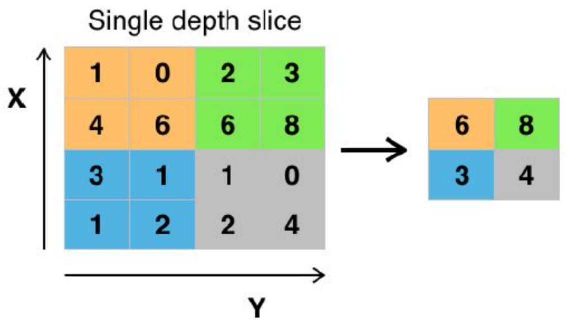


Figure 2.6: Example of max pooling in a CNN

Fully connected layer

Since the output array does not need to map directly to each input value, convolutional (and pooling) layers are commonly referred to as *partially connected* layers. In the last fully-connected layers, each node in the output layer connects directly to a node in the previous layer. This type of layer is exactly the same as any layer of a classic artificial neuron network. The main function of the fully connected layers is to carry out a kind of grouping of the information that has been obtained up to that moment, which will serve in later calculations for the final classification.

The last of these layers, also known as **loss layer**, will have the parameter K which is the number of classes that are present in the data set. The final values of K will be fed to the output layer, which through a certain probabilistic function will perform the classification. This layer performs the task of classification based on the features extracted through the previous layers and their different filters, producing a probability from 0 to 1 computed by certain function. When it comes to classifying and choosing between K possible levels, a *softmax loss classifier* would be used. The use of a *Euclidean function* is also common for the purpose of regression against image labels.

- *Softmax loss function*:
$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}},$$
where
  - $z_j = \sum_{k=0}^{d} W_{ik} x_k$ is a vector of posterior probabilities.
  - j represents the ith neuron of the output layer, that is, of the loss layer.
  - K represents the total number of neurons in the loss layer.
  - W_{ik} are the weights.
  - x_i are the input values received by the loss layer.
  - $\sigma(z)_j$ is the activation of the K neurons of the loss layer.

- *Euclidean loss function*:
$$E = \frac{1}{2K} \sum_{i=1}^{K} \|\hat{y}_i - y_i\|_2^2 \,,$$
where
  - K represents the total number of neurons in the loss layer.
  - $\hat{y}_i$ represents the predictions of the images.
  - y_i represents the actual values of the images.

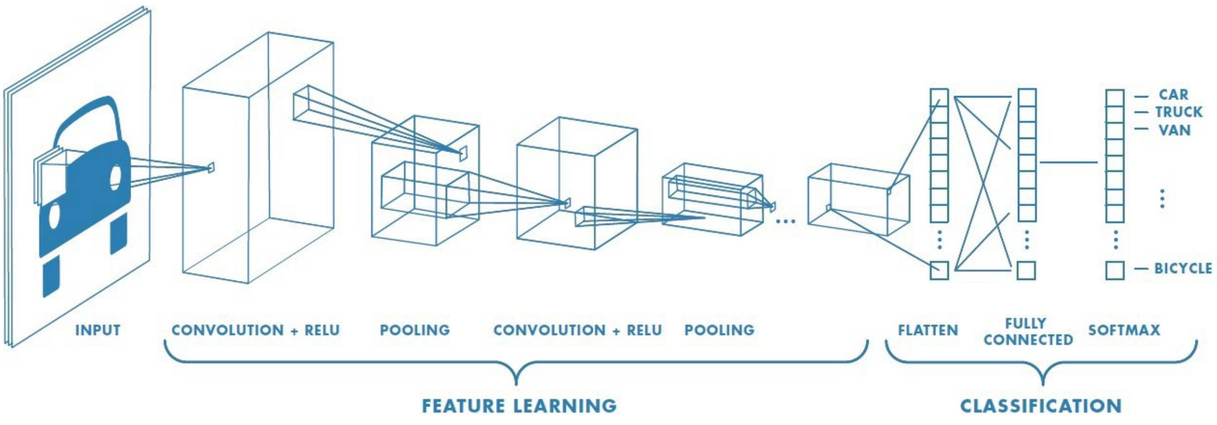


Figure 2.7: Architecture of a CNN

To become good at anything you have to know how to apply basic principles.
To become great at it, you have to know when to violate those principles.

Garry Kasparov

3 Introduction to Reinforcement Learning

3.1 Basic concepts

Reinforcement learning (RL) is the computational approach to learning from interaction, an agent interacts with its environment and learn to improve its behavior through being given subsequent rewards. RL is learning how to map situations to actions so as to maximize a numerical reward signal, following the scheme of a trial-and-error process with a delayed reward, positives or negative, after each trial. The elements of reinforcement learning are the following:

- **Agent**: An agent takes actions that reflects its decisions.
- **Action** (A): is the possible moves the agent has to choose. It should be noted that agents usually choose from a discrete set possible actions.
- **Environment**: The world through which the agent moves, and which responds to the agent. The environment takes the current state and action of the agent as input, and returns as output the reward and its next state.
- **State** (S): A state is a concrete and immediate situation in which the agent finds itself in teh environment. It can the current situation returned by the environment, or any future situation.
- **Reward** (R): A reward is the feedback by which we measure the success or failure of an agent's actions in a given state. Rewards can be immediate or delayed and they effectively evaluate the agent's action.
- **Policy** (π): The policy is the strategy that the agent employs to determine the next action based on the current state. It maps states to actions, moving towards the actions that promise the highest reward.
- **Value function**: The expected long-term return, as opposed to the short-term reward. There are two types of value function: only for states and for state-action pairs.
- **Model of the environment**. This is something that simulate the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. Given a state and action, the model might predict the resultant next state and next reward. Models are used for *planning*.

The agent and the environment interact in a sequence of discrete times $t = 0, 1, 2, \ldots$. At each step, the agent receives some representation of the environment state $S_t \in \mathcal{S}$ and on that basis selects an action $A_t \in \mathcal{A}(s)$, or$\mathcal{A}$ if the set of actions is the same in all states. In the next time step $t + 1$, as consequence of its action, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and finds itself in a new state S_{t+1}[1].

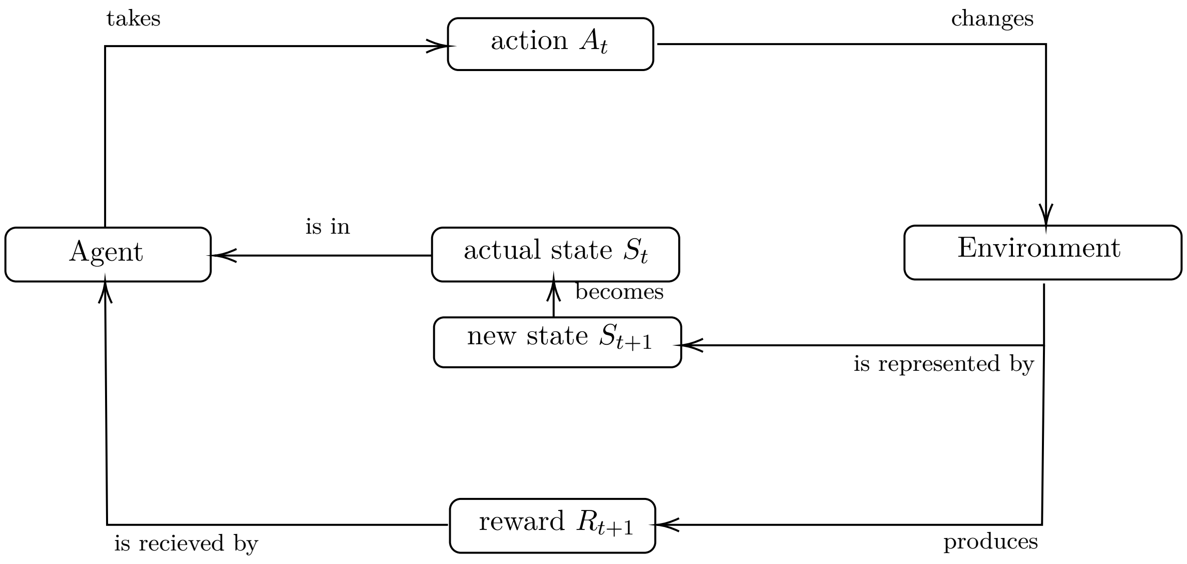


Figure 3.1: Agent–environment interaction

To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to *exploit* what it has already experienced in order to obtain reward, but it also has to *explore* in order to make better action selections in the future. This is known has the **exploitation-exploration trade off**. Actions choices are then made based on value judgments, we seek actions that bring states of highest values, not highest reward.

As one can see, the **feedback** the agent receives is truly important in RL. The feedback can be received in two ways:

- **Evaluative feedback**: indicates how good the action taken was, but not whether it was the best or the worst action possible.
- **Instructive feedback**: indicates the correct action to take, independently of the action actually taken.

We also distinguish between two types of tasks at first:

- **Non-associative tasks**: tasks in which there is no need to associate different actions with different situations, there is no notion of such states considered.
- **Associative tasks**: in general, in RL tasks you receive a signal and you choose your policy based on that association.

[1]The reward is given when leaving the state not at the beginning of it, that is the reason why to action A_t it corresponds reward R_{t+1}.

■ **Example 3.1 — Bandits problem.**

In this example you are asked to pull one lever of the casino bandits at a time over k possible levers, $a \in \{1, \ldots, k\}$.

- $A_t \equiv$ lever selected on time step t.
- $R_{t+1} \equiv$ price won when selecting lever A_t.

In the simplest bandits problem, we only have actions and rewards, thus there are no states to consider, and it is a *non-associative task*. Now lets suppose we have a 2-armed bandit, with two levers, and when pulling one lever we can find ourselves in two cases as show in Figure 3.1. If, somehow, we are told in which case we are before pulling the lever, then we will be able to make the following associations:

- *If case A then pull lever 2* (thus get reward of 20).
- *If case B then pull lever 1* (thus get reward of 90).

This is called **contextual bandits problem**, an it is an example of *associative task*, where the states can be considered the different cases.

■

	lever 1	lever 2
case A	10	20
case B	90	80

Table 3.1: 2-armed bandit example

Another important concept to consider is **stationarity**. Stationary dynamics refers to the environment, and states that the rules of the environment do not change over time. The rules of the environment are often represented as an MDP model, which consists of all the state transition probabilities and reward distributions. The stationary assumption asserts that these probabilities do not change over time. Most of RL problems are non-stationary.

3.2 Markov Decision Process

In advance we employ notation and main formalization of the concepts from [1]. We formally define a Markov Decision Process as

Definition 3.2.1 — Markov Decision Process (MDP).

An MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$ where:

- S is the state space.
- A is the action space.
- $p : S \times \mathcal{R} \times \mathcal{A} \times S \to [0, 1]$ is the transition function.
- $r : S \times A \times S \to \mathcal{R}$ is the reward function, where $\mathcal{R}$ is a continuous set of possible rewards in a range $R_{\max} \in \mathbb{R}^+$.
- $\gamma \in [0, 1)$ is the discount factor.

In a finite MDP, $\mathcal{S}$, $\mathcal{A}$ y $\mathcal{R}$ all have a finite number of elements.

We will suppose the random variables R_t and S_t have well defined discrete probabilities distribution, depending on the preceding state and action.

$$
\begin{aligned}
&p : S \times \mathcal{R} \times \mathcal{A} \times S \to [0,1] \\
&p\left(s', r \mid s, a\right) \doteq \Pr\left\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right\}, \quad \forall s, s' \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A}(s),
\end{aligned}
$$

such as $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) = 1,$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$. The probabilities given by p completely characterize the environment dynamics, the states. The state must include information about all aspects of the past agent-environment interaction that make a difference for the future. We will say that a state is a Markov state if it satisfy the Markov property:

$$
P\left[S_{t+1} \mid s_t\right] = P\left[S_{t+1} \mid s_0, \ldots, s_t\right].
$$

Using the joint probabilities defined before, we have the following probability functions that we will be using for the rest of the project:

- **State-transition probabilities**

$$
\begin{aligned}
&p : S \times S \times A \to [0,1] \quad \text{(abuse of notation)}, \\
&p\left(s' \mid s, a\right) \doteq \Pr\left\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\right\} = \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right).
\end{aligned}
$$

- **Expected rewards for state-action pairs**

$$
\begin{aligned}
&r : S \times \mathcal{A} \to \mathbb{R}, \\
&r(s,a) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right] = \sum_{r \in \mathcal{R}} r \sum_{s' \in S} p\left(s', r \mid s, a\right).
\end{aligned}
$$

- **Expected rewards for state action-next-state**

$$
\begin{aligned}
&r : S \times \mathcal{A} \times S \to \mathbb{R}, \\
&r\left(s, a, s'\right) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'\right] = \sum_{r \in \mathcal{R}} r \frac{p\left(s', r \mid s, a\right)}{p\left(s' \mid s, a\right)}.
\end{aligned}
$$

3.3 Goals

The rewards indicate how well an agent is doing at a step t, but, in particular, the reward signal is not the place to impart to the agent prior knowledge about how to achieve what we want it to do. For example, a chess playing agent should be rewarded only for actually winning, not for achieving subgoals such of taking its opponents pieces or gaining control of the center of the board. If achieving these subgoals were rewarded, then the agent might find a way to achieve them, without achieving the real goal, e.g. it might find a way to take the opponent's pieces even at the cost of losing the game.

How do we formalize this goal? Using the concept of **return**.

Definition 3.3.1 — Reward hypothesis. All goals can be described by the maximization of the expected cumulative reward or return.

Let denote the sequence of rewards received after time step t by $R_{t+1}, R_{t+2}, R_{t+3}, \dots, R_T$. We shall then differentiate two cases:

- **Episodic tasks** ($T < \infty$). Agent-environment interaction breaks naturally into subsequences called *episodes* or *trials* (interactions from initial to end states). For this tasks we distinguish between the set of non-terminal state, denoted by $\mathcal{S}$, and the set of terminal states, $\mathcal{S}^+$. An episodic task last a finite amount of time, $T \equiv$ time of termination. T is a random variable that normally varies from episode to episode. In episodic tasks there might also be only a single reward at the end of the task, and one option is to distribute the reward evenly across all actions taken in that episode.

$$G_t \doteq R_{t+1} + R_{t+2} + \cdots + R_T. \tag{3.1}$$

- **Continuous tasks** ($T = \infty$). There is not a terminal state, continuous tasks will never end. Therefore, as the tasks might never end, the return expressed as 3.1 could be infinite. We use then a *discount factor*, more recent reactions receive greater reward and actions long time in the past receive smaller rewards.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{3.2}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate.

If $\gamma < 1$, the infinite sum 3.2 has a finite value as long as the reward sequence is bounded. For example, if the reward is a constant +1, the return is $G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$. If $\gamma = 0$, the agent is *myopic* in being concerned only with maximizing immediate rewards. On the other hand, if $\gamma \approx 1$, the agent is considered to be *far-sighted.*

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned}$$

Problems might arise when distinguishing between episodic and continuous tasks.

■ **Example 3.2 — Pole-balancing.**

Imagine we have a situation similar to Figure 3.2, where a toy car can be moved right and left to get a pole balanced. The objective is to avoid the pole to fall. The pole is reset to vertical after each failure.

We can see this game as an episodic tasks where episodes are the attempts to balance the pole. A reward of +1 is given every time on which failure does not happens. Return at each time would be the number of steps until failure. In this case successful balancing forever would have an infinite return. So we might see it has a continuous task where the reward should be -1 on failure and 0 in other case. In either cases maximized return is obtained by keeping balance, but returns are different.

■

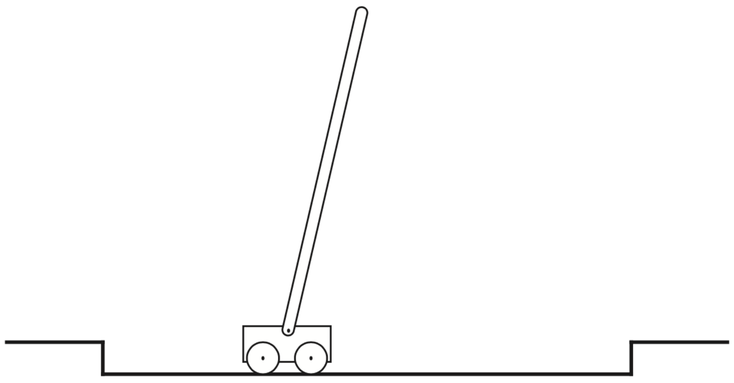


Figure 3.2: Pole balancing example

We need therefore a unified notation for episodic and continuous tasks. We consider **absorbing states** for episodic tasks, a state that once entered cannot be left, and we define a general concept of return for both:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k,$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both).

3.4 Policies and value functions

Value functions for states (or for state-action pairs) estimate how good it is for the agent in terms of expected return to be in a given state (or how good is to perform a given action in a given state). Value functions are defined depending on particular ways to act, which are known as **policies**. RL methods will specify how the agent's policy is changed as a result of its experience.

Definition 3.4.1 — Policy.

A **policy** is a mapping from states to probabilities of selecting each possible action. We find two types of policies:

- **Deterministic policy:** $\pi(s) = a$.
- **Stochastic policy:** $\pi(s \mid a) = P[A_t = a \mid S_t = s]$.

Given a policy π we formally define value functions as follow:

- **State value function for policy π or value function**

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right], \text{ for all } s \in \mathcal{S}.$$

The value of a terminal state, if any, is defined as 0.

- **Action-value function for policy π or Q-function**

$$q_\pi(s,a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right].$$

v_π and q_π can be estimated (Monte Carlo methods) or approximate (approximate solution methods)

3.4.1 Bellman equations

Bellman equation helps us to find optimal policies and evaluate states and actions. It claims that the value of a state (or state-action pair) is the sum of the immediate reward and the discounted value of successor states.

- **Bellman equation for state value functions**

$$v_\pi(s) = E_\pi\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) \mid S_t = s\right].$$

- **Bellman equation for action-state value function**

$$q_\pi(s,a) = E_\pi\left[R_{t+1} + \gamma q_\pi\left(S_{t+1}, A_{t+1}\right) \mid S_t = s, A_t = a\right].$$

We can give a more treatable expression with the help of back-up diagrams. For the state value function we consider the following diagram

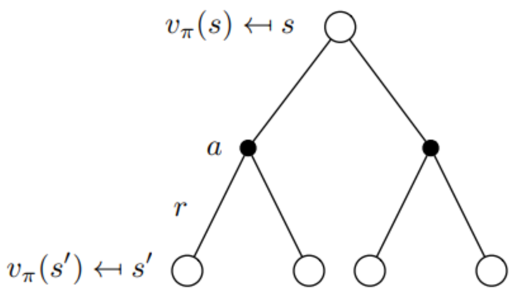

$v_\pi(s) \leftarrow\!\shortmid s$
a
r
$v_\pi(s') \leftarrow\!\shortmid s'$

Figure 3.3: Back up diagram for state value functions

We can evaluate how good is state s following policy π:

$$v_\pi(s) = \sum_{a\in A} \pi(a \mid s) q_\pi(s,a).$$

How good is then to take action a?

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s'\in S} p\left(s' \mid s,a\right) v_\pi(s').$$

Combining both expressions we get

$$\begin{aligned} v_\pi(s) &= \sum_{a\in A} \pi(a \mid s)(r(s,a) + \gamma \sum_{s'\in S} p(s' \mid s,a) v_\pi(s')) = \\ &= \sum_{a\in A} \pi(a \mid s)[\sum_{r\in R} r \sum_{s'\in S} p(s', r \mid s,a) + \gamma \sum_{s'\in S}\sum_{r\in R} p(s', r \mid s,a) v_\pi(s')] = \\ &= \sum_{a\in A} \pi(a \mid s) \sum_{s',r} p(s', r \mid s,a)[r + \gamma v_\pi(s')]. \end{aligned}$$

Similarly, for state-action value functions we have the following back up diagram and expression

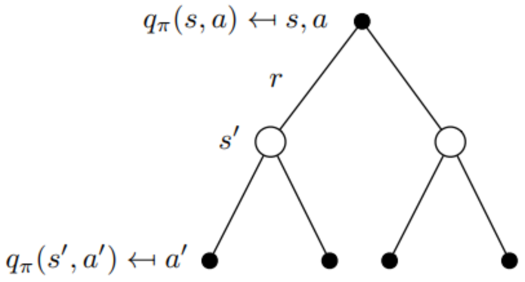


Figure 3.4: Back up diagram for action-state value functions

$$\begin{aligned} q_\pi(s,a) &= r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) \sum_{a \in A} \pi(a' \mid s') q_\pi(s',a') = \\ &= \sum_{r \in R} r \sum_{s' \in S} p(s', r \mid s,a) + \gamma \sum_{s',r} p(s',r \mid s,a) \sum_{a \in A} \pi(a' \mid s') q_\pi(s',a') \\ &= \sum_{s',r} p(s',r \mid s,a)[r + \gamma \sum_{a \in A} \pi(a' \mid s') q_\pi(s',a')] \\ &= \sum_{s',r} p(s',r \mid s,a)[r + \gamma v_\pi(s')]. \end{aligned}$$

3.5 Behaving optimally

Solving a RL tasks is to find a policy that achieves maximum reward in the long run. We can define a partial ordering over policies: π is better than π' if tis expected returns are greater or equal to the π' for all states

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'(s)}, \quad \forall s \in \mathcal{S}.$$

Under this conditions, there always exists a policy that is better or equal tha the others, the **optimal policy** π_*.

Theorem 3.5.1 For any MDP,

- There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \ \forall \pi$.
- All optimal policies achieve the optimal value function $v_{\pi_*}(s) = v_*(s)$.
- All the optimal policies achieve the optimal action state-value function $q_{\pi_*}(s,a) = q_*(s,a)$.

These optimal functions are defined as follow:

- **Optimal state value function**: v_* tell us what is the maximum reward we can get from the system.

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

- **Optimal state-action value function**: q_* tell us maximum reward we are going to get if we are in state s and take action a.

$$q_*(s,a) = \max_\pi q_\pi(s,a), \quad \forall s \in \mathcal{S}, a \in \mathcal{A}(s).$$

We can write q_* in terms of v_*:

$$q_*(s,a) = E\left[R_{t+1} + \gamma v_*\left(s_{t+1}\right) \mid S_t = s, A_t = a\right].$$

3.5.1 Bellman optimality equations

The Bellman optimality equations express the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a E_{\pi_*}\left[G_t \mid S_t = s, A_t = a\right] \\ &= \max_a E_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\ &= \max_a E\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) \mid S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_*\left(s'\right)\right]. \end{aligned}$$

$$\begin{aligned} q_*(s,a) &= E\left[R_{t+1} + \gamma \max_{a'} q_*\left(S_{t+1}, a'\right) \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \max_{a'} q_*\left(s', a'\right)\right]. \end{aligned}$$

For finite MDPs, the Bellman optimality equation for v_* has unique solution. The Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns. If the dynamics of the environment are know, then one can solve this system of equations for v_* using a variety of method for solving non-linear systems. One can solve a related system for q_*. The running time complexity for this computation is $O(n^3)$ [1]. Therefore, this is clearly not a practical solution for solving larger MDPs. In later sections we will use more efficient methods like Dynamic Programming, Monte Carlo methods and TD Learning.

■ **Example 3.3 — Recycling Robot.** A mobile robot has the job of collecting trash of the environment. Decisions are made following the battery level:

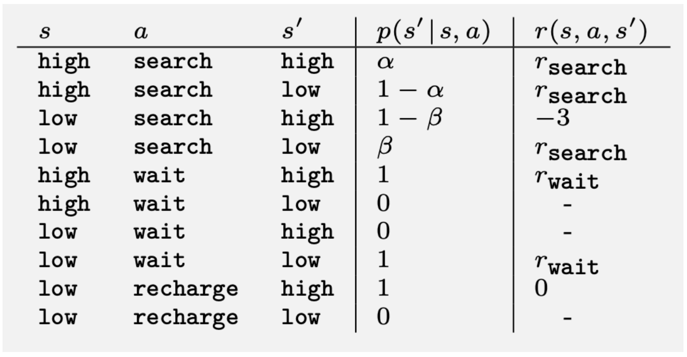
- $\mathcal{S} = \{\texttt{high battery level}, \texttt{low battery level}\}$.
- $\mathcal{A}(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$.
- $\mathcal{A}(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$.

The transition probabilities and rewards for each state-action-next-state triple are shown in Figure 3.5 and the MDP graph, for better understanding the dynamics of the environment, is in Figure 3.6. We assume that $r_{\texttt{search}} > r_{\texttt{wait}}$.

Using the transition probabilities and reward we can give the Bellman optimality equation expression for value function. Let's denote $h \equiv \texttt{high}$, $l \equiv \texttt{low}$, and $s \equiv \texttt{search}$, $w \equiv \texttt{wait}$, $re \equiv \texttt{recharge}$. For any choice of r_s, r_w, α, β and γ such as $0 \le \alpha, \beta, \gamma \le 1$ and $r_w \le r_s$, there is exactly one pair of numbers $v_*(h), v_*(l)$ that satisfies:

$$
\begin{aligned}
v_*(h) &= \max \left\{ \begin{array}{l} p(h \mid h,s)\left[r(h,s,h) + \gamma v_*(h)\right] + p(l \mid h,s)\left[r(h,s,l) + \gamma v_*(l)\right], \\ p(h \mid h,w)\left[r(h,w,h) + \gamma v_*(h)\right] + p(l \mid h,w)\left[r(h,w,l) + \gamma v_*(l)\right] \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} \alpha\left[r_s + \gamma v_*(h)\right] + (1-\alpha)\left[r_s + \gamma v_*(l)\right], \\ 1\left[r_w + \gamma v_*(h)\right] + 0\left[r_w + \gamma v_*(l)\right] \end{array} \right\} \\
&= \max \left\{ \begin{array}{l} r_s + \gamma\left[\alpha v_*(h) + (1-\alpha) v_*(l)\right], \\ r_w + \gamma v_*(h) \end{array} \right\}.
\end{aligned}
$$

$$
v_*(l) = \max \left\{ \begin{array}{l} \beta r_s - 3(1-\beta) + \gamma\left[(1-\beta) v_*(h) + \beta v_*(l)\right], \\ r_w + \gamma v_*(l), \\ \gamma v_*(h). \end{array} \right\}
$$



s	a	s'	$p(s' \mid s,a)$	$r(s,a,s')$
high	search	high	α	$r_{\texttt{search}}$
high	search	low	$1-\alpha$	$r_{\texttt{search}}$
low	search	high	$1-\beta$	-3
low	search	low	β	$r_{\texttt{search}}$
high	wait	high	1	$r_{\texttt{wait}}$
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	$r_{\texttt{wait}}$
low	recharge	high	1	0
low	recharge	low	0	-

Figure 3.5: Recycling Robot transition probabilities and rewards

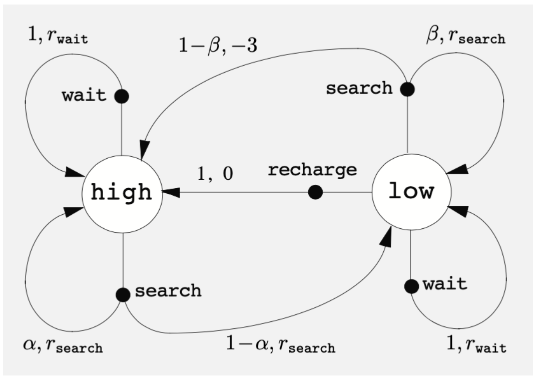


Figure 3.6: Recycling Robot MDP graph

Solving a RL problem can be seen as giving the solution to the Bellman optimality equation. This solution relies on at least three assumptions that are rarely true in practice:

1) Accurate knowledge of the environment dynamics.
2) Computational power.
3) Markov property.

For example, although the first and third assumptions present no problems for the game of Backgammon, the second is a mayor impediment, because game has 10^{20} states (see [20]).

We do not know what the rules of the game are; all we are allowed to do is to watch the playing. Of course, if we watch long enough, we may eventually catch on to a few of the rules.

Richard P. Feynman

4 Reinforcement Learning solution methods

In this chapter we will review the main models exposed in [1], unifying the notation and following the same structure for all methods to facilitate its understanding. The statements of the examples are also taken from [1].

4.1 Dynamic programming

Assuming the environment is a finite MDP, the key idea of dynamic programming (DP) is the use of value functions to organize the search for good policies. The requirements for DP are:

- **Optimal substructure.** The problem can be divided into subproblems which hold the principle of optimality.
- **Overlapping subproblems.** The solution of a subproblem is saved and then used to solve similar subproblems.

MDP satisfy both of these properties:

$\rightarrow$ *Optimal substructure.* Bellman equation is defined as $v(s) = E\left[R_{t+1} + \gamma v\left(S_{t+1}\right) \mid S_t = s\right]$, this is, the value of a state is equal to the immediate reward our agent gets leaving the state plus the value of the next state, so the equation is breaking down the process of finding the value function of a state by dividing it into subproblems.

$\rightarrow$ *Overlapping subproblems.* Value functions have already stored how good a particular state is so we do not need to recompute the value of that state again and again.

DP assumes full knowledge of MDP, for this reason we will study methods that avoid this assumption in further sections.

In general, a RL task is broken into two parts:

- **Prediction.** Evaluate the future given a policy. Suppose we have a MDP defined as $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$, given a policy π we want to find the value function v_π.
- **Control.** Optimize the future, find the best policy. This involves optimizing the value function to use it to find an optimal policy.

Although our final goal will be getting control methods (which might include a prediction part), the reason why differentiating between these two procedures is because most models differs on prediction and use similar control methods.

4.1.1 Prediction

Recall, $v_\pi = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma v_\pi(s')]$.

Consider a sequence of approximate value functions $v_0, v_1, v_2, \ldots$ each mapping $\mathcal{S} \to \mathbb{R}$. The initial approximation v_0 is chosen arbitrarily, except for the terminal state, if any, that must be given value 0. For the rest of states we follow the **iterative policy evaluation rule**:

$$v_{k+1}(s) \doteq E_\pi \left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s\right] = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right]. \quad (4.1)$$

Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures the equality in this case. The sequence $\{v_k\}$ can be shown to converge to v_π under the same conditions that guarantees the existence of v_π, i.e. $\{v_k\} \underset{k\to\infty}{\longrightarrow} v_\pi$.

In order to formalize a prediction algorithm using dynamic programming formulation we need a stoppage condition. We can compute $\max_{s\in\mathcal{S}} |v_{k+1}(s) - v_k(s)|$ and stop if it is small enough.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

```
Input $\pi$, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
       $v \leftarrow V(s)$
       $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
```

4.1.2 Control

We know how good is to follow a given policy π (v_π), but would it be better to change to a new policy? An idea is to consider deterministic policies, policies that gives full probability to an action a, $\pi'(s) = a$. The true value of behaving like this should then be computed using $q_\pi(s, a)$.

Theorem 4.1.1 — Policy improvement theorem. Let π, π' be any pair of deterministic policies such that $\forall s \in S$

$$q_\pi(s, \pi'(s)) \geqslant v_\pi(s).$$

Then the policy π' must be as good as, or better than π. That is, it must obtain greater or equal expected return $\forall s \in \mathcal{S}$

$$v'_\pi(s) \geqslant v_\pi(s).$$

Proof.

$$\begin{aligned}
v_\pi(s) &\leq q_\pi\left(s, \pi'(s)\right) = q_\pi(s, a) \\
&= E\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) \mid S_t = s, A_t = \pi'(s)\right] \\
&= E_{\pi'}\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) \mid S_t = s\right] \\
&\leq E_{\pi'}\left[R_{t+1} + \gamma q_\pi\left(S_{t+1}, \pi'\left(S_{t+1}\right)\right) \mid S_t = s\right] \\
&= E_{\pi'}\left[R_{t+1} + \gamma \mathbb{E}\left[R_{t+2} + \gamma v_\pi\left(S_{t+2}\right) \mid S_{t+1}, A_{t+1} = \pi'\left(S_{t+1}\right)\right] \mid S_t = s\right] \\
&= E_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi\left(S_{t+2}\right) \mid S_t = s\right] \\
&\leq E_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi\left(S_{t+3}\right) \mid S_t = s\right] \\
&\vdots \\
&\leq E_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \mid S_t = s\right] \\
&= v_{\pi'}(s).
\end{aligned}$$

■

For states other than s, equation holds because two sides are equal.

If $q_\pi(s, \pi'(s)) > v_\pi(s)$ then π' is better than π. Thus we want to select the best possible $q_\pi(s, a)$ at each step, this is **acting greedy** at each step. The **policy improvement rule** is build then following the **greedy policy** on the state-action value function[1]:

$$\begin{aligned}
\pi'(s) &\doteq \underset{a}{\operatorname{argmax}}\ q_\pi(s, a) \\
&= \underset{a}{\operatorname{argmax}}\ E\left[R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) \mid S_t = s, A_t = a\right] \\
&= \underset{a}{\operatorname{argmax}} \sum_{s', r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right].
\end{aligned}$$

Suppose the new policy π' is as good as π but not better, $v_\pi = v_{\pi'}$. In this case we get the Bellman optimality equation and therefore $v_{\pi'}$ must be v_* and π, π' are optimal. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

We call General Policy Iteration (GPI) the alternation between policy evaluation and policy iteration.

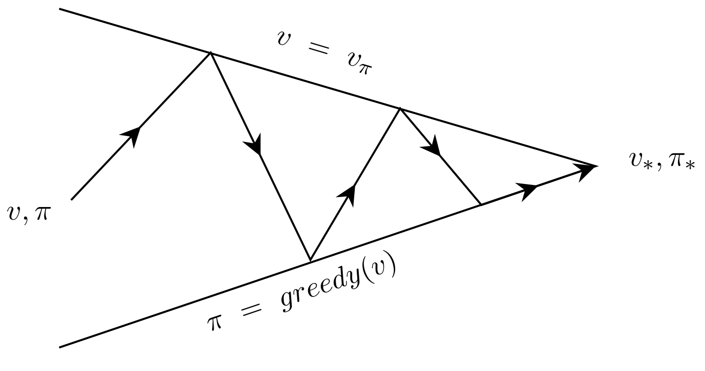


Figure 4.1: Convergence to optimality following general policy iteration diagram

[1]By construction greedy policy meets conditions of the theorem.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

where $\xrightarrow{\text{E}}$ denotes a policy evaluation and $\xrightarrow{\text{I}}$ denotes a policy improvement. It turns out that if we act greedily w.r.t. the values of the states we will eventually end up with an optimal policy. In a finite MPD, the convergence is gotten after a finite number of iterations.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   $\textit{policy-stable} \leftarrow \textit{true}$
   For each $s \in \mathcal{S}$:
       $\textit{old-action} \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       If $\textit{old-action} \neq \pi(s)$, then $\textit{policy-stable} \leftarrow \textit{false}$ (!!!)
   If $\textit{policy-stable}$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

(!!!) : This algorithm is written as found in [1], but line marked can present some problems. Imagine if we are in a state s where either actions a_1, a_2 predicted by the policy π lead to the same state s'. In this case the policy will keep on oscillating and might never terminate. In order to fix this, one can simply set *policy_stable* $\leftarrow F$ and change marked line for:

$$\text{If } v_{\pi'}(s) = v_\pi(s) \text{ then } \textit{policy\_stable} \leftarrow T$$

So far we have considered only deterministic policies. In general, we have stochastic policies $\pi(s \mid a)$. In the stochastic case we do not need to select a single action among them, instead each maximizing action can be given a portion of the probability of being selected by the greedy policy (any apportioning scheme is allowed as long as all suboptimal actions are given 0 probability).

■ **Example 4.1 — Gridworld.** Lets consider the grid in 4.2. Here $\mathcal{S} = \{1, 2, \ldots, 14\}$ is the set of non-terminal states and the two shaded squares are the terminal states. In each non terminal state, the set of possible actions is $\mathcal{A} = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. Actions that would take the agent off the grid leave the state unchanged. All transitions have a reward of $R_t = -1$ until the terminal state is reached. This is $r(s, a, s') = -1 \; \forall s, s' \in \mathcal{S}, a \in \mathcal{A}$. Examples of transitions:

$$\begin{aligned} &p(6, -1 \mid 5, \rightarrow) = 1, \\ &p(7, -1 \mid 7, \rightarrow) = 1, \\ &p(10, r \mid 5, \rightarrow) = 0 \quad \forall r \in R. \end{aligned}$$

■

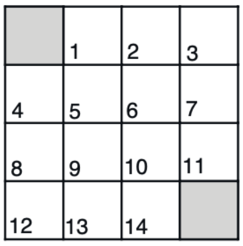


Figure 4.2: Gridworld example

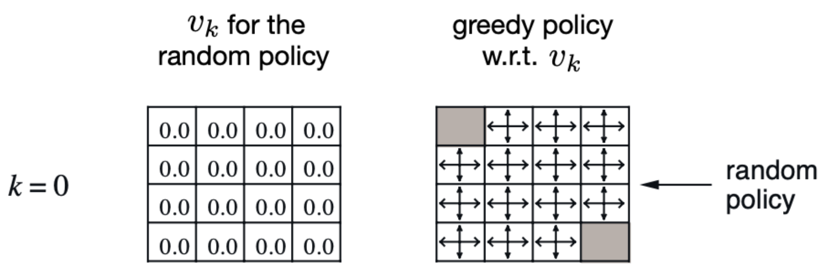
We are going to compute optimal policy for Example 4.1 using the control algorithm built.

Let suppose we want to evaluate the equiprobable random policy:

$$\pi(\uparrow, s) = 1/4,$$
$$\pi(\downarrow, s) = 1/4,$$
$$\pi(\rightarrow, s) = 1/4,$$
$$\pi(\leftarrow, s) = 1/4.$$

We will follow the 4.1 equation to evaluate states and we will take $\gamma = 1$.

$\boxed{v0}$ Initialize all values to zero. Terminal state will always be 0.

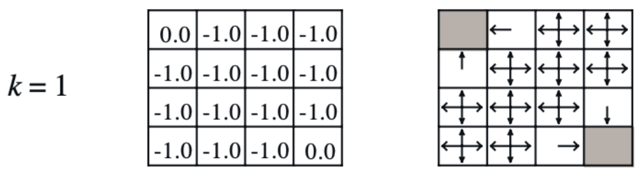


0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$\boxed{v1}$ Let evaluate state 5:

$$\begin{aligned} v_1(5) &= \pi(\uparrow\mid 5)[p(1,-1 \mid 5,\uparrow)(-1 + 1 \cdot v_0(1))]+ \\ &= \pi(\leftarrow\mid 5)[p(4,-1 \mid 5,\leftarrow)(-1 + 1 \cdot v_0(4))]+ \\ &= \pi(\rightarrow\mid 5)[p(6,-1 \mid 5,\rightarrow)(-1 + 1 \cdot v_0(6))]+ \\ &= \pi(\downarrow\mid 5)[p(9,-1 \mid 5,\downarrow)(-1 + 1 \cdot v_0(9))] = -1. \end{aligned}$$

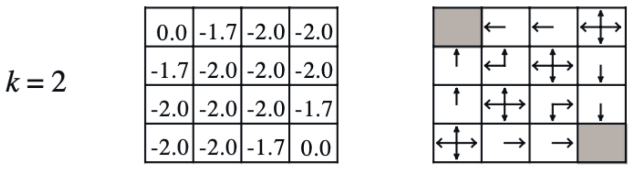
Once all the states are evaluated we will behave greedily following the state values:



0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

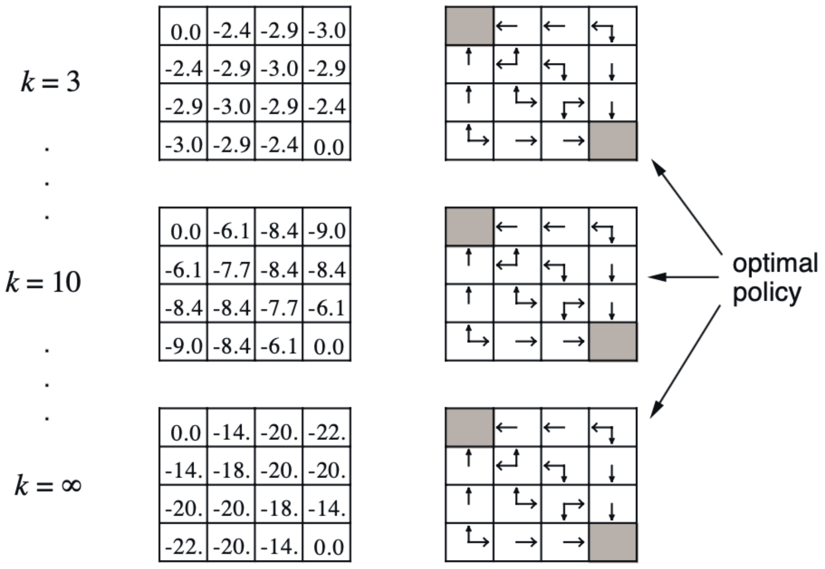
$\boxed{v2}$ We can now evaluate, for example, states 1 and 5:

$$v_2(1) = \tfrac{1}{4} \cdot [1 \cdot (-1 + (-1))] \times 3 + \tfrac{1}{4}[1 \cdot -1] = -1'5 - 0'25 = -1'75.$$
$$v_2(5) = \left(\tfrac{1}{4} \cdot [1 \cdot (-1 + (-1))]\right) \times 4 = -2.$$



0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

And we follow the same pattern until we meet the stoppage condition.



0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Value iteration improvement

One drawback to policy iteration is that each of its iterations involves policy evaluation, which might require multiple *sweeps* through the whole state set. Also if policy evaluation is done iteratively, then convergence to v_π occurs only in the limit. Must we wait for the exact convergence or can we stop short than that? The value iteration improvement combines policy improvement and truncated policy evaluation steps, but policy evaluation is stopped after just one sweep (one update of each step).

$$v_{k+1}(s) = \max_a E\left[R_{t+1} + \gamma v_k\left(S_{t+1}\right) \mid S_t = s, A_t = a\right] = \max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_k\left(s'\right)\right]. \tag{4.2}$$

Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
```

4.1.3 Efficiency of dynamic programming

If the state set is very large, even a single sweep can be prohibitely expensive. *Asynchronous* DP methods are iterative methods that update states in an arbitrary order. For example, one version of asynchronous value iteration updates the value of only one state s_k on each step k using expression 4.2. Asynchronous algorithms also make easier to intermix computation with real-time interaction. To solve a MDP, we can run an iterative DP algorithm at the same time that an agent is actually experiencing the MDP. The experience of the agent can be used to determine the states to which the DP algorithm applies its updates. At the same time, the latest value and policy information from the DP algorithm can guide the decision making by the agent. For example, we can apply updates to states as the agent visits them.

DP might not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient. If we ignore a few technical details, in the worst case, the time DP methods take to find an optimal policy is polynomial in the number of states and actions. A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is k^n, where n and k denote the number of states and actions. Linear programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods. But linear programming methods become impractical at a much smaller number of states than do DP methods, for the largest problems, only DP methods are feasible.

Finally, we note one last special property of DP methods, all of them update estimates of the state values based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea **bootstrapping**. Many RL methods perform bootstrapping, even those that do not require, as MDP requires, a complete and accurate model of the environment. In the next sections we explore RL methods that do not require a model and do not bootstrap (Monte Carlo) and methods that do not require a model but do bootstrap (TD learning).

4.2 Monte Carlo methods

Now we do not assume complete knowledge of the environment, that is full knowledge of the probability distribution of transitions. Monte Carlo methods (MC) learns directly from **experience**, sample sequence of states, actions and rewards:

$$S_0, A_0, R_1, S_1, A_1, \ldots, R_T.$$

We can get two types of experience:

- **Actual experience**; despite not having prior knowledge of the environment dynamics.
- **Simulated experience**; no need for a model.

Samples are **only be defined for episodic tasks** and updated after each episode (not step-by-step). MC methods sample and average returns for each state-action pair.

4.2.1 Prediction

Value estimation (v_π)

The goal is to learn value function for some policy v_π from episodes of experience under that policy π, instead of using an expected return $v_\pi = E[G_t \mid S_t = s]$ we use emperical mean. We will average returns observed after visits to s, there exists two forms of doing this:

- **First visit MC** (FVMC): average returns only for first time s is visited in an episode. Only considers each state once in an episode, we do not consider if in some loop we visit that state again. We get the optimal value function from the law of large numbers and keep on averaging them over multiple episodes.
- **Every visit MC** (EVMC): average returns for every time s is visited in an episode. This is, increment the state counter every time the state is visit and not only once per episode.

■ **Example 4.2 — MC policy value evaluation example.** Consider two episodes of the same MDP where the possible states are $\{A, B, terminate\}$:

$$A \xrightarrow{+3} A \xrightarrow{+2} B \xrightarrow{-4} A \xrightarrow{+4} B \xrightarrow{-3} terminate$$

$$B \xrightarrow{-2} A \xrightarrow{+3} B \xrightarrow{-3} terminate$$

$A \xrightarrow{+3} A \equiv$ transition from state A to A with a reward of $+3$ for this transition.

- **FVMC**
  - $V(A)$
    - 1^{st} episode: $+3 + 2 - 4 + 4 - 3 = +2$.
    - 2^{nd} episode: $+3 - 3 = 0$.
    - $V(A) = \frac{2+0}{2} = 1$.
  - $V(B)$
    - 1^{st} episode: $-4 + 4 - 3 = -3$.
    - 2^{nd} episode: $-2 + 3 - 3 = 2$.
    - $V(A) = \frac{(-3)+(-2)}{2} = -2.5$.

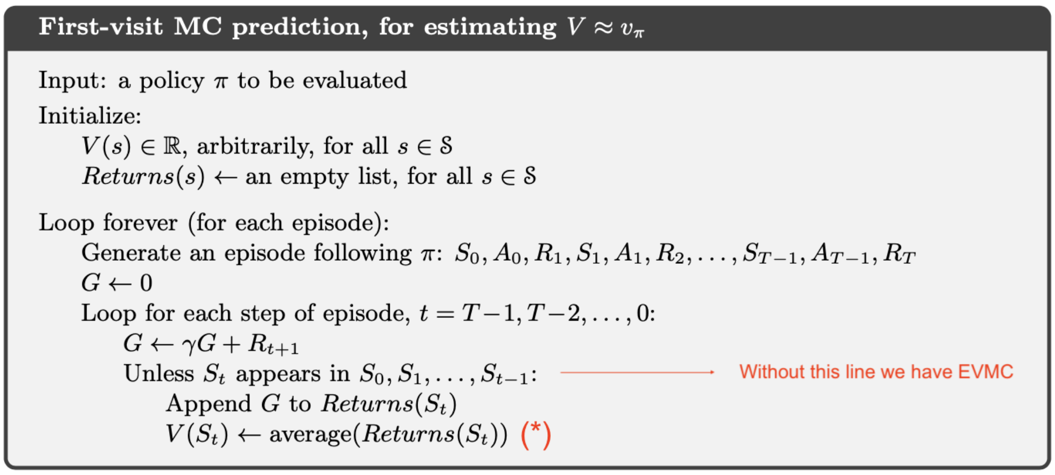
- **EVMC**
  - $V(A)$
    - 1st episode: $(3+2-4+4-3)+(2-4+4-3)+(4-3) = 2-1+1 = 2$.
    - 2nd episode: $3+(-3) = 0$.
    - $V(B) = \frac{2-1+1+0}{4} = 0.5$.
  - $V(B)$
    - 1st episode: $(-4+4-3)+(-3) = (-3)+(-3) = -6$.
    - 2nd episode: $(-2+3-3)+(-3) = (-2)+(-3) = -5$.
    - $V(B) = \frac{-3-3-2-3}{2} = -2.75$.

We can advert that the results both methods arises are different.

Therefore we can build an algorithm for value estimation:

0. Choose FVMC or EVMC.
1. Generate an episode following π.
2. Increment counter $N(S) \leftarrow N(S)+1$.
3. Increment total return $S(s) \leftarrow S(s)+G_t$.
4. Value is estimated by mean return $V(s) \leftarrow S(s)/N(s)$.
5. Repeat from 1 until convergence.

Here we show FVMC algorithm for prediction more detailed.



First-visit MC prediction, for estimating $V \approx v_\pi$

```
Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:  ⟶  Without this line we have EVMC
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$) (*)
```

Incremental average implementation

We can reduce the computational running time of the algorithm by using the incremental implementation of the mean. The mean $\mu_1, \mu_2, \ldots$ of a sequence $x_1, x_2, \ldots$ can be computed incrementally,

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\
&= \frac{1}{k}\left(x_k + (k-1)\mu_{k-1}\right) \\
&= \mu_{k-1} + \frac{1}{k}\left(x_k - \mu_{k-1}\right)
\end{aligned}$$

In our algorithm we will only need to change line $(*)$ for:

$$\begin{aligned} N(S_t) &\leftarrow N(S_t) + 1 \\ V(S_t) &\leftarrow V(S_t) + \alpha(G_t - V(S_t)) \end{aligned}$$

where $\alpha = 1/N(S_t)$ (step-size) for stationary distribution and is some value for non-stationary distribution for running mean, i.e. forget old episodes so we do not have the baggage of past.

Action values estimation (q_π)

With a model, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state. Without a model state values are not sufficient. One must explicitly estimate the value of each action in order to the values to to be useful in suggesting a policy. We use same ideas of estimating value of states: every visit and first visit.

Problem: is π is deterministic, some (many) (s, a) pairs will never be visited.

Solution: **exploration**. Both methods converges asymptotically if every state action pair is visited. We can use *exploring starts* assumption which states that every state-action pair has non-zero probability of being the starting pair.

4.2.2 Control

The overall idea for MC control is to proceed as DP:

- **MC policy iteration:** using previous methods.
- **MC policy improvement:** greedyfing respect to action-state value.

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*$$

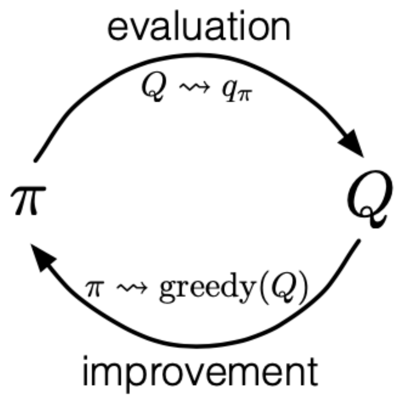

evaluation
$Q \rightsquigarrow q_\pi$
π
Q
$\pi \rightsquigarrow \text{greedy}(Q)$
improvement

Figure 4.3: Control diagram for Monte Carlo methods

Again, policy improvement is done greedyfing. In this case we will be making the policy greedy with respect to to current action-state value, and therefore no model is needed to construct the greedy policy:

$$greedy(q) \sim \pi(s) = \arg\max_a q(s, a)$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}.

N Does the greedified policy meet the conditions for policy?

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max q_{\pi_k}(s,a)) \\ &= \max_a q_{\pi_k}(, a) \\ &\geqslant q_{\pi_k}(s, \pi_k(s)) \\ &\geqslant v_{\pi_k}(s) \end{aligned}$$

Thus $\pi_{k+1} \geqslant \pi_k$ by the policy improvement theorem.

We can then give a control algorithm holding exploring starts assumption:

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

```
Initialize:
    $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
    Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability $> 0$
    Generate an episode from $S_0, A_0$, following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$ (*)
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
```

We can again reforce our algorithm using incremental averaging implementation changing line $(*)$ for:

$$\begin{aligned} N(S_t, A_t) &\leftarrow N(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G - Q(S_t, A_t)) \end{aligned}$$

It is easy to see that MCES cannot converge to a suboptimal policy. If it did, then the value function of that policy would cause the policy to change. Though practically demonstrated, it has not been theoretically proven and is one of the most fundamentals open theoretical questions in RL [21].

4.2.3 Improving Monte Carlo methods

We made two unlikely assumptions above in order fo easily obtain convergence for MC:

Infinite number of episodes

Two approaches for getting this working. One is to hold firm to the idea of approximating q_{π_k} in each policy evaluation, measurements and assumptions are made to obtain bounds on the magnitude and probability of errors in estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficient small. This approach might require far too many episodes to be useful in practice. Second approach is we give up trying to complete policy evaluation before returning to policy evaluation before returning to policy improvement. On each evaluation step we move the value function toward q_{π_k}, but we do not expect to actually get close except over many steps.

Episode have exploring starts

To get rid of the unrealistic hypothesis of exploring starts we have two approaches:

- **On-policy methods:** evaluate or improve the policy that is used to make decision repeatedly.
- **Off-policy methods:** evaluate or improve a policy different from that used to generate the data. Indeed, we can learn directly form taking different actions, so a policy is not needed at all.

In advance, we will be differentiating between on-policy and off-policy methods.

4.2.3.1 On-policy methods

The prediction algorithm reviewed before is a on-policy method. In on-policy control methods the policy is generally *soft*, meaning that $\pi(a \mid s) > 0 \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$, but gradually shifted closer to a deterministic policy.

We use ε-greedy policies, which are examples of ε-soft policies, defined as policies for which $\pi(a \mid s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$ $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

```
Algorithm parameter: small $\varepsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \dots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \dots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
            $A^* \leftarrow \operatorname{argmax}_a Q(S_t, a)$                (with ties broken arbitrarily)
            For all $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

We can show that ε-greedy w.r.t. q_π is an improvement over any ε-soft policy.

Proof. Using the policy improvement theorem, let π' be the ε-greedy policy, then

$$\begin{aligned} q_\pi\left(s, \pi'(s)\right) &= \sum_a \pi'(a \mid s) q_\pi(s,a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \underbrace{\frac{\pi(a \mid s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon}}_{(*)} q_\pi(s,a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a \mid s) q_\pi(s,a) \\ &= v_\pi(s). \end{aligned}$$

Thus by the policy improvement theorem $\pi' \geq \pi$, $(v_{\pi'}(s) \geq v_\pi(s) \quad \forall s \in \mathcal{S})$.

(*): $\pi(a \mid s) = 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|} \implies 1 = \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon}$.

■

We can also prove that the equality can hold only when both π' and π are optimal among the set of ε-soft policies, that is, when they are better than or equal to all other ε-soft policies [1]. Thus we have showed that there will be improvements in every step if we already are not at the optimal value function, but as the previous section, this only happens among the set of ε-soft policies although we have eliminated the assumption of exploring starts.

4.2.3.2 Off-policy methods

All learning control methods face a dilemma: they seek to learn action values conditional on subsequent optimal behavior, but they need to behave non-optimally in order to explore all actions. To deal with this, an approach is to use two separate policies: the policy learned about (*target policy*) and the policy used to generate behaviour (*behaviour policy*). In off-policy methods improvement and evaluation are done on a different policy:

- **Target policy** $\pi(a \mid s)$: the value function of learning is based on π. We want the target policy to be optimal. The target policy will be used for action selection after the learning process is complete (*deployment*).
- **Behaviour policy** $b(a \mid s)$: the behaviour policy is used for action selection while gathering episodes to train the agent.

Some concepts to review before building our off-policy method are the following:

Coverage assumption

In order to use episodes from b to estimate values for π we require that every action taken under π is also taken, at least occasionally, under b. That is, we require that $\pi(a \mid s) > 0 \implies b(a \mid s) > 0$.

Importance sampling

Suppose we have a random variable $X \sim b$ sampled from behaviour policy distribution b. We want to estimate the expected value of X w.r.t. the target distribution π, $E_\pi[X]$.

$$E_\pi[x] = \sum_x x\pi(x) = \sum_x x\pi(x)\frac{b(x)}{b(x)} = \sum_x x\frac{\pi(x)}{b(x)}b(x) = \sum x\rho(x)b(x).$$

where $\rho(x) = \frac{\pi(x)}{b(x)}$ is called the *importance sampling ratio.*

Let $x\rho(x)$ be a new random variable $X\rho(X)$, then

$$E_\pi[X] = \sum_x x\rho(x)b(x) = E_b\left[X\rho(X)\right],$$

and we will be able to estimate expectation as follows

$$E_\pi[X] = E_b\left[X\rho(X)\right] \approx \frac{1}{n}\sum_{i=1}^{n} x_i\rho\left(x_i\right) \quad (\text{with}\, x_i \sim b).$$

In off-policy methods we have $v_b(s) = E_b[G_t \mid S_t = s]$ and we want to compute $v_\pi(s)$. Applying *importance sampling*:

$$v_\pi(s) = E_\pi\left[G_t \mid s_t = s\right] = E_b\left[\rho G_t \mid S_t = s\right],$$

where $\rho = \frac{P[\text{ trajectory under } \pi]}{P[\text{ trajectory under } b]}$.

The probability of a trajectory, under a policy π, can be given by

$$\begin{aligned} P[\text{trajectory under policy } \pi] &= P\left[A_t, S_{t+1}, A_{t+1}, \ldots, S_T \mid S_t, A_{t:J} \sim \pi\right] \\ &= \pi\left(A_t \mid S_t\right) p\left(S_{t+1} \middle| S_t, A_t\right) \pi\left(A_{t+1} \middle| S_{t+1}\right) \ldots P\left(S_T \mid S_{T-1}, A_{T-1}\right) \\ &= \prod_{k=1}^{T-1} \pi\left(A_k \mid S_k\right) p\left(S_{k+1} \mid S_k, A_k\right). \end{aligned}$$

where p is the state-transition probability function.

$$\implies p_{t:T-1} = \frac{\prod_{k=1}^{T-1} \pi(A_k \mid S_k)p(S_{k+1} \mid S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k \mid S_k)p(S_{k+1} \mid S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.$$

Therefore, using this expression we can estimate $v_\pi(s)$. Let,

- $J(s) \equiv$ set of all time steps in which state s is visited.
- $T(s) \equiv$ first time of termination following t.
- $G_t \equiv$ return after t up to $T(t)$.

We have two strategies of importance sampling:

- **Ordinary importance sampling**

$$V(s) \doteq \frac{\sum_{t\in J(s)} \rho_{t:T(t)-1} G_t}{|J(s)|}.$$

- **Weighted importance sampling**

$$V(s) \doteq \frac{\sum_{t\in J(s)} \rho_{t:T(t)-1} G_t}{\sum_{t\in J(s)} \rho_{t:T(t)-1}},$$

or zero if the denominator is zero. For action value we got similar expression, the only change is that instead of counting the state visit we count the state-action visit i.e. $J(s) \to J(s,a)$

$$Q(s,a) \doteq \frac{\sum_{t\in J(s,a)} \rho_{t:T(t)-1} G_t}{\sum_{t\in J(s,a)} \rho_{t:T(t)-1}}.$$

In practice, the weighted estimator has dramatically lower variance and is strongly preferred. Also every-visit algorithms are used because it removes the need to maintain which states have been visited.

Incremental implementation[2]

Suppose we have a sequence of returns $G_1, G_2, \ldots, G_{n-1}$ all starting in the same state and each with a corresponding random weight $W_i \equiv \rho_{t_i:T(t_i)-1}$.

We derive the the estimate of

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}.$$

Let $C_{n+1} \doteq C_n + W_{n+1}$, $C_0 \doteq 0$ and V_1 arbitrarily, then we have

$$\begin{aligned} V_{n+1} &= \frac{\sum_{k=1}^{n} W_k G_k}{\sum_{k=1}^{n} W_k} = \frac{1}{W_n + \sum_{k=1}^{n-1} W_k}\left(W_n G_n + \sum_{k=1}^{n-1} W_k G_k\right) = \\ &= \frac{W_n G_n}{C_n} + \frac{\sum_{k=1}^{n-1} W_k G_k}{C_n} = \frac{W_n G_n}{C_n} + \frac{\sum_{k=1}^{n} W_k}{C_n} V_n = \\ &= \frac{W_n G_n}{C_n} + \frac{C_n - W_n}{C_n} V_n. \end{aligned}$$

$$\implies V_{n+1} = V_n + \frac{W_n}{C_n}\left(G_n - V_n\right), \quad n \geq 1.$$

[2]For the weighted importance sampling case

Now we have settled the conditions to build a prediction off-policy method using incremental implementation.

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

```
Input: an arbitrary target policy $\pi$
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of $\pi$
    Generate an episode following $b$: $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \dots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
```

For control the behaviour policy can be anything, but in order assure convergence of π to the optimal policy an infinite number of returns must be obtained for all possible state-action pairs, which is achieved using ε-soft policies. The target policy $\pi \approx \pi_*$ is the greedy policy with respect to Q, which is an estimate of q_π. When the target policy π is ε-greedy, it is deterministic and $\pi(A_t \mid S_t = s) = 1$.

Thus, the off-policy MC control algorithm can be written as follows.

Off-policy MC control, for estimating $\pi \approx \pi_*$

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$    (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using $b$: $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \dots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)} [G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$    (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

4.3 Temporal Difference Learning

Temporal difference (TD) is a combination of MC and DP ideas. MC in the sense of learning from raw experience and without a model of the environment and DP because of bootstrapping.

4.3.1 Prediction

Monte Carlo methods wait until the return following the visit is known and the use that as a target for $V(S_t)$, then it depends upon the complete roll out of any episode:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t + V(S_t)]$$

TD methods need to wait only until the next step:

$$V(S_t) \leftarrow V(S_t) + \alpha(\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{estimated\ G_t} - V(S_t))$$

where α is a *step size/learning rate* and γ is the *discount factor*. We call $R_{t+1} + \gamma V(S_{t+1})$ the **TD target** and $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ the **TD error**.

The $TD(0)$ is the name given to TD prediction algorithm and is the base of the most important algorithms in TD and RL so far. We use the idea of bootstrapping, we start off with a guess and make a move based on the returns updating our original guess.

Tabular TD(0) for estimating v_π

```
Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Before reviewing TD control methods, some important concerns:

Step size and discount factor. . .

- *Learning rate/step size α*. Determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent explore possibilities. In fully deterministic environments, a learning rate of $\alpha = 1$ is optimal. When the problem is stochastic, the algorithm converges under some considerations on the learning rate that require it to decrease to zero. In practice, often choose a constant rate such $\alpha = 0.1$.
- *Discount factor γ*. Determines the importance of future rewards. A factor of 0 will make the agent consider the most current rewards (*myopic*), while a factor of 1 will make it strive for a long-term high reward. If the discount factor exceeds 1, the action values may diverge. For $\gamma = 1$, without a terminal state, or if the agent never reaches one, rewards generally

become infinite. Even for γ only slightly lower than 1, Q-functions leads to propagation error and instabilities when using approximation value function with a neural network.

About the TD error...

Notice that the TD error δ_t is the error made in estimate at that time. Because TD error depends on the next state and next reward, it is not available until one step later. We can calculate the Monte Carlo error as sum of TD errors from that point:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\ &= \delta_t + \gamma (G_{t+1} - V(S_{t+1})) \\ &= \delta_t + \gamma \delta_{t+1} + \gamma^2 (G_{t+2} - V(S_{t+2})) \\ &= \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} + \cdots + \gamma^{T-t-1} \delta_{T-1} + \gamma^{T-t} (G_T - V(S_T)) \\ &= \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} + \cdots + \gamma^{T-t-1} \delta_{T-1} + \gamma^{T-t} (0 - 0) \\ &= \sum_{k=1}^{T-1} \gamma^{k-t} \delta_k \end{aligned}$$

This identity is not exact if the V changes during the episode (as in TD(0)), but if the step size is small holds approximate.

MC vs TD...

- TD can *learn before knowing the final outcome.*
  - TD can learn online after each step.
  - MC must wait until end of episode before return is known.
- TD can *learn without the final outcome.*
  - TD can learn from incomplete sequences. Also works in continuing tasks.
  - MC can only, learn from complete sequences. Only works for episodic tasks.
- **Bias/variance trade-off**. Return $G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{\top-1} R_T$ is an unbiased estimate of $v_\pi(S_t)$. True TD target $R_{t+1} + \gamma_{\pi_\pi}(S_{t+1})$ is also an unbiased estimate of $v_\pi(S_t)$. But TD target $R_{t+1} + \gamma V(s_{t+1})$ is biased estimate of $v_\pi(s_t)$. TD target is much lower variance that the return, return depends on many random actions, transitions, rewards meanwhile TD depend on one random action, transition and reward.
  - MC has high variance, zero bias. It has good convergence properties (even with function approximation). It is not very sensitive to initial value and is very simple to understand and use.
  - TD has low variance, some bias. It is usually more sensitive to initial value. TD(0) converges to $v_\pi(s)$ but not always with function approximation.
- **Markov property**.
  - MC does not exploits Markov Property.
  - TD exploits Markov property.

4.3.2 SARSA (on-policy control)

The proceeding of SARSA is simple: you start with a state-action pair (S, A), you sample the environment, get reward R and end up in state S', where sampling your policy take you to action A' (i.e. $SARS'A'$).

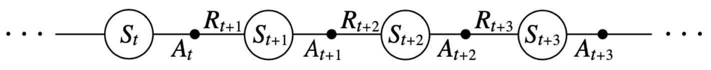


Figure 4.4: Transition diagram for SARSA

We consider these transitions and get the following equation for the value function:

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

This update is done for each non-terminal state, for a terminal state we define $Q(S, A) = 0$.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

The convergence properties of SARSA depend on the nature of the policy's dependence on Q, e.g. one could use ε-greedy or ε-soft policies. SARSA converges with probability 1 to an optimal policy and optimal action-value function as long as all the states-actions pairs are visited infinite times.

4.3.3 Q-learning (off-policy control)

Q-learning has gained popularity recently due to the implementation as a Deep RL algorithm (which we will review in chapter 5) thanks to DeepMind Google's company. The idea was first defined by Watkings in 1969 in his PhD thesis *"Learning from Delayed Rewards"* [22].

The Q-function is defined as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4.3}$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

In this case, the learned action-value function Q directly approximates q_*, the optimal action-value function, independently on the policy being followed (also off-policy). The policy still has an effect, it determines which state-action pairs are visited and updated. All that is required for correct convergence is that all pairs continue to be updated.

Initial conditions (Q_0)

Since Q-learning is an iterative algorithm, it implicitly assumes an initial condition before the first update occurs. High initial values, also known as *optimistic initial conditions*, can encourage exploration: no matter what action is selected, the update rule will cause it to have lower values than the other alternatives, thus increasing their choice probability. The first reward can be used to reset the initial condition. According to this idea, the first time an action is taken the reward is used to set the value of Q. This allows immediate learning in case of fixed deterministic rewards. A model that incorporates reset of initial conditions (RIC) is expected to predict participants behaviour better than a model that assumes any arbitrary initial condition (AIC).

Implementation of Q-learning: Q-tables

The easiest implementation of Q-learning is through the use of Q-tables, which are tables whose cells are the combination of state-action pairs first initialized to zero and then each cell is updated through training. Each cell represents the maximum expected reward. The update of each state-action is carried using Q-learning Q function 4.3.

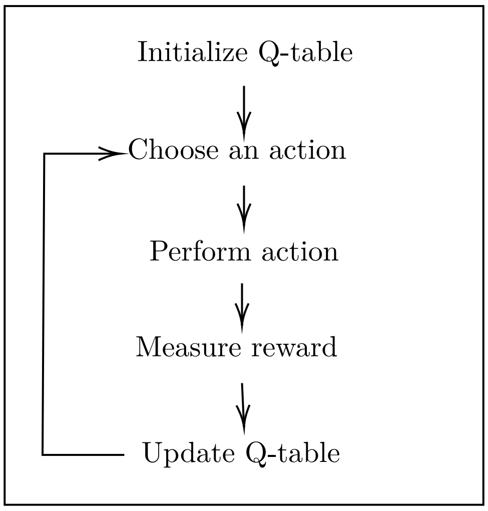


Figure 4.5: Q-learning scheme using Q-tables

■ **Example 4.3 — Q-table Gridworld.**

Consider the Gridworld of states in Figure 4.6. Actions are $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. Rewards at each step are -1, unless you jump into a bomb, which is rewarded with -100 and you will have to start again, or if you get power (ray), in which case you get +1. The negative -1 is given so that the robot takes shortest path and reaches the goal as fast as possible. If the robot reach the End, he gets 100 points. We choose $\alpha = 0.1$ and $\gamma = 0.9$. ■

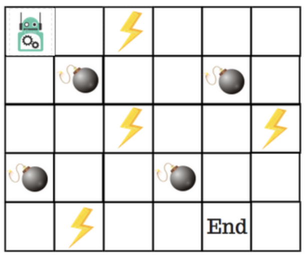


Figure 4.6: Q-table Gridworld

First, we initialize the Q-table:

	↑	→	↓	←
Start	0	0	0	0
Blank	0	0	0	0
Power	0	0	0	0
Bomb	0	0	0	0
End	0	0	0	0

Table 4.1: Gridworld Q-table

On first iteration, we choose action →, evaluate it following 4.3 and we actualize the Q-table:

$$\begin{aligned}&Q^{new}(start, \rightarrow) = \\ &= Q(start, \rightarrow) + 0'1.[-1 + 0'9 \max\{Q(start, \downarrow), Q(start, \uparrow), Q(start, \leftarrow), Q(start, \uparrow)\} - Q(start, \rightarrow)] = \\ &= 0 + 0'1.[-1 + 0'9.0 - 0] = -0'1\end{aligned}$$

For a second iteration, we might behave greedily, but if so we will choose another random action because all are equally valued.

$$\begin{aligned}&Q^{new}(blank, \downarrow) = \\ &= 0 + 0'1.[-100 + 0'9. \max\{Q(start, \downarrow), Q(start, \uparrow), Q(start, \leftarrow), Q(start, \uparrow)\} - 0] = \\ &= 0 + 0'1.[-100 + 0'9.0 - 0] = -10\end{aligned}$$

After two iterations we will have the next Q-table

	↑	→	↓	←
Start	0	-0’1	0	0
Blank	0	0	-10	0
Power	0	0	0	0
Bomb	0	0	0	0
End	0	0	0	0

Table 4.2: Gridworld Q-table after 2 iterations

Now the episode is over, so we start a new one and follow the same idea until convergence (when satisfying stoppage condition for example).

Q-learning at its simplest stores data in tables. This approach fails with increasing numbers of states/actions since the likelihood of the agent visiting a particular state and performing a particular action might be increasingly small. Some solutions for this issue:

- Q-learning can be combined with function approximation. This makes possible to apply the algorithm to larger problems, even when the state space is continuous. One possibility is to use an (adapted) artificial neural network as a function approximator (chapter 5). Function approximation may speed up learning in finite problems, due to the fact that the algorithm can generalize earlier experiences to previously unseen states.

- Another technique is to try to decrease the state action space. For example, in the previous gridworld we could have used positions as states like we did in the DP section, but instead we just use the element the agent will find in a state and this result in less states; another way of planning.

4.3.3.1 Double Q-learning

In these algorithms, a maximum over estimates values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. One way to handle this bias is to introduce two q-values: one q-value Q_1 to determine the maximizing action $A^* = \operatorname{argmax}_a Q_1(a)$ and another Q_2, to provide the estimate of its value $Q_2(A^*) = Q_2(\operatorname{argmax}_a Q_1(a))$. Both $Q_1(a)$ and $Q_2(a)$ are estimates of the true value $q(a)$ for all $a \in A$. The estimate is unbiased in the sense that $E[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(\operatorname{argmax}_a Q_2(a))$. Then two separate value functions are trained in a mutual environment using separate experiences. The double Q-learning update step is then as follows:

$$
\begin{aligned}
Q_1(S_t, A_t) &\leftarrow Q_1(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q_2\left(S_{t+1}, \operatorname{argmax}_a Q_1(S_{t+1}, A_{t+1})\right) - Q_1(S_t, A_t)\right] \\
Q_2(S_t, A_t) &\leftarrow Q_2(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q_1\left(S_{t+1}, \operatorname{argmax}_a Q_2(S_{t+1}, A_{t+1})\right) - Q_2(S_t, A_t)\right]
\end{aligned}
$$

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A$ from $S$ using the policy $\varepsilon$-greedy in $Q_1 + Q_2$
      Take action $A$, observe $R$, $S'$
      With 0.5 probabilility:
         $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
      else:
         $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
      $S \leftarrow S'$
   until $S$ is terminal
```

Of course there are double versions of $SARSA$. There are several more algorithms that might perform slightly better depending on the circumstances, for example n-step bootstrapping and $TD(\lambda)$, which also combines the ideas of both MC and TD, but for now we have settled the basis to reach our final goal: **understanding the Deep Q-Networks**.

In games, we know who has won... You get the reinforcement for having played well.

Michael A. Stackpole

5 Example: Blackjack

■ **Example 5.1 — Blackjack.**

Player is given two cards. Dealer is given also two cards, but you can only see one. If a player is given 21, he wins (get a *natural*) unless dealer gets 21, which ends in a draw. In the simplest Blackjack version, if the player does not have a natural, then he can *hit* cards until he either *stick* or goes *bust* (cumulative sum above 21). The set of actions is therefore $\mathcal{A} = \{HIT, STICK\}$. If he goes bust, he loses, and if he sticks, is the dealer's turn, which can hit or stick following a fixed strategy:

$$\text{Dealer's fixed startegy} = \begin{cases} STICK & \text{if} \quad \text{dealer cumsum} \geq 17 \\ HIT & \text{if} \quad \text{dealer cumsum} < 17 \end{cases}$$

Card values are:

- A (*ace*): 1 or 11. If the player holds an *ace* he can use as 11 without going bust, then that *ace* is said to be *usable*.
- $\#i$: i $\forall i = \{2, \ldots, 10\}$.
- J, Q, K: 10.

The objective of the game is to have your card sum greater than the dealer's without going exceeding 21. The states are based in three facts: - Current sum $[12, 21]$ (for less than 12 agent should always HIT). - Dealer's showing card $\{A, 2, \ldots, 10\}$ - Having a usable card or not.

■

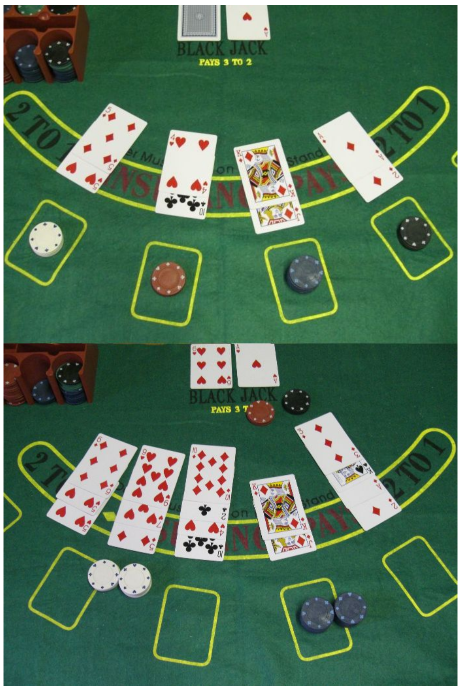


Figure 5.1: Example of a Blackjack hand

We have a total of 200 possible states. DP methods require knowing the action-state transitions probabilities and this can be complex to compute for Blackjack because they might change on each episode. It is straightforward to apply Monte Carlo and TD learning to Blackjack because episodes can be considered the different hands, having a clear beginning and ending.

Also, a well known system to play Blackjack is following *Thorp's basic strategy.* This is simply a table containing each possible combination of states in Blackjack (the sum of your cards and the value of the card being shown by the dealer) along with the best action to take (*HIT* or *STICK*) according to probability computations. This is an example of policy and we will compare our model optimal policies with this policy.

	A	2	3	4	5	6	7	8	9	10
≤ 11	H	H	H	H	H	H	H	H	H	H
12	H	H	H	S	S	S	H	H	H	H
13	S	S	S	S	S	S	H	H	H	H
14	S	S	S	S	S	S	H	H	H	H
15	S	S	S	S	S	S	H	H	H	H
16	S	S	S	S	S	S	H	H	H	H
17	S	S	S	S	S	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S
20	S	S	S	S	S	S	S	S	S	S
21	S	S	S	S	S	S	S	S	S	S

Table 5.1: Thorp's basic strategy for Blackjack

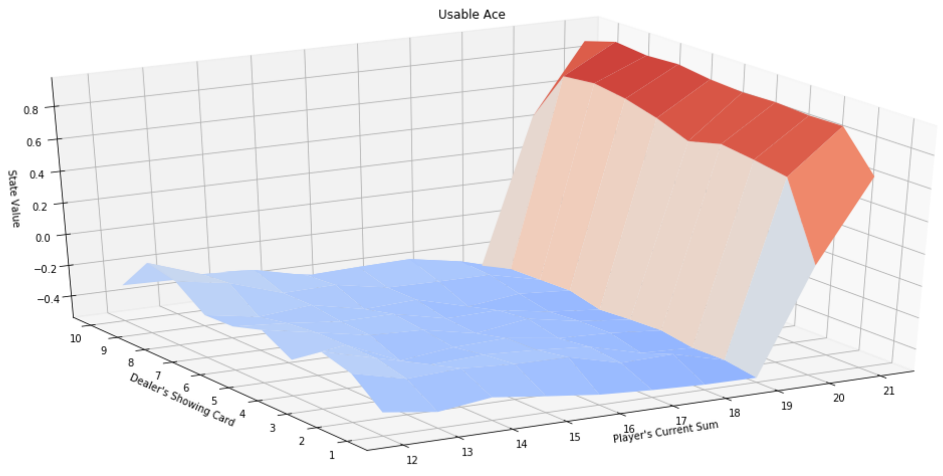
The `gym` library in `Python` have already programmed this `Blackjack` version, so we can use it to simulate different episodes. In this version, the agent only has 2 options: *HIT* (1) or *STICK* (0). The episode will end whenever the player wins or lose (if tied then keep playing). The reward for winning is +1, drawing is 0, and losing is -1.

5.1 Prediction

Just for Monte Carlo methods, we first see how the prediction process works to evaluate a given policy. We consider the following policy:

$$\pi \equiv \begin{cases} STICK & \text{if} \quad \text{current sum } \geq 20 \\ HIT & \text{if} \quad \text{current sum} < 20 \end{cases}$$

We will use the simplest prediction algorithm, **first visit MC prediction algorithm** we get following evaluation of our policy π. After 500,000 episodes we get the following results:



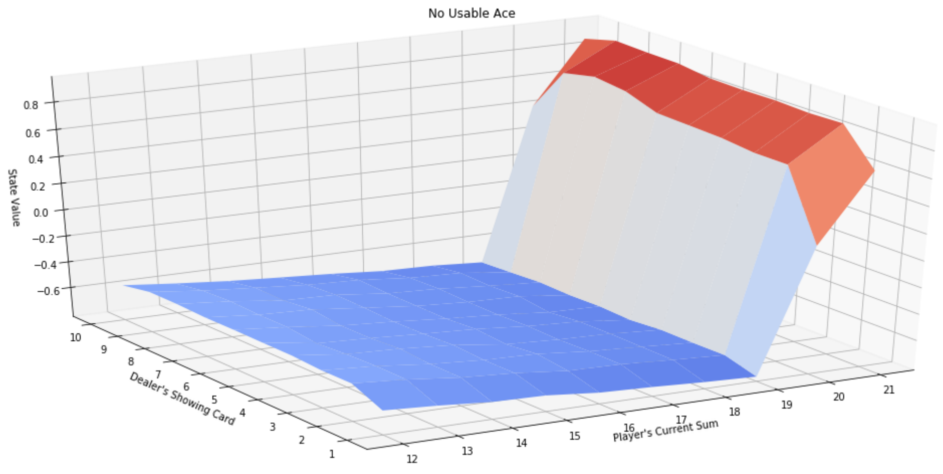


Figure 5.2: Evaluation of the policy using First Visit Monte Carlo algorithm for prediction

We notice that having a sum of 20 or 21 is positive for our behaviour following policy π. Precisely, this is because we *STICK* when we get 20 or 21, situation at which we will be more likely to win in most of the cases. Meanwhile, if we always *HIT* when we have less then 20, we can see a slope in the policy values; hitting with a value of 19 is bad, we will end up losing most of the hands, but hitting with a sum of 11 is better (although not much) because we could receive a sum near 20 with a few cards.

Another aspect to note is that if the dealer shows an 1 (*ace*) or a 10, the value of the state decreases. In these cases, the dealer could easily exceed our sum with another card. In the case that we have a *usable ace*, the value function of the states is more abrupt, although it reflects the same conclusions. This is because it is more unlikely that we have a usable ace and although we have played 500,000 episodes, it is not enough to make it smoother.

5.2 Control

Now we can use control methods to look for an optimal policy (also for optimal value function, although we will not compute it). After running 500,000 episodes these are the optimal policies computed:

- **On-policy first visit MC algorithm**.

Average payout after 1000 rounds is -48.862.

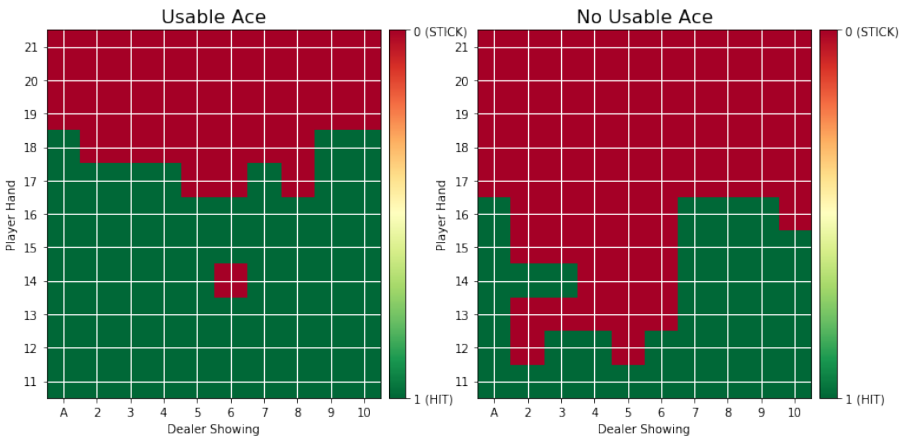


Figure 5.3: Optimal policy for on-policy MC

- **Off-policy first visit MC algorithm** using *weighted importance sampling*.

Average payout after 1000 rounds is -114.669.

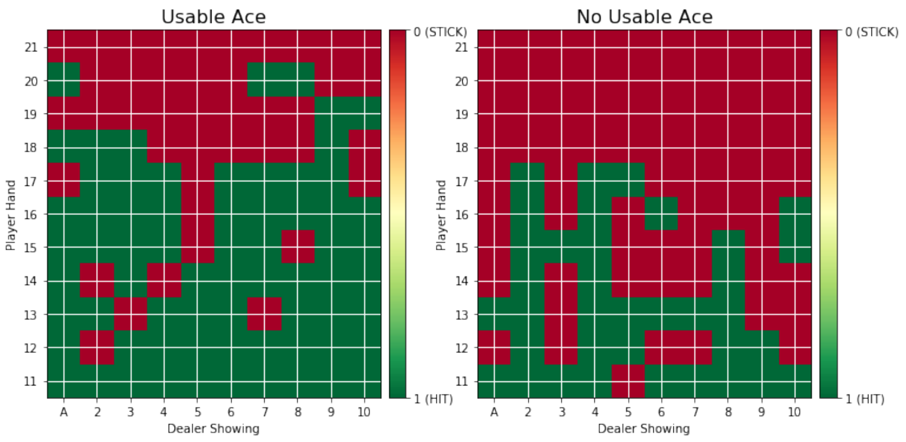


Figure 5.4: Optimal policy for off-policy MC

- **SARSA** (On-policy TD algorithm)

Average payout after 1000 rounds is -174.733.

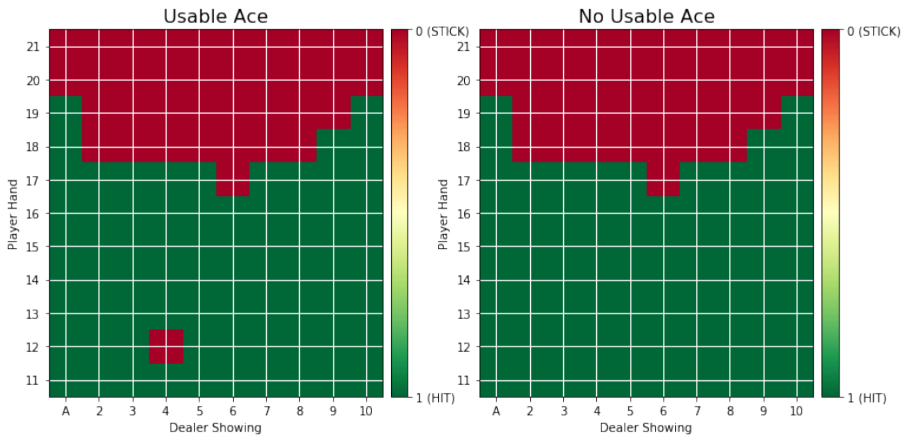


Figure 5.5: Optimal policy for SARSA

- **Q-Learning** (Off-policy TD algorithm)

Average payout after 1000 rounds is -134.343.

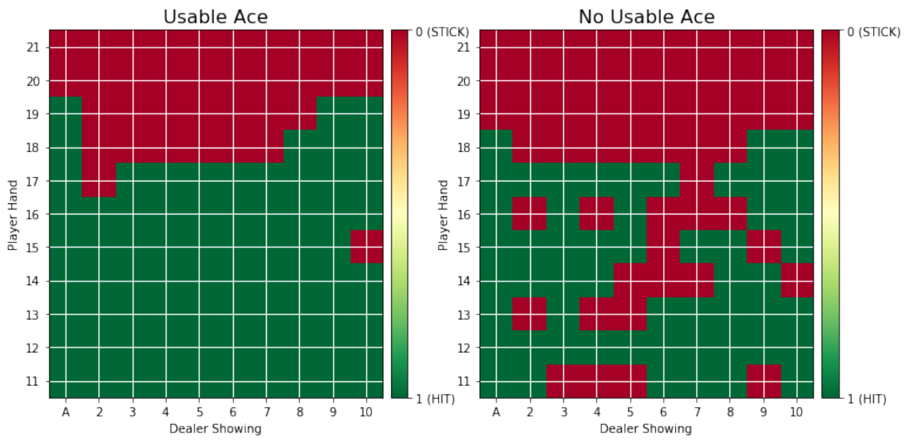


Figure 5.6: Optimal policy for Q-learning

We can see the different policies. The best policy we trained is using the Monte Carlo on-policy algorithm, with an highest average payout, although it is still negative. The fact that we get negative payouts means that, even with an optimal policy to follow, our payout for playing Blackjack will be negative, suggesting that this game should not be played.

Also, none of the policies matches the policy of the *basic strategy* nor the policy in Figure 5.7, showed in [1] using just Monte Carlo exploring starts. Only on-policy first visit MC shows some similarities[1]. This means, although mostly similar, our models suggest different ways to behave in certain cases.

Full code is available in *Appendix A*.

[1]To compare the *basic staregy* with our policies we must just focus on the *not usable ace* case.

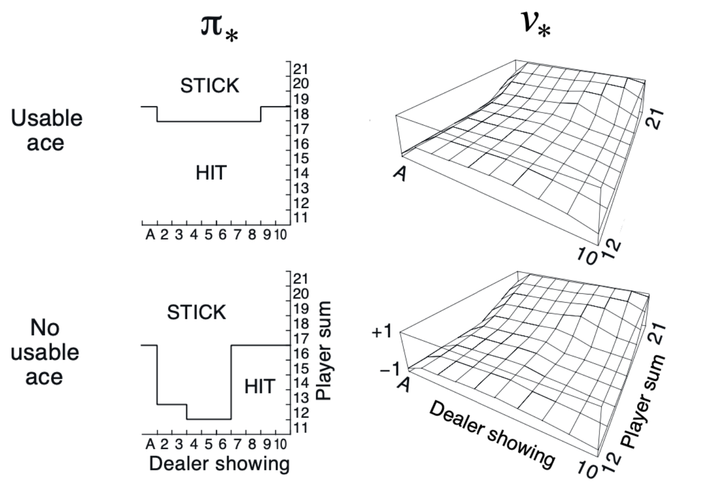


Figure 5.7: Optimal policy and state-value function found by Monte Carlo ES

What would life be if we had no courage to attempt anything new?

Vincent Van Gogh

6 Deep Q-Networks

For this chapter, the basic concepts have been reviewed using the book [23], adjusting the notation to our previous work. As we mentioned in the previous chapter, *vanilla* Q-learning (Tabular Q-learning) is not practical when we have a high amount of states and actions, but RL still can be used to solve large problems, e.g. Backgammon (10^{20} states), Go (10^{170} states) or helicopter driving (continuous state space).

Recapping, $q_*(s,a)$ is the expected utility starting in s, taking action a and thereafter acting optimally. We could understand q_* using the Bellman optimality equation[1]:

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

Using Q-Value iteration expressed as[2]:

$$Q_{k+1}(s,a) = \sum_{s'} p\left(s' \mid s,a\right)\left[r\left(s,a,s'\right) + \gamma \max_{a'} Q_k(s',a')\right]$$

The scheme of *vanilla* Q-learning is the following:

- For an state-action pair (s,a) receive $s' \sim p(s' \mid s,a)$.
- Consider your old estimate $Q_k(s,a)$.
- Consider your new sample estimate:

$$\textit{target}\left(s'\right) = r\left(s,a,s'\right) + \gamma \max_{a'} Q_k\left(s'a'\right)$$

- Incorporate the new state in to a running average:

$$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha[\textit{target}\left(s'\right)]$$

[1]Note we will be using state-transition probabilities and expected rewards for state action-next-state, which were also introduce in *Chapter 2*, and arises similar formulas.

[2]We can rewrite this as an expectation as follows $Q_{k+1}(s,a) = E_{s' \sim p(s'|s,a)}\left[r(s,a,s') + \gamma \max_{a'} Q_k\left(s',a'\right)\right]$

Algorithm 6.0.1 Vanilla or tabular Q-learning

```
Start with $Q_0(s,a)$ for all $s, a$
Get initial state $s$
for $k = 1, \ldots,$ till convergence do
    sample action $a$, get next state $s'$
    if $s'$ is terminal then
        $target = r(s, a', s')$
        sample new initial state $s'$
    else
        $target = r(s, a', s') + \gamma \max_{a'} Q_k(s', a')$
    end
    $Q_{k+1} \leftarrow (1-\alpha) Q_k(s,a) + \alpha[target]$
    $s \leftarrow s'$
end
```

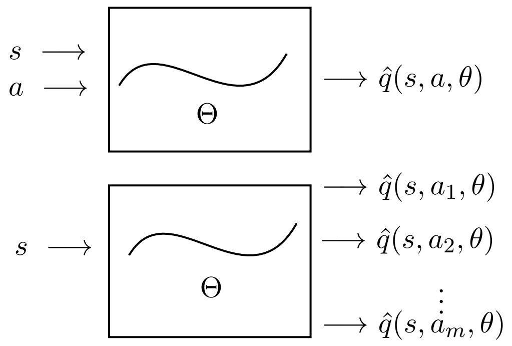
6.1 Approximate Q-learning

Now, instead of a table, we will have a parametrized function:

$$\hat{v}(s,\theta) \text{ or } \hat{v}_\theta(s) \approx v_\pi(s)$$
$$\hat{q}(s,a,\theta) \text{ or } \hat{q}_\theta(s,a) \approx q_\pi(s,a)$$

For $\hat{q}$ we got two possibilities:

- Use state-action pairs as input and returning just a q value.
- Use only states as inputs and returning a q value for each action.



There are a lot of possible function approximators, but we want to consider differentiable approximators. Furthermore, we require a training method that is suitable for non-stationary, non-iid data:

- Linear combination of features ($q_\theta(s,a) = \theta_0 f_0(s,a) + \ldots + \theta_n f_n(s,a)$).
- Artificial Neural Network.

6.2 Stochastic gradient descent

Let review gradient descent methods, already introduced in chapter 2, and explain in detail the Stochastic Gradient Descent. Let $J(\theta)$ he a differentiable function of parameter vector θ. We then define the gradient of $J(\theta)$ to be:

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

This vector indicates the direction of deepest descent, in order to find a local minimum of $J(\theta)$ we follow the next instructions:

$$\Delta\theta = \theta_{k+1} - \theta_k = -\frac{1}{2}\alpha\nabla_\theta J(\theta)$$

where α is some step size parameter.

In our case we have training data stored in batches and our goal is to find parameter vector θ minimizing mean-squared error between approximate value function $\widehat{V}(s,\theta)$ and true value function $v_\pi(s)$:

$$J(\theta) = E_\pi\left[\left(v_\pi(s) - \hat{v}(s,\theta)\right)^2\right]$$

The standard gradient descent algorithm updates the parameters θ of the objective $J(\theta)$ as

$$\theta = \theta - \alpha\nabla_\theta E[J(\theta)]$$

where the expectation in the above equation is approximated by evaluating the cost and gradient over the full training set. **Stochastic gradient descent** (SGD) simply does away with the expectation in the update and computes the gradient of the parameters using only a single or a few training examples. The new update is then given by,

$$\theta = \theta - \alpha\nabla_\theta J\left(\theta; x^{(i)}, y^{(i)}\right)$$

with a pair $\left(x^{(i)}, y^{(i)}\right)$ of samples from the trainig set (*mini-batch*). In our case,

$$\Delta\theta = \alpha\left(v_\pi(s) - \hat{v}(s,\theta)\right)\nabla_\theta\hat{v}(s,\theta)$$

Here we assume the true value function $v_\pi(s)$ is given by the supervisor, but in RL there is no supervisor, only rewards (difference between SL and RL). In practice, we substitute a target for

$\mapsto$ for MC the target is the return G_t:

$$\Delta\theta = \alpha\left(G_t - \hat{v}\left(S_t,\theta\right)\right)\nabla_\theta\hat{v}\left(S_t,\theta\right)$$

$\mapsto$ For TD the target is the TD target $R_{t+1} + \gamma\hat{v}\left(S_{t+1},\theta\right)$:

$$\Delta\theta = \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1},\theta) - \hat{v}(S_t,\theta)\nabla_\theta\hat{v}(S_t,\theta)$$

■ Example 6.1 — Linear value function approximation.

The common way to represent a state is through feature vectors.

$$x(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

These features are explanatory variables relating to the state (or state and action) and need to be sufficient to explain the value of that state. In RL literature, there is often no differencee between "state representation" or "features", although you might need to process the state representation using feature engineering into suitable features to apply SL methods. For example, for the cart-pole example, we can have 4 features:

- Cart position.
- Cart velocity.
- Pole angle.
- Pole velocity at tip.

We can represent then value function by linear combination of features:

$$\hat{v}(s, \theta) = x(s)^t \theta = \sum_{j=1}^{n} x_j(s) \theta_j$$

Our objective function is then quadratic in parameters θ.

$$J(\theta) = E_\pi \left[\left(v_\pi(S) - x(s)^t \theta \right)^2 \right]$$

Stochastic descent converges to a global optimum and the update rule is particularly simple[a]:

$$\begin{aligned} &\nabla_\theta \hat{v}(s, \theta) = x(s) \\ &\Rightarrow \Delta\theta = \alpha \left(v_\pi(s) - \hat{v}(s, \theta) \right) x(s) \end{aligned}$$

■

[a] update = step-site × prediction error × feature value.

The same results can be applied for action-value function $\hat{q}(s, a, \theta)$. Here the update rule can be written as follows:

$$\theta_{k+1} \leftarrow \theta_k - \frac{\alpha}{2} \nabla_\theta \left[\left(\hat{q}(s, a, \theta) - \text{target}\,(s') \right)^2 \right]$$

where $target(s') = r(s, a, s') + \gamma \max_{a'} q(s', a', \theta)$ for Q-learning, and has different expressions for ML, TD or SARSA.

Operating we also get the following simplified expression:

$$\Delta\theta \leftarrow \alpha \left(\text{target}\,(s') - \hat{q}(s, a, \theta) \right) \nabla_\theta \hat{q}(s, a, \theta)$$

We now spot the connection between tabular Q-learning and approximation function. Suppose we have $\theta \in \mathbb{R}^{|S|\times|A|}$ and $q_\theta(s,a) \equiv \theta_{\mathrm{Sa}}$ then

$$\begin{aligned}
&V_{\theta_{sa}}\left[\frac{1}{2}\left(q_\theta(s,a) - \text{target}\left(s'\right)\right)^2\right] = \\
&= \nabla_{\theta_{sa}}\left[\frac{1}{2}\left(\theta_{Sa} - \text{target}\left(s'\right)\right)^2\right] = \\
&= \theta_{sa} - \text{target}\left(s'\right)
\end{aligned}$$

When plugging this expression into the learning rule we get an expression similar to the one we get for tabular Q-learming:

$$\begin{aligned}
&\theta_{s_a} \leftarrow \theta_{sa} - \alpha\left(\theta_{sa} - \text{target}\left(s'\right) = \right. \\
&= (1-\alpha)\theta_{sa} + \alpha\,\text{target}\left(s'\right)
\end{aligned}$$

$$\left(\implies Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left[\text{target}\left(s'\right)\right]\right)$$

Although gradient descent is simple and appealing, it is not *sample efficient*, we do not use all the maximum information on the data, given the agent's experience (*"training data"*).

6.3 Batch learning

Batch methods seek to find the best fitting value function. In this methods we use the previous experience stored in an element D consisting on $\langle state, value\rangle$ pairs:

$$D = \left\{\left\langle s_1, v_1^\pi\right\rangle, \langle s_2, v_2^\pi\rangle, \cdots, \langle S_T, v_T^\pi\rangle\right\}$$

Each batch is then a succession of states, actions and rewards:

$$\begin{gathered}
s_1^1, a_1^1, r_2^1, \ldots, s_{T_1}^1 \\
\vdots \\
s_1^K, a_1^K, r_2^K, \ldots, s_{T_K}^K
\end{gathered}$$

So, given value function approximation $\hat{v}(s,\theta) \approx v_\pi(s)$, which parameters θ gives the best fitting value function $\hat{v}(s,\theta)$? Least squares algorithm find parameter vector θ minimizing sum-squared error between $\hat{v}\left(S_t,\theta\right)$ and target values v_t^π.

$$\mathrm{LS}(\theta) = \sum_{t=1}^{T}\left(v_t^\pi - \hat{v}\left(S_t,\theta\right)\right)^2 = E_D\left[\left(v^\pi - \hat{v}(s,\theta)\right)^2\right]$$

It turns out that, given experience D, if we follow the batch method:

```
Repeat:
  (1) Sample state, value from experience
  (2) Apply stochastic gradient descent update
```

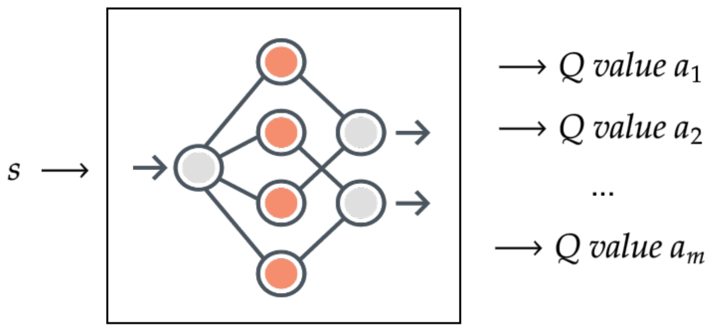
$$\Delta\theta = \alpha\left(v^\pi - \widehat{v}(S,\theta)\right)\nabla_\theta \hat{v}(s,\theta)$$

It converges to the LS solution:

$$\theta^{\pi} = \arg\min_{\theta} LS(\theta)$$

6.4 Deep Q-Learning

Now we are ready to present the Deep Q-learning algorithm (DQL), that is the algorithm behind the Deep Q-Networks (DQN), which have successfully being applied to play Atari games learning of values $Q(s, a)$ from pixels s. DQL is an approximation Q-learning algorithm that uses an Artificial Neural Network as a approximator function.



$s \longrightarrow$
$\longrightarrow Q\ value\ a_1$
$\longrightarrow Q\ value\ a_2$
...
$\longrightarrow Q\ value\ a_m$

Figure 6.1: Deep Q-learning diagram

The idea is to make Q-learning look like supervised learning. DQL also uses **memory replay** and **fixed Q-targets**:

- **Memory replay.** The NN is not updated immediately after every step. Instead, it stores each experience in batches and the updates are then made on a set of batches randomly selected from the replay memory. The batches can be of different time steps.
- **Separated target networks**. Using the same network for both computing the predicted value and estimating the target value used in the loss function can lead to important instabilities. The solution for this problem can be training two separate networks with the same architecture.

DQN algorithm idea

- Take action A_t according to ε-greedy policy.
- Store transition $(S_t, A_t, R_{t+1}, S_{t+1})$ in replay memory D.
- Sample random mini-batch of transitions $\left(s, a, r, s'\right)$ from D.
- Compute Q-learning targets w.r.t. old fixed parameters (θ^{-}).
- Optimize MSE between Q-network and Q-learning targets using stochastic gradient descent.

$$\mathcal{L}_i(\theta_i) = E_{s,a,r,s' \sim D_i}[(r + \gamma \max_{a'} Q\left(s', a'; \theta_i\right) - Q(s, a; \theta_i))^2]$$

- Every C steps, the weights from main network are copied to the target network.

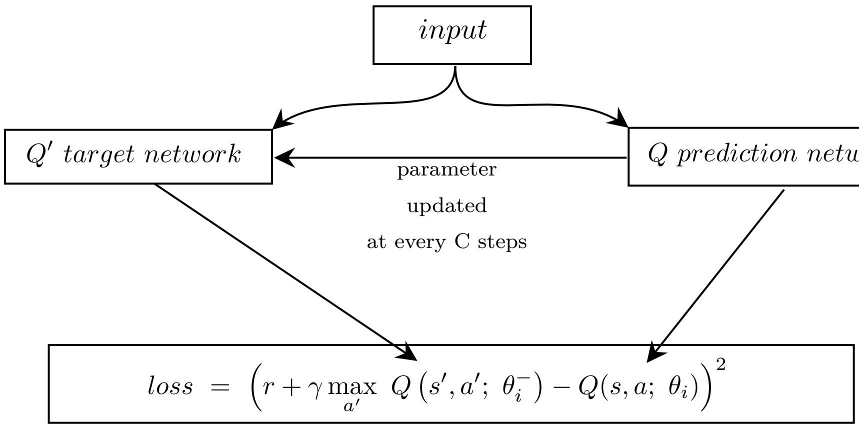

input
Q' target network
Q prediction network
parameter
updated
at every C steps
$loss \;=\; \left(r + \gamma \max_{a'} Q\left(s', a';\ \theta_i^-\right) - Q(s, a;\ \theta_i)\right)^2$

Figure 6.2: Two networks diagram Deep Q-Learning

Algorithm 6.4.1 Deep Q-learning

```
Initialize network $Q$
Initialize target network $\hat{Q}$
Initialize experience replay memory $D$
while not convergence do
    /* Sample phase
    Choose an action $a$ from state $s$ using policy $\epsilon - \text{greedy}(Q)$
    Agent takes action $a$, observe reward $r$, and next state $s'$
    Store transition $(s, a, r, s', terminal)$ in the experience replay memory $D$
    if enough experiences then
        /* Learn phase Sample a random mini-batch of $N$ transitions from $D$
        for every transition $(s_i, a_i, r_i, s_i')$ in mini-batch do
            if $terminal_i$ then
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}\left(s_i', a'\right)$
            end
        end
    end
    Calculate the loss $\mathcal{L} = 1/\text{N} \sum_{i=0}^{N-1} \left(Q\left(s_i, a_i\right) - y_i\right)^2$
    Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
    Every $C$ steps, copy weights from $Q$ to $\hat{Q}$.
end
```

DQN details and possible improvements

1) Having the right model parameter update frequency is important. If you update model weights too often (e.g. after every step) the algorithm will learn very slowly when not much has changed. In practice, every 3 or 4 time steps.

2) Setting the correct frequency to copy weights from the main network to the target network also helpful. We must be careful establishing the value because episodes can have different numbers of steps.

3) Using *Huber loss* function has been proven to boost the learning in certain cases, mainly because *Huber* weights outliers less.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for} |a| < \delta \\ \delta(|a| - \frac{1}{2}\delta\ |) & \text{otherwise} \end{cases}$$

where the variable a often refers to the residuals, the difference between the observed and predicted values.

4) Also important to choose the right initialization of the network: *He*, *Xavier*, ...

DQN in ATARI

The Deep Q-Networks has been successfully applied to play Atari 2600 games, reaching a human level expertise.

As explained in [5], Atari frames are 210 $\times$ 160 pixel images with a 128 color palette. Thus it is necessary to apply a basic preprocessing step aimed at reducing the input dimensionality. The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to a 110$\times$84 image. The final input representation is obtained by cropping an 84 $\times$ 84 region of the image that roughly captures the playing area.

Once the images are preprocessed, the model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards; input state s is stack of raw pixels from last 4 frames and the output is $Q(s, a)$ for 18 joystick/button positions. Rewarding is the change in score for that step. Here we show the network architecture used across all the games.

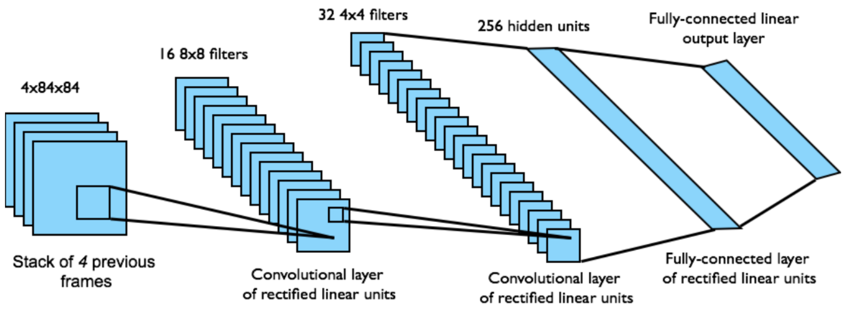

32 4x4 filters
256 hidden units
Fully-connected linear output layer
16 8x8 filters
4x84x84
Stack of 4 previous frames
Convolutional layer of rectified linear units
Convolutional layer of rectified linear units
Fully-connected layer of rectified linear units

Figure 6.3: Network Architecture used to play Atari games

In the article they compare the performance between random selection, Sarsa, DQN and human players for some games computing average total reward by running an ε-greedy policy with $\varepsilon = 0.05$ for a fixed number of steps.

	B. Rider	**Breakout**	**Enduro**	**Pong**	**Q*bert**	**Seaquest**	**Space Invaders**
Random	354	1.2	0	-20.4	157	110	179
SARSA	996	5.2	129	-19	614	665	271
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	-3	18900	28010	3690

Table 6.1: Average total reward for various learning methods when playing Atari games.

The way positive reinforcement is carried out is more important than the amount.

B. F. Skinner

7 Example: Deep Q-Networks and Gridworld

This example is from [24]. The code is programmed in Python using the deep learning framework `PyTorch`. Other resources as the `Gridworld` module to visualize states and actions in a grid were uploaded by the authors to a Github repository.

The Gridworld we are solving here is similar to the one in Example 4.1, but adding a few barriers that complicates the process of learning. Again, the set of possible actions is $\mathcal{A} = \{up, down, left, right\}$. The point of the game is to get the player to the goal, where the player will receive a positive reward. The player must not only reach the goal but to do so following the shortest path. This is a simple Gridworld game setup. The agent (P) must navigate along the shortest path to the goal $(+)$ and avoid falling into the pit $(-)$. There is a wall (W) the agent can not pass through. The starting position of the agent is given.

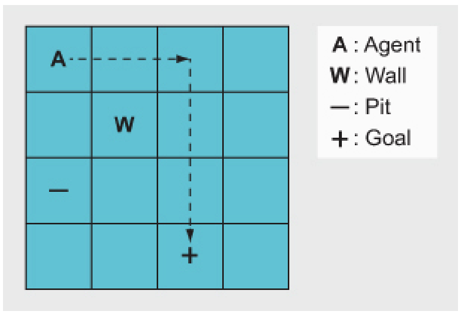


Figure 7.1: Example of Gridworld

Which can be implemented in the code as an array:

```
[['P', ' ', ' ', ' '],
 [' ', 'W', ' ', ' '],
 ['-', ' ', ' ', ' '],
 [' ', ' ', '+', ' ']]
```

The agent is given a reward of -1 in each step while he does not get to the goal. Falling into the pit is rewarded with -10 and reaching the goal with 10. All the agent has access to is the grid and the position of the elements on it.

7.1 Representing the states

The states are represented by a $4 \times 4 \times 4$ tensor. The 4×4 are the grid dimensions (width of 4 cells and height of 4 cells). There are 4 different elements located in the grid the agent, the goal, the wall and the pit. For each of this elements we have 4×4 matrix of zeros and only one 1 in the place where the element is located. At the end, we have 4 matrices of dimensions 4×4 of zeros and ones.

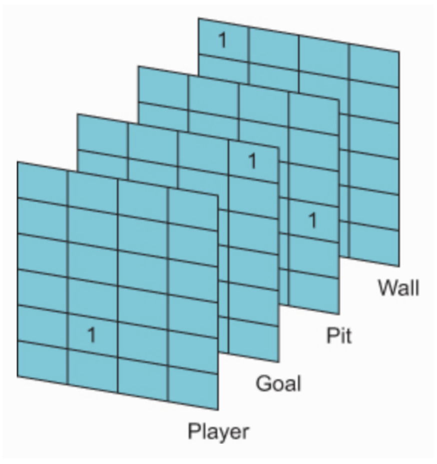


Figure 7.2: Grid elements representation for Gridworld example

7.2 Building the network

Because of the simplicity of the example we do not need to build a complex neural network. For the input layer we need a total of $4^3 = 64$ elements ($\{0, 1\}^{64}$). We add two hidden layers with widths of 100 and 150, where we use the *ReLu* activation function. Finally, given a state we must return a value for each possible action in that state, so the output layer will produce 4 dimensional vectors. The loss function to minimize is the mean square error as we have already explain in the previous chapter.

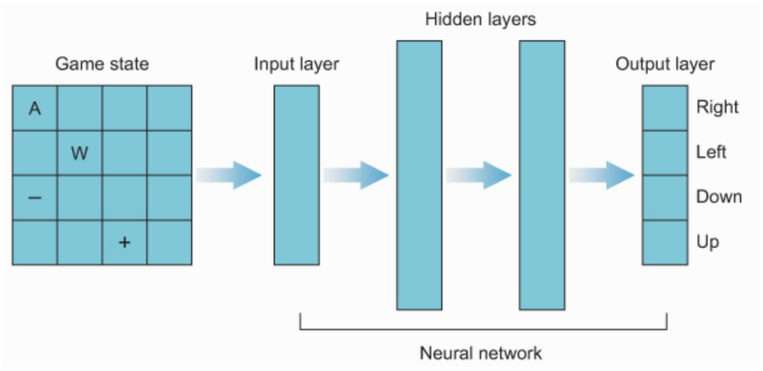


Figure 7.3: Network arquitecture for Gridworld example

As most of the matrix values are zeroes, we might found some problems because the *ReLu* activation function is technically non-differentiable at 0. To deal with this problems we add a bit of noise so that none of the values in the state array are exactly 0. This might also help with overfitting, which is when a model learns by memorizing spurious details in the data without learning the abstract features of the data, ultimately preventing it from generalizing to new data.

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

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

For the ε-greedy policies, we initialize ε to a large value and we decrease it while the training. Other two parameters we need to tune are the step size and the discount factor.

7.3 Results

We build the same Deep Q-Network as in chapter 5, the DQN model with *experience replay* and adding a *target network*, but in order two show the improvements of adding such techniques, we build three different DQN versions:

- **Version 1**: Without experience replay, nor target network.
- **Version 2**: With experience replay but without target network.
- **Version 3**: With both experience replay and target network.

Also, the game can be played in two modes:

- **Static mode**: the items are placed in the same location for each episode.
- **Random mode**: the items are placed randomly for each different episode.

For each epoch, we start a new game. We then compute the loss function over the epochs to compare each version for each mode. For the static mode we get the 100% of victories and the losses decays as the number of epochs increases. Here we show the losses for the 3rd version.

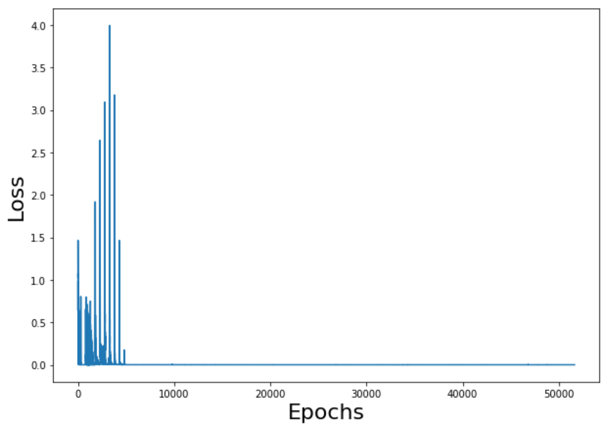


Figure 7.4: Loss plot for static mode and version 3 of the DQN

The loss plot is pretty noisy, but the moving average of the plot is significantly trending toward zero. This gives us some confidence the training worked, let test our model for a static game and see if it converges:

```
Initial State:
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move # 0; Taking action: l
[['+' '-' 'P' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move # 1; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' 'P' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move # 2; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' 'P' ' ']
 [' ' ' ' ' ' ' ']]
Move # 3; Taking action: l
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' 'P' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move # 4; Taking action: l
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 ['P' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move # 5; Taking action: u
[['+' '-' ' ' ' ']
 ['P' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move # 6; Taking action: u
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Game won! Reward: 10
```

The most interesting case is when items are placed randomly after each episode. Here we present the losses over the number of epochs and the percentage of victories for each version of the DQN:

- **Version 1**

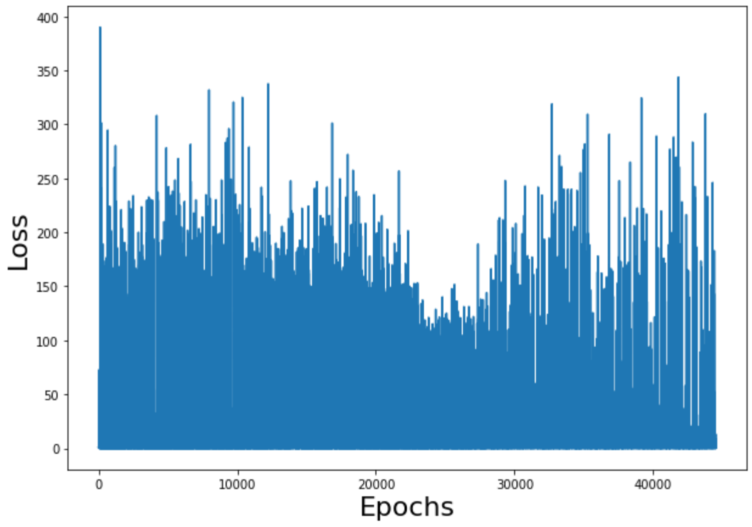


Figure 7.5: Loss plot for random mode and version 1 of the DQN

```
Games played: 1000, # of wins: 843
Win percentage: 84.3%
```

- **Version 2**

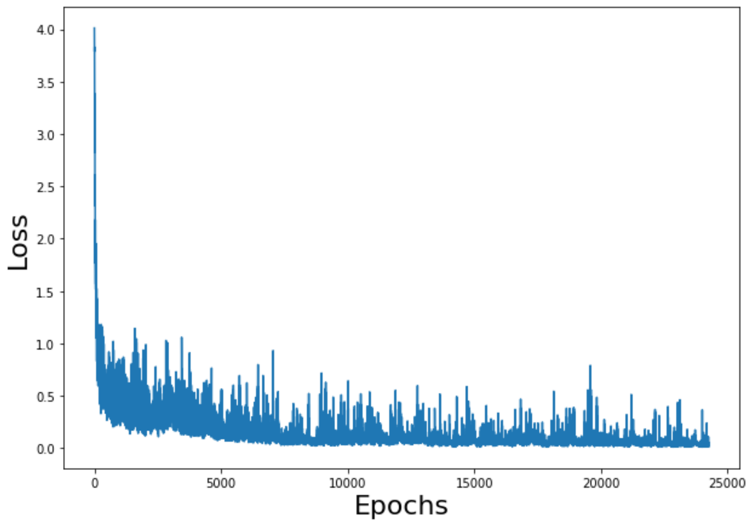


Figure 7.6: Loss plot for random mode and version 2 of the DQN

```
Games played: 1000, # of wins: 942
Win percentage: 94.19%
```

- **Version 3**

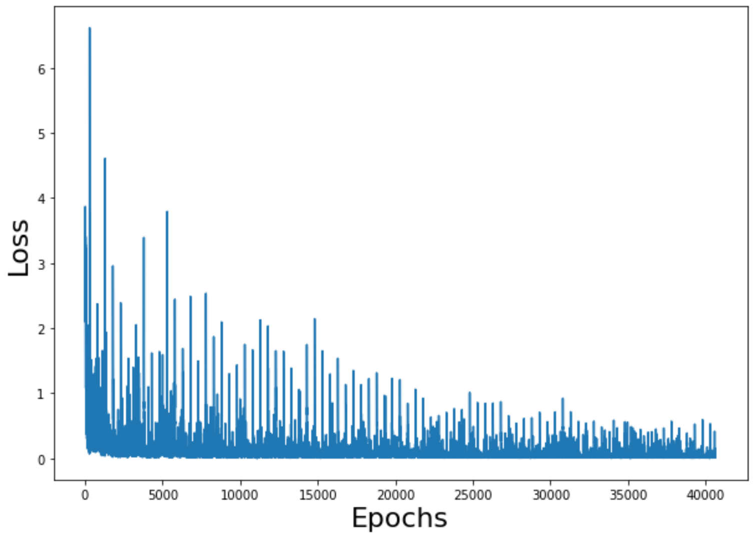


Figure 7.7: Loss plot for random mode and version 3 of the DQN

```
Games played: 1000, # of wins: 908
Win percentage: 90.8%
```

We notice that the simplest version of DQN do not perform well in the random case, the total winning percentage is 84.3%. This is mainly due to *catastrophic forgetting*, when two game states are very similar and yet lead to very different outcomes, the Q function will get *confused* and will not be able to learn what to do. This is actually a very important issue associated with gradient descent-based training methods in online training. Online training is what we've been doing: we backpropagate after each move as we play the game. To avoid *catastrophic forgetting* we use learning using *experience replay*, which clearly arises improvements with a winning percentage of 94.19%. Finally, when using a *target network*, a copy of the main DQN that we use to stabilize the update rule for training the main DQN, we get a lower winning percentage, 90.8%, and has noticeable spikes of error that coincide when the target synchronizes with the main DQN. This might suggest that for this basic example, using *experience replay* is well enough.

Full code is available in *Appendix B*.

Conclusions and improvements

The objective of this project was none other than to introduce the student to the leading field of deep reinforcement learning. The idea of the project was to build from scratch the theoretical basis necessary for the subsequent understanding of what are known as Deep Q-Networks, and apply it to simple cases that show the potential of these techniques.

For this, in the first chapters we introduced deep learning, reviewing the convolutional neural network model that is used to extract the necessary inputs to feed Deep Q-Networks, and the basic concepts of reinforcement learning. The starting point of reinforcement learning is the agent-environment interaction that arises naturally when they meet. To formalize it, we use finite Markov processes, which is the problem we intend to solve with the algorithms that are reviewed below: dynamic programming, Monte Carlo methods and TD-learning. Basically, these methods seek to find an optimal way of behaving for the agent when facing his environment, a policy. We were able to verify that in dynamic programming, we end up posing a system of equations that do not always have a solution, or whose solution is difficult to calculate. That is why we need models that provide a different approach to the exact one and that maintain convergence properties towards an optimal policy and value function. The application of Monte Carlo and TD-learning methods are reviewed for this purpose. These techniques are an improvement in that you do not need a transition model to learn when implementing them. In addition, they allow computing to be done in a lighter way, becoming able to be implemented, as discussed in some cases, in real time. A distinction is made between prediction, evaluating how good a state is, and control, optimizing behavior. Mainly, for this, the following algorithms were reviewed, and versions under certain hypotheses, providing the pseudocode for most of them:

- Every visit Monte Carlo
- First visit Monte Carlo
- On-policy control Monte Carlo
- Off-policy control Monte Carlo
- TD(0)
- SARSA (on-policy TD-learning)
- Q-learning (off-policy TD-learning)

The performance of the algorithms can vary when faced with the same environment, as was seen in the BlackJack example. However, it is the last of them, Q-learning, which was of most interest to us, since it is the basis of the Deep Q-Networks reviewed in the last chapter. Deep Q-Networks arise from the need to deal with problems with a large set of states and actions and from the intention of implementing reinforcement learning in real time. In order to effectively attack these types of problems, Deep Q-Networks use experience or memory replay and learning with two networks of the same architecture. As an example they discussed how these techniques were applied by the *DeepMind* team in Atari games and an example about a Gridworld.

Therefore, it can be considered that the objective that was set for the project has been achieved, and even exceeded. However, everything is subject to possible improvements and this project was not going to be different. Despite the extension of the project, mainly due to the fact that it has been tried to be meticulous following the steps to build a solid base, there are many other algorithms such as TD(λ), n-step bootsrapping and especially solution methods. approximate that could not be reviewed. It would also have been interesting to refer to how to face the exploration versus exploitation dilemma and the correct way of planning to apply these methods. On the other hand, Deep Q-Networks are the most basic model, more complex structures such as Double Deep Q Networks or Dueling Deep Q Networks could have been revised. Finally, it would be good to have given another approach to the problem by applying Game Theory models or reviewing a totally different perspective such as that proposed by Sergio Hernández Cerezo and Guillem Duran Ballester in their recent *Fractal AI* theory (see [25]).

Appendix A

Python code for BlackJack example

```python
import gym
from gym import envs
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as mpl
from matplotlib import cm
from collections import defaultdict
from IPython.display import clear_output
%matplotlib inline
from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.mplot3d import Axes3D


def calc_payoffs(env,rounds,players,pol):
    """ Average payoff """
    average_payouts = []
    for player in range(players):
        rd = 1
        total_payout = 0
        while rd <= rounds:
            action = np.argmax(pol(env._get_obs()))
            obs, payout, is_done, _ = env.step(action)
            if is_done:
                total_payout += payout
                env.reset()
                rd += 1
        average_payouts.append(total_payout)
    print ("Average payout after {} rounds is {}"
           .format(rounds, sum(average_payouts)/players))


def plot_policy(policy):
    def get_Z(player_hand, dealer_showing, usable_ace):
        if (player_hand, dealer_showing, usable_ace) in policy:
            return policy[player_hand, dealer_showing, usable_ace]
        else:
            return 1
    def get_figure(usable_ace, ax):
        x_range = np.arange(1, 11)
        y_range = np.arange(11, 22)
        X, Y = np.meshgrid(x_range, y_range)
        Z = np.array([[get_Z(player_hand, dealer_showing, usable_ace)
                       for dealer_showing in x_range]
                          for player_hand in range(21, 10, -1)])
        surf = ax.imshow(Z, cmap=plt.cm.RdYlGn, vmin=0, vmax=1,
                         extent=[0.5, 10.5, 10.5, 21.5])
        plt.xticks(x_range, ('A', '2', '3', '4', '5', '6', '7', '8', '9', '10'))
        plt.yticks(y_range)
        ax.set_xlabel('Dealer Showing')
        ax.set_ylabel('Player Hand')
        ax.grid(color='w', linestyle='-', linewidth=1)
        divider = make_axes_locatable(ax)
        cax = divider.append_axes("right", size="5%", pad=0.1)
        cbar = plt.colorbar(surf, ticks=[0, 1], cax=cax)
        cbar.ax.set_yticklabels(['0 (STICK)','1 (HIT)'])
        cbar.ax.invert_yaxis()
    fig = plt.figure(figsize=(12, 12))
    ax = fig.add_subplot(121)
    ax.set_title('Usable Ace', fontsize=16)
    get_figure(True, ax)
    ax = fig.add_subplot(122)
    ax.set_title('No Usable Ace', fontsize=16)
    get_figure(False, ax)
    plt.show()

# Monte Carlo

## On-policy

def create_epsilon_greedy_action_policy(env,Q,epsilon):
    def policy(obs):
        P = np.ones(env.action_space.n, dtype=float) * epsilon / env.action_space.n
        best_action = np.argmax(Q[obs])  #get best action
        P[best_action] = 1 - epsilon + (epsilon / env.action_space.n )
        return P
    return policy


def On_pol_mc_control_learn(env, episodes, discount_factor, epsilon):

    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    pol = create_epsilon_greedy_action_policy(env,Q,epsilon)

    for i in range(1, episodes + 1):
        if i% 1000 == 0:
            print("\rEpisode {}/{}.".format(i, episodes), end="")
            clear_output(wait=True)

        # Generate an episode.
        episode = []
        state = env.reset()
        for t in range(100):
            probs = pol(state)
            action = np.random.choice(np.arange(len(probs)), p=probs)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        sa_in_episode = set([(tuple(x[0]), x[1]) for x in episode])
        for state, action in sa_in_episode:
            sa_pair = (state, action)
            #First Visit MC
            first_occurence_idx = next(i for i,x in enumerate(episode)
                                       if x[0] == state and x[1] == action)
            # Sum up all rewards since the first occurance
            G = sum([x[2]*(discount_factor**i)
                     for i,x in enumerate(episode[first_occurence_idx:])])
            # Calculate average return for this state over all sampled episodes
            returns_sum[sa_pair] += G
            returns_count[sa_pair] += 1.0
            Q[state][action] = returns_sum[sa_pair] / returns_count[sa_pair]

    return Q, pol

env = gym.make('Blackjack-v0')

env.reset()
Q_on_pol,On_MC_Learned_Policy = On_pol_mc_control_learn(env, 500000, 0.9, 0.05)

env.reset()
calc_payoffs(env,1000,1000,On_MC_Learned_Policy)

on_pol = {key: np.argmax(On_MC_Learned_Policy(key)) for key in Q_on_pol.keys()}
plot_policy(on_pol)

## Off-policy

def create_random_policy(nA):
    A = np.ones(nA, dtype=float) / nA
    def policy_fn(obs):
        return A
    return policy_fn

def create_greedy_action_policy(env,Q):
    def policy(obs):
        P = np.zeros_like(Q[obs], dtype=float)
        best_action = np.argmax(Q[obs])
        P[best_action] = 1
        return P
    return policy

def Off_pol_mc_control_learn(env, num_episodes, policy, discount_factor):

    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    C = defaultdict(lambda: np.zeros(env.action_space.n))
    target_policy = create_greedy_action_policy(env,Q)

    for i_episode in range(1, num_episodes + 1):
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            clear_output(wait=True)

        # Generate an episode
        episode = []
        state = env.reset()
        for t in range(100):
            # Sample an action from our policy
            probs = target_policy(state)
            action = np.random.choice(np.arange(len(probs)), p=probs)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        G = 0.0 # Sum of discounted returns
        W = 1.0 # Importance sampling ratio

        # For each step in the episode, backwards
        for t in range(len(episode))[::-1]:
            state, action, reward = episode[t]
            # Update the total reward since step t
            G = discount_factor * G + reward
            # Update weighted importance sampling formula denominator
            C[state][action] += W
            # Update the action-value function using the incremental update formula
            # This also improves our target policy which holds a reference to Q
            Q[state][action] += (W / C[state][action]) * (G - Q[state][action])
            # If the action taken by the policy is not the action
            # taken by the target policy the probability will be 0 and we can break
            if action !=  np.argmax(target_policy(state)):
                break
            W = W * 1./policy(state)[action]

    return Q, target_policy

env = gym.make('Blackjack-v0')
env.reset()
rand = create_random_policy(env.action_space.n)
Q_off_Pol,off_MC_Learned_Policy = Off_pol_mc_control_learn(env, 500000, rand,0.9)

env.reset()
calc_payoffs(env,1000,1000,off_MC_Learned_Policy)

pol_test = {key: np.argmax(off_MC_Learned_Policy(key)) for key in Q_off_Pol.keys()}
```

```
plot_policy(pol_test)

# TD Learning

## SARSA

def create_epsilon_greedy_action_policy(env,Q,epsilon):
    def policy(obs):
        P = np.ones(env.action_space.n, dtype=float) * epsilon / env.action_space.n
        best_action = np.argmax(Q[obs])
        P[best_action] = 1 - epsilon + (epsilon / env.action_space.n )
        return P
    return policy

def SARSA(env, episodes, epsilon, alpha, gamma):

    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    pol = create_epsilon_greedy_action_policy(env,Q,epsilon)

    for i in range(1, episodes + 1):
        if i% 1000 == 0:
            print("\rEpisode {}/{}.".format(i, episodes), end="")
            clear_output(wait=True)

        curr_state = env.reset()
        probs = pol(curr_state)
        curr_act = np.random.choice(np.arange(len(probs)), p=probs)
        while True:
            next_state,reward,done,_ = env.step(curr_act)
            next_probs = create_epsilon_greedy_action_policy(env,Q,epsilon)(next_state)
            next_act = np.random.choice(np.arange(len(next_probs)),p=next_probs)
            td_target = reward + gamma * Q[next_state][curr_act]
            td_error = td_target - Q[curr_state][curr_act]
            Q[curr_state][curr_act] = Q[curr_state][curr_act] + alpha * td_error
            if done:
                break
            curr_state = next_state
            curr_act = next_act
    return Q, pol

env = gym.make('Blackjack-v0')
env.reset()
Q_SARSA,SARSA_Policy = SARSA(env, 500000, 0.1, 0.1,0.95)

env.reset()
calc_payoffs(env,1000,1000,SARSA_Policy)

pol_sarsa = {key: np.argmax(SARSA_Policy(key)) for key in Q_SARSA.keys()}
print("SARSA Learning Policy")
plot_policy(pol_sarsa)
```

```python
## Q-learning

def off_pol_TD_Q_learn(env, episodes, epsilon, alpha, gamma):

    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    pol = create_epsilon_greedy_action_policy(env,Q,epsilon)

    for i in range(1, episodes + 1):
        if i% 1000 == 0:
            print("\rEpisode {}/{}.".format(i, episodes), end="")
            clear_output(wait=True)

        curr_state = env.reset()
        while True:
            probs = pol(curr_state)
            curr_act = np.random.choice(np.arange(len(probs)), p=probs)
            next_state,reward,done,_ = env.step(curr_act)
            next_act = np.argmax(Q[next_state])
            td_target = reward + gamma * Q[next_state][next_act]
            td_error = td_target - Q[curr_state][curr_act]
            Q[curr_state][curr_act] = Q[curr_state][curr_act] + alpha * td_error
            if done:
                break
            curr_state = next_state
    return Q, pol

env = gym.make('Blackjack-v0')
env.reset()
Q_QLearn,QLearn_Policy = off_pol_TD_Q_learn(env, 500000, 0.1, 0.1,0.95)

env.reset()
calc_payoffs(env,1000,1000,QLearn_Policy)

pol_QLearn = {key: np.argmax(QLearn_Policy(key)) for key in Q_QLearn.keys()}
print("Off-Policy Q Learning Policy")
plot_policy(pol_QLearn)
```

Based on the publication of Ang Peng Seng: *Blackjack Strategy using Reinforcement Learning*

Appendix B

Python code for Gridworld example

```
import numpy as np
import torch
from Gridworld import Gridworld
from IPython.display import clear_output
import random
from matplotlib import pylab as plt
from collections import deque


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

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

gamma = 0.9
epsilon = 1.0

action_set = {
    0: 'u',
    1: 'd',
    2: 'l',
    3: 'r',
}


def test_model(model, mode='random', display=True):
    i = 0
    test_game = Gridworld(mode=mode)
    state_ = test_game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
    state = torch.from_numpy(state_).float()
    if display:
        print("Initial State:")
        print(test_game.display())
    status = 1
    while(status == 1):
        qval = model(state)
        qval_ = qval.data.numpy()
        action_ = np.argmax(qval_)
        action = action_set[action_]
        if display:
            print('Move # %s; Taking action: %s' % (i, action))
        test_game.makeMove(action)
        state_ = test_game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
        state = torch.from_numpy(state_).float()
        if display:
            print(test_game.display())
        reward = test_game.reward()
        if reward != -1:
            if reward > 0:
                status = 2
                if display:
                    print("Game won! Reward: %s" % (reward,))
            else:
                status = 0
                if display:
                    print("Game LOST. Reward: %s" % (reward,))
        i += 1
        if (i > 15):
            if display:
                print("Game lost; too many moves.")
            break
    win = True if status == 2 else False
    return win

epochs = 5000
losses = []
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
    state1 = torch.from_numpy(state_).float()
    status = 1
    while(status == 1):
        qval = model(state1)
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_]
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
        with torch.no_grad():
            newQ = model(state2.reshape(1,64))
        maxQ = torch.max(newQ)
        if reward == -1:
            Y = reward + (gamma * maxQ)
        else:
            Y = reward
        Y = torch.Tensor([Y]).detach()
        X = qval.squeeze()[action_]
        loss = loss_fn(X, Y)
        optimizer.zero_grad()
        loss.backward()
        losses.append(loss.item())
        optimizer.step()
        state1 = state2
        if reward != -1:
            status = 0
    if epsilon > 0.1:
        epsilon -= (1/epochs)


plt.figure(figsize=(10,7))
plt.plot(losses)
plt.xlabel("Epochs",fontsize=22)
plt.ylabel("Loss",fontsize=22)

max_games = 1000
wins = 0
for i in range(max_games):
    win = test_model(model, mode='random', display=False)
    if win:
        wins += 1
win_perc = float(wins) / float(max_games)
print("Games played: {0}, # of wins: {1}".format(max_games,wins))
print("Win percentage: {}%".format(100.0*win_perc))

test_model(model, 'random')





epochs = 5000
losses = []
mem_size = 1000
batch_size = 200
replay = deque(maxlen=mem_size)
max_moves = 50
h = 0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64)
      + np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float()
    status = 1
    mov = 0
    while(status == 1):
        mov += 1
        qval = model(state1)
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_]
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64)
          + np.random.rand(1,64)/100.0
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
        done = True if reward > 0 else False
        exp =  (state1, action_, reward, state2, done)
        replay.append(exp)
        state1 = state2

        if len(replay) > batch_size:
            minibatch = random.sample(replay, batch_size)
            state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
            action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
            reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
            state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
            done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])

            Q1 = model(state1_batch)
            with torch.no_grad():
                Q2 = model(state2_batch)

            Y = reward_batch + gamma * ((1 - done_batch) * torch.max(Q2,dim=1)[0])
            X = Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze()
            loss = loss_fn(X, Y.detach())
            clear_output(wait=True)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.item())
            optimizer.step()

        if reward != -1 or mov > max_moves:
            status = 0
            mov = 0
losses = np.array(losses)


plt.figure(figsize=(10,7))
plt.plot(losses)
plt.xlabel("Epochs",fontsize=22)
plt.ylabel("Loss",fontsize=22)


max_games = 1000
wins = 0
for i in range(max_games):
    win = test_model(model, mode='random', display=False)
    if win:
        wins += 1
win_perc = float(wins) / float(max_games)
print("Games played: {0}, # of wins: {1}".format(max_games,wins))
print("Win percentage: {}%".format(100.0*win_perc))


test_model(model, mode='random')


import copy

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

model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3,l4)
)

model2 = copy.deepcopy(model) #A
model2.load_state_dict(model.state_dict()) #B

loss_fn = torch.nn.MSELoss()
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

gamma = 0.9
epsilon = 0.3


from collections import deque
epochs = 5000
losses = []
mem_size = 1000
batch_size = 200
replay = deque(maxlen=mem_size)
max_moves = 50
h = 0
```

```
sync_freq = 500 #A
j=0
for i in range(epochs):
    game = Gridworld(size=4, mode='random')
    state1_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
    state1 = torch.from_numpy(state1_).float()
    status = 1
    mov = 0
    while(status == 1):
        j+=1
        mov += 1
        qval = model(state1)
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_]
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/100.0
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
        done = True if reward > 0 else False
        exp =  (state1, action_, reward, state2, done)
        replay.append(exp) #H
        state1 = state2

        if len(replay) > batch_size:
            minibatch = random.sample(replay, batch_size)
            state1_batch = torch.cat([s1 for (s1,a,r,s2,d) in minibatch])
            action_batch = torch.Tensor([a for (s1,a,r,s2,d) in minibatch])
            reward_batch = torch.Tensor([r for (s1,a,r,s2,d) in minibatch])
            state2_batch = torch.cat([s2 for (s1,a,r,s2,d) in minibatch])
            done_batch = torch.Tensor([d for (s1,a,r,s2,d) in minibatch])
            Q1 = model(state1_batch)
            with torch.no_grad():
                Q2 = model2(state2_batch) #B

            Y = reward_batch + gamma * ((1-done_batch) * torch.max(Q2,dim=1)[0])
            X = Q1.gather(dim=1,index=action_batch.long().unsqueeze(dim=1)).squeeze()
            loss = loss_fn(X, Y.detach())
            clear_output(wait=True)
            optimizer.zero_grad()
            loss.backward()
            losses.append(loss.item())
            optimizer.step()

            if j % sync_freq == 0: #C
                model2.load_state_dict(model.state_dict())
        if reward != -1 or mov > max_moves:
            status = 0
            mov = 0

losses = np.array(losses)

plt.figure(figsize=(10,7))
plt.plot(losses)
plt.xlabel("Epochs",fontsize=22)
plt.ylabel("Loss",fontsize=22)


max_games = 1000
wins = 0
for i in range(max_games):
    win = test_model(model, mode='random', display=False)
    if win:
        wins += 1
win_perc = float(wins) / float(max_games)
print("Games played: {0}, # of wins: {1}".format(max_games,wins))
print("Win percentage: {}%".format(100.0*win_perc))
```

The code is from book [24].

References

[1] R.S. Sutton, A.G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[2] F. Rosenblatt, Principles of neurodynamics. Perceptrons and the theory of brain mechanisms, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[3] C.C. Tappert, Who is the father of deep learning?, (2019) 343–348.

[4] N.N. Schraudolph, P. Dayan, T.J. Sejnowski, Temporal difference learning of position evaluation in the game of go, Advances in Neural Information Processing Systems. (1994) 817–817.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, et al., Playing atari with deep reinforcement learning, arXiv Preprint arXiv:1312.5602. (2013).

[6] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, et al., Mastering the game of go with deep neural networks and tree search, Nature. 529 (2016) 484–489.

[7] N. Brown, T. Sandholm, Superhuman ai for multiplayer poker, Science. 365 (2019) 885–890.

[8] M. Moravcik, M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, et al., Deepstack: Expert-level artificial intelligence in heads-up no-limit poker, Science. 356 (2017) 508–513.

[9] A.E. Sallab, M. Abdou, E. Perot, S. Yogamani, Deep reinforcement learning framework for autonomous driving, Electronic Imaging. 2017 (2017) 70–76.

[10] X. Pan, Y. You, Z. Wang, C. Lu, Virtual to real reinforcement learning for autonomous driving, arXiv Preprint arXiv:1704.03952. (2017).

[11] S. Gu, E. Holly, T. Lillicrap, S. Levine, Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates, in: 2017 Ieee International Conference on Robotics and Automation (Icra), IEEE, 2017: pp. 3389–3396.

[12] J. He, J. Chen, X. He, J. Gao, L. Li, L. Deng, et al., Deep reinforcement learning with a natural language action space, arXiv Preprint arXiv:1511.04636. (2015).

[13] Z. Zhang, S. Zohren, S. Roberts, Deep reinforcement learning for trading, The Journal of Financial Data Science. 2 (2020) 25–40.

[14] Y. Deng, F. Bao, Y. Kong, Z. Ren, Q. Dai, Deep direct reinforcement learning for financial signal representation and trading, IEEE Transactions on Neural Networks and Learning Systems. 28 (2016) 653–664.

[15] Z. Zhou, X. Li, R.N. Zare, Optimizing chemical reactions with deep reinforcement learning, ACS Central Science. 3 (2017) 1337–1344.

[16] V. François-Lavet, Contributions to deep reinforcement learning and its applications in smartgrids, PhD thesis, Universite de Liege, Liege, Belgique, 2017.

[17] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N.J. Yuan, X. Xie, et al., DRN: A deep reinforcement learning framework for news recommendation, in: Proceedings of the 2018 World Wide Web Conference, 2018: pp. 167–176.

[18] A. Torrejón Valenzuela, Teoría de juegos, B.S. thesis, Universidad de Sevilla, 2020. https://torrejonvalenzuela.com/pdf/TFG.pdf.

[19] Z. Gao, Understanding the future of deep reinforcement learning from the perspective of game theory, in: Journal of Physics: Conference Series, IOP Publishing, 2020: p. 012076.

[20] G. Tesauro, TD-gammon, a self-teaching backgammon program, achieves master-level play, Neural Computation. 6 (1994) 215–219.

[21] C. Wang, K. Ross, On the convergence of the monte carlo exploring starts algorithm for reinforcement learning, arXiv Preprint arXiv:2002.03585. (2020).

[22] C.J.C.H. Watkins, Learning from delayed rewards, (1989).

[23] M. Sewak, Deep reinforcement learning, Springer, 2019.

[24] A. Zai, B. Brown, Deep reinforcement learning in action, Manning Publications, 2020.

[25] S.H. Cerezo, G.D. Ballester, Fractal ai: A fragile theory of intelligence, arXiv Preprint arXiv:1803.05049. (2018).

[26] P.L. Luque-Calvo, Escribir un trabajo fin de estudios con r markdown, Disponible en http://destio.us.es/calvo, 2017.