Mathematical Biology

Master thesis

On the performance of Bandit and Reinforcement Learning algorithms

with applications in the biomedical sciences

by

Vincent Hennink

2019

Supervisor: prof. dr. S. Bhulai

Second reviewer: dr. R. Hindriks

Afdeling Wiskunde
Faculteit der Beta Wetenschappen

VU VRIJE UNIVERSITEIT AMSTERDAM

Contents

1 Abstract

The temporal difference learning and Monte Carlo method are important algorithms in reinforcement learning. Despite the importance of both algorithms, their rate of convergence has not been studied extensively in Markov reward processes. This thesis, therefore reports on the rate of convergence of these algorithms in different types of Markov reward processes. We find that the rate of convergence of these algorithms depends strongly on the structure of the Markov reward process. The second part of the thesis compares the performance of bandit algorithms to human decision making in a learning task, that is known as the Iowa gambling task. We found that all relevant bandit algorithms outperform the human participants. In addition, we found that the Bayesian-based bandit algorithms outperform the frequentist-based bandit algorithms significantly. Finally, this thesis examines the possibility of using bandit algorithms, in designing a clinical trial. We indeed found empirical confirmation of their effectiveness in designing a clinical trial, under certain constraints.

2 Introduction

In recent years, there has been an enormous increase of interest in statistical learning. Although the theory of statistical learning goes back to the 1950s, it is only in recent years that this field has become the center of attention for mathematicians, computer scientists, and scientists in general. This is with good reason, given the potential of statistical learning to solve relevant problems. The theory of statistical learning, which is more commonly known as machine learning, has already been applied to different tasks with tremendous success. Interesting tasks include for instance: face recognition, self-driving cars, robotic vacuum cleaners, and advertisement recommendation. In addition, machine learning (ML) algorithms have been used to address problems in biology. Relevant problems that have been addressed by ML include: tumor classification, personalized medicine, image segmentation, and behavior modeling. All problems except for the tumor classification are approached within the reinforcement learning framework. This framework will be explored in more detail in the subsequent section. Till this day, researchers are still actively working within this framework, to improve on the used algorithms, their accuracy, and on the rate of convergence of these algorithms.

Broadly speaking there are two main methods to utilize within the reinforcement learning framework: the temporal difference methods (TD methods) and the Monte Carlo methods (MC methods). These methods will be discussed in a later chapter. Singh and Dayan [1], found that TD methods are generally speaking more efficient than MC methods. Many mathematicians and computer scientists have argued in fact that TD methods are more efficient than MC methods. That is, TD methods converge faster than MC methods. However, Beleznay et al. [2] have revealed that there are situations in which MC methods converge faster than TD methods. This situation will be discussed in chapter 4. In practice, researchers still often use TD methods to address their problems. Komorowski et al. [3] have, for instance, shown that TD methods can be used to select the optimal treatment strategy for patients in the intensive care, that are affected by sepsis. Furthermore, Sahba et al. [4] have applied TD methods to segment a transrectal ultrasound images. These are just two research papers of a plethora of papers, that have used TD methods to solve problems in the biological sciences.

A key problem in reinforcement learning is that there are many results that indicate that TD methods are in general more efficient than Monte Carlo methods. However, no proof of this is given. The results that show that TD methods are more efficient than MC methods are often empirical in nature. That is, by using simulations one can show that TD methods converge faster, in general, than MC methods. It remains somewhat unclear if this can be proven formally. Moreover, it is unknown in which setting MC converges faster than TD and vice versa. Another topic that has not extensively been

studied in reinforcement learning, is that of comparing human decision making against artificial decision making. In particular, this has not extensively been examined in the so-called Iowa gambling task (IGT). What this learning task precisely entails will become clear later on.

The aim of this thesis is twofold. Firstly, we want to expand on the current knowledge of reinforcement learning algorithms. In particular, we will compare the rate of convergence of TD methods with those of MC methods under different circumstances. Secondly, we will apply reinforcement learning algorithms to a learning task (the IGT) and will compare its performance with its human counterpart. Here, we will evaluate the effectiveness of using reinforcement learning algorithms for this decision-making task. Moreover, inspired by the results of the reinforcement learning algorithms we will discuss the utility of these algorithms in designing clinical trials. The former problem is more theoretical of nature, whereas the latter two problems are both practical and theoretical in their nature.

This thesis has been divided into five main parts. The first part is the introduction. In this section, we will give a background in both reinforcement learning and in the relevant principles of neuroscience. Reinforcement learning is the general framework in which we will work during this thesis. Thus, a proper introduction is in order. The neuroscience section will discuss the relevant biological background of learning/decision-making in more detail. The second part will lay out the mathematical framework in which we will work. In this part, we will discuss Markov decision processes (MDPs) and dynamic programming (DP). After this has been discussed we will focus our attention on how the different reinforcement learning algorithms work. Moreover, we will examine the mathematical properties of different reinforcement learning algorithms. Next, we discuss the mathematical properties of TD and MC methods. More specifically, we will examine if these algorithms converge and under which conditions these algorithms converge. What the term converges exactly entails, will become clear later on. The third part will deal with the rate of convergence of TD and MC methods under different settings. Here, we will give proofs on the rate of convergence for both methods. In addition, we will use simulations to compare MC methods with Q-learning in a random walk setting. The fourth part will be devoted to the applications of reinforcement learning algorithms to the Iowa gambling task and to the design of a clinical trial. The goal of this section is to inform the reader, about the impact reinforcement learning algorithms can have in the biomedical domain. In the last part of this thesis, we will summarize and discuss our findings of the various reinforcement learning algorithms. Furthermore, we will evaluate the performance of the trained agent on the Iowa gambling task. In this section, we will also compare the effectiveness of different reinforcement learning algorithms on this task. Here, we will also report on the performance of reinforcement learning algorithms in designing a clinical trial. But, before we can discuss any of this, we must have a clear understanding of what reinforcement learning entails and how it can be used in the biomedical domain. This will be discussed in the next few subsections.

2.1 Statistical learning

Statistical learning (also known as machine learning) can broadly be divided into three categories. These categories are: supervised learning, unsupervised learning and reinforcement learning (RL). In supervised learning, one learns from a training set of data. Each instance in the training set consists of variables that are known as features and a single target variable. During the training phase, the learner tries to find a model that fits the data best. Moreover, during this phase the model learns the relationship between the features and the target, so that it can later generalize to unseen cases. This occurs on the test set. The test set is a dataset that only contains the features and does not have any target variable. In this phase, the model must make predictions on the value of the target variable. After this phase, one can compute the performance of the model and see how accurate it was in making its prediction. Supervised learning problems occur quite frequently in practice. Typical supervised problems include classification and regression problems. The formerly mentioned problem requires the learner to map features to a certain class. For example, one can use supervised learning for tumor classification. If the learner is given certain tumor characteristics (cell shape, nucleus shape, mitosis activity etc.), then it can predict whether the tumor is benign or malignant by using supervised learning algorithms. A typical regression problem is, for instance, predicting house prices based upon a set of variables (area of house, location, etc.). RL differs from supervised learning in the sense that RL is about sequential decision-making. This entails that given an input, or state, the learner makes a decision/action (based upon some policy) to get to a new input, or state. The next state depends on the decision made by the agent. Hence, in RL the decision made by the agent affects the future. This is not the case for supervised learning. Thus, RL is different from supervised learning.

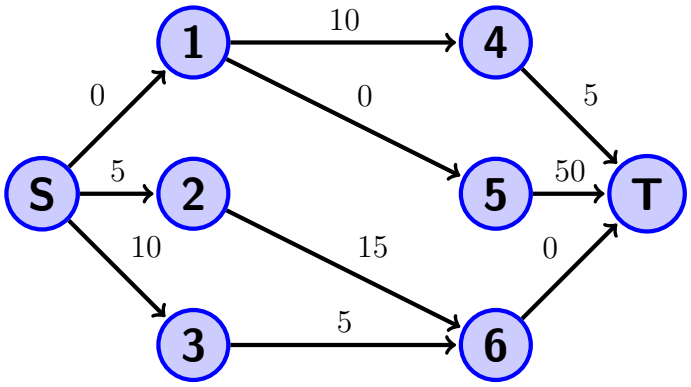
RL is also different from unsupervised learning. In unsupervised learning the learner has unlabeled data at its disposal. In contrast to supervised learning, unsupervised learning has no target given in the dataset. Only features are present. The goal for the learner, in this case, is to find hidden structures within the data. For example, given a dataset that contains the expression of many genes, one can invoke unsupervised learning methods to group genes that share characteristics. Unsupervised learning is all about finding structure within datasets, that are potentially enormous in size. RL also significantly differs from unsupervised learning. In RL, the agent has as goal to choose actions in a sequential decision-making task in order to maximize a certain reward signal. It does not try to find a hidden structure within the data.

We have seen that reinforcement learning differs from both supervised, and unsupervised learning. Hence, it is fair to consider reinforcement learning as its own learning paradigm, rooted within statistical learning. This will be done in the next subsection.

2.1.1 Reinforcement Learning

In reinforcement learning, the learner is interacting with an environment, with the intent to attain a long-term goal. This goal is related to the state of the environment. RL is about learning from experience. The learning agent must take actions such that the reward is maximized in the long run. The agent, however, is not told a priori which action it should take in order to achieve this goal. Instead, the agent will try different actions. In this way, the learning agent will discover which action gives the highest reward in the long run. It can then choose to take this action with high probability in the future. In other words, the agent must learn from experience before it can select an action with a high-reward payoff. In reinforcement learning there must be a goal for the learner. The goal of the task is defined by the reward signal. This is the signal that the learner wants to maximize. In addition, the agent must be capable of observing the state of the environment, either partially or fully. It must also be capable of performing actions that influence the state of the environment.

As was already stated previously, we want to get as much reward as possible. To achieve this, the learner must have a preference for actions that had a high yield of reward in the past. A new challenge that arises in RL, but not in (un)supervised learning, is that of the trade-off between exploration and exploitation. This principle can be made clear if we consider the following graph:



In this graph the vertices represent the states of the environment, the edges represent the actions the agent can take. The numbers next to the arrows indicate the reward the agent receives, if it takes that action in that state. At the start of a trial, the learner begins in state S. The trial comes to an end once the learner hits the terminal state: T. Then a new trial begins again. A learner can choose to be greedy and take the path $\{S, 3, 6, T\}$. But, by choosing this path we get a total reward of 15. The learner can exploit its knowledge of the previous trial and take the same path which guarantees a reward of 15. The learner can, however, obtain a greater reward by following $\{S, 1, 5, T\}$. Therefore, it would be wise for the agent to also explore different actions in the state-action space. But, if the agent explores too much of its time and does not exploit the knowledge it already has from the environment, then it might select actions that will yield less reward relative to the reward it could have received. In short, the learner has

to exploit what it knows, but must also explore in order to take better actions in future trials. This balance between exploration and exploitation that the learner has to keep in mind, is known as the exploration-exploitation trade-off.

In the reinforcement learning framework we have four main elements that we have to consider. These are the **policy, reward signal, value function** and the (optional) **model** of the environment. A policy is a mapping from the state space to the action space. The policy is used by the agent in its decision-making. A policy function can be both deterministic and stochastic. In the former case, the agent always takes the same action in a given state. In the latter case, the agent selects an action with a certain probability. The reward signal is the signal that defines the goal of the agent. At each point in time the agent performs an action and finds itself in a new state. The agent also receives a reward from the action it took in the previous state. The reward must be maximized over the long run, and is also used by the agent to alter its policy. The value function computes as its name suggest, the value of a state. The value of a state S is defined as the total amount of reward the agent can expect to receive over the future, given that the agent starts at state S. The difference between the reward function and the value function is that the reward function states what is good in the short run, whereas the value function states what is good in the long run. As was already discussed previously, the agent must prefer actions that maximize the long-term reward rather than the short-term reward. Finally, there are two main methods to model the environment. These two methods are called the model-based method and the model-free method. The model-based method uses models to capture the dynamics of the environment. This is not done by the model-free methods. These model-free methods learn from the environment by trial and error. Now that we have covered the main ingredients of RL, we can formalize these ideas mathematically. This will be done in chapter 3. In chapter 3, we will use the theory of Markov Decision Processes to make these ideas formal. We will, however, first continue with the next subsection where we will give the reader some background on neuroscience. In this section, we will discuss how decision-making occurs in the human brain. Here, we will also see the similarities and differences between biological and artificial decision-making. The upcoming section serves as additional background information for chapter 5. In this chapter we will analyze the decision-making data of humans at a certain task (the IGT). A precise understanding of the processes that occur in the participant's brains that lead to a decision is not required for the analysis in this chapter. Nonetheless, it is interesting to discuss these processes.

2.2 The biology of learning

The purpose of this section is to give the reader some background in neuroscience. This background is relevant to some degree, because in Chapter 5 we will study the learning/decision-making process in humans. This learning/decision-making process in humans is guided by the nervous system. Since neuroscience is the field of science that studies the nervous system, it is necessary to give some background in neuroscience. The upcoming sections will deal with this topic. We will start by shortly introducing the

smallest object of study in neuroscience: the neuron. Then we will continue our story by introducing larger structures such as brain regions and brain lobes. Here we will talk about how these larger structures are involved with learning/decision-making. But, before we can do this we will introduce the basic unit in neuroscience: the neuron.

2.2.1 The basic element of neuroscience: the neuron

The neuron is a cell that transmits an electrical signal. In the past couple of years entire textbooks have been written about this cell (e.g., Purves et al. [5]). However, there is still a lot we do not know about the neuron. The aim of this subsection is not to give an entire overview of the current body of knowledge of the neuron, but to discuss the fundamental principles that underlie the processes that occur within the neuron. The figure below is a classical representation of the neuron.

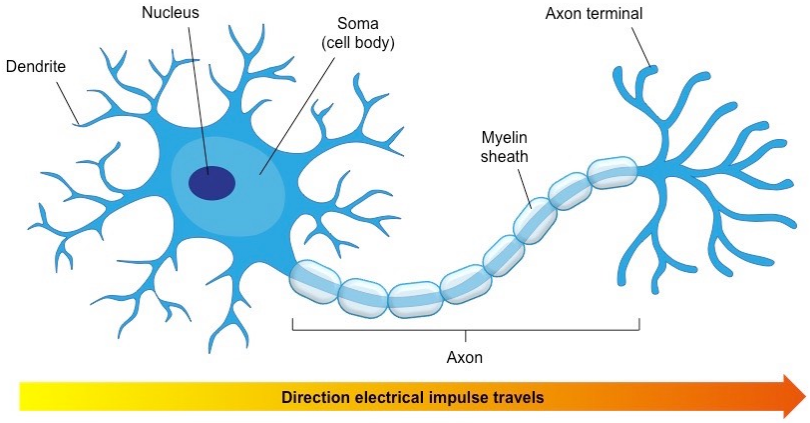


Figure 2.1: Graphical representation of the neuron.

In the figure above we see that the neuron consists of several structures. The most important structures to consider are: the dendrites, the soma (including the nucleus) and the axon. The dendrites are filaments that carry an electric signal to the soma (= the cell body). The axon carries the signal from the soma to an axon terminal, where the signal will be transmitted to another neuron. Note that in the figure above, we see that the axon is enclosed by a myelin sheath. This myelin sheath leads to insulation of the axon, which in turn increases the speed of the electrical conduction of the signal. Another key structure that is worth mentioning, but is not depicted in the figure above is the synapse. The synapse is a structure where transmission of the electrical signal occurs from one neuron to another. The synapse is essential for neurons to communicate with each other. When many neurons (= neural network) communicate with each other, emergent properties such as thinking, planning, sensing and also learning/decision-making can occur. When scientists explain how the brain is responsible for these different emergent properties, they explain it at the (macro-)scale of brain regions/neural network, rather than at the (micro-)scale of the neuron. Since our main interest goes out to decision-making in humans we will turn our attention to the macro-scale of neuroscience. Thus, we will examine the different brain regions that are responsible for these functions in more detail.

2.2.2 Learning/decision-making & the brain

Throughout human history, humans were faced with the challenge to interact with the environment as effectively as possible. This entails that humans had to maximize some reward (resources/food needed to survive) over time, while minimizing the risk for situations that would lead to a bad outcome. In other words, their goal is to learn to prefer actions in certain states that lead to a high reward with high probability. In order to achieve this goal, humans had to develop (learning) strategies that were effectively adapted to the environment. Here we mean with learning strategies the same as the policy function in the reinforcement learning framework. Humans that could construct such a strategy, could also maximize the reward over time more easily compared to humans that could not construct such a learning strategy. This is, of course, a desirable trait from an evolutionary point of view. A fundamental question that arises is: what are the relevant biological components that underlie the construction of a learning strategy? This question boils down to finding the brain regions that are involved with decision-making. A (learning) strategy or policy is nothing more than selecting actions given a certain current state. Therefore, it is relevant to discuss how decision-making occurs in the brain. Although the precise details of decision-making in the brain remains somewhat unknown, there is still a basic understanding of the brain regions that play a key role in decision-making among neuroscientists. In the upcoming section we will introduce and discuss these brain regions.

2.2.3 Relevant brain regions for decision-making

There exists a strong link between artificial reinforcement learning and biological decision-making. In Section 2.1.1 we introduced the four elements that form the fundamental building blocks for reinforcement learning. These four elements were: the policy, reward signal, value function, and model of the environment. These elements also have a biological counterpart to some extent.

An important factor for decision-making is the value function. Most neuroscientists ([6,7,8]) believe that decision-making in the brain is value-based. That is, before we humans take an action the brain first 'computes' the expected value that would follow from this action. This is done for all conceivable actions one can take. The expected rewards for each action are then compared, and the action with the highest expected reward is chosen by the agent. The prediction of the value of each action takes place in several regions of the brain. Most notably the amygdala, orbitofrontal cortex (OFC), ventromedial prefrontal cortex (vmPFC) and the ventral/dorsal striatum. Some of these regions can be found in the figure on the next page.

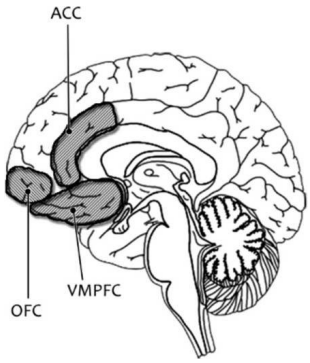


Figure 2.2: Median view of the brain.

All structures listed above play a key role in assigning the value for an action (the value signal). However, it must be noted that the nature of the learning problem very much determines the brain regions that are most relevant. In this thesis, we will examine the so-called Iowa Gambling Task (IGT) in Chapter 5. It turns out that the OFC is a key region for the decision-making process during this task. The OFC can thus be regarded as the most important brain region that is involved with decision-making in our case. Therefore, we will focus our attention to this particular brain region. A first question that naturally arises is: how do we know that this particular brain region is crucial for the decision-making process in the IGT? The answer lies in brain lesion studies. Patients that have lesions (brain damage due to trauma or disease) in the OFC typically perform poorly in the IGT [9]. Healthy individuals on the other hand, do not perform as poorly as these patients on this task. This finding indicates that the OFC is indispensable for decision-making in these particular tasks.

It has been suggested that the OFC plays a key role in processing reward [10]. The OFC integrates various inputs of information relating to the reward outcome to derive a value signal. How the decision-making process occurs can best be clarified by means of an example. For instance, how does one decide or does not decide to consume a beverage? The process begins by integrating several inputs. These inputs provide sensory, affective and motivational information. The gustatory cortex provides information about the (specific) taste of the beverage whereas the amygdala provides information about the pleasantness of the drink. The hypothalamus signals that the drink will quench the thirst. All these inputs are delivered to the OFC. The OFC subsequently computes the value of the outcome and how rewarding it is (the values signal). There are several theories about the physiological details of how this computation occurs, but we will not dive into these theories. Once the value of the outcome is computed, the information is transmitted to the dorsolateral part of the pre-frontal cortex (DLPFC). This section of the cortex is very much involved with the formation of behavioural plans. Furthermore, this part of the brain plays a central role in prioritizing goals. The OFC and the DLPFC provide their information with the median PFC (MPFC). This middle part of PFC can then use information from both the OFC and the DLPFC to determine if an action is worth performing. The DLPFC and MPFC together determine the behavioral response. In our example, the behavioral response would be to either get a drink or not. The behavioral

response, of course, depends on the state of the individual. (For example, if one is thirsty the action would be to take a drink). The process is summarized in Figure 2.3

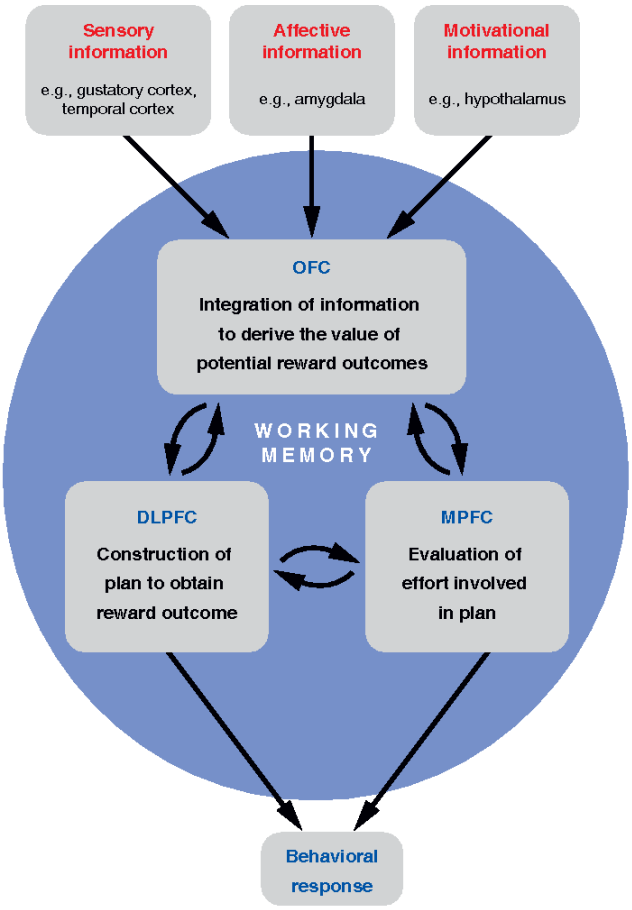


Figure 2.3: Model of the neuronal mechanisms underlying decision-making in PFC adapted from J.D. Wallis [10].

Several remarks are in order. Firstly, most processes we just described occur unconsciously. We do not make a cost-benefit analysis for each conceivable action consciously. Secondly, the processes we just described apply to healthy individuals. It goes without saying that patients with brain lesions in the PFC do not have a normal functioning PFC. Hence, these patients typically suffer from impaired decision-making. Finally, as we already stated previously, decision-making is a broad topic in neuroscience and the relevant brain regions for decision-making strongly depend on the type of task. In the introduction so far, we have discussed the fundamentals on how a biological agent and how an artificial agent makes decisions. In the next chapter we will mathematically formalize the sequential decision-making process for the artificial agent.

3 Mathematical framework

In the previous chapter we have already introduced RL and what its main ingredients are to consider. In this chapter we will make these ideas formal. In addition, proofs of mathematical properties of value functions and RL algorithms will be provided during this chapter.

3.1 Markov Decision Processes

The introduction of RL that was given in the previous chapter can neatly be summarized in the following figure:

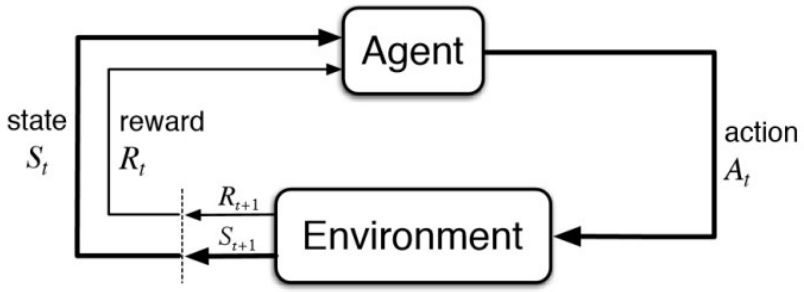


Figure 3.1: Graphical representation of the agent-environment interaction.

The agent finds itself in a state S_t at time step t and takes action A_t. Next (one time step later), the environment gives a reward R_{t+1} and the agent finds itself in state S_{t+1}. This is done until some terminal state S_T is reached. The reward is a function that maps a state and action to a numerical value. That is: $r : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$.
The set of states, actions, and rewards are denoted by $\mathcal{S}, \mathcal{A}$, and $\mathcal{R}$, respectively. There is an uncertainty that the agent reaches the new state $S_{t+1} = s_{t+1}$. This uncertainty is denoted by: $p(S_{t+1} = s'|S_t = s, A_t = a) = p(s'|s, a)$ and is known as the state transition probability. Now we can introduce some definitions.

Definition 3.1 (Finite Markov Decision Process)**.** A four-tuple $\Big(\mathcal{S}, \mathcal{A}, \mathcal{R}, p(s'|s, a)\Big)$ is called a finite Markov decision process (MDP) if $|\mathcal{S}| < \infty$, $|\mathcal{A}| < \infty$, $|\mathcal{R}| < \infty$ and if each set is non-empty.

The dynamics of the environment in an MDP are governed by the transition probability $p(\cdot|s, a)$. Note that the probability of reaching state s_{t+1} only depends on s_t and a_t. Hence, the stochastic process $\{S_t\}_{t=0}^{T}$ (given the previous state and action) is indeed a Markov process since it satisfies the Markov property. The same is true for the stochastic process

$\{R_t\}_{t=0}^{T}$. We have already introduced the reward function. In the finite MDP setting we define the reward function as follows:

$$r(s,a) := \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_r r(s,a,s') \sum_{s'} p(S_{t+1} = s'|S_t = s, A_t = a).$$

We already stated in previous sections, that the goal of the agent is to maximize the return in the long run, and not necessarily in the short run. To capture this idea we will use the following definition for the return G_t.

Definition 3.2 (Return)**.** The return at time t, denoted as G_t , is defined as the sum of the subsequent rewards in time:

$$G_t = \sum_{i=0}^{T-t-1} \gamma^i R_{t+i+1}$$

where $\gamma \in [0,1]$.

A few remarks are in order. Firstly, so far we have only considered episodic tasks, meaning that the task ends at some time T. We can also have a continuing task. This entails that there is no natural ending to the episode. If we are dealing with a task that is a continuing task, then the index of the sum in the definition of the return goes from zero to infinity. Secondly, the term γ is known as the discount factor and is used for weighing the rewards. There are two extreme cases that we can consider. If $\gamma = 0$, then the learner only maximizes the immediate reward. In this case, the learner is short-sighted. In the case where $\gamma = 1$, the learner is far-sighted and tries to maximize all future rewards. When $\gamma \in [0,1]$ we get a combination. Future rewards become exponentially discounted, but are not neglected entirely. A simple result that follows is given below.

Corollary 3.2.1. *If the stochastic process $\{R_t\}_{t=1}^{\infty}$ is bounded, then the return G_t of a continuing task converges given that $\gamma < 1$.*

Proof. The proof can be seen very easily. Since the stochastic process is bounded, we have that: $R_t \leq M \ \ \forall t \geq 1$ for some $M \in \mathbb{R}$. We can use this to determine the upper bound of the return:

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1} \leq M \sum_{i=0}^{\infty} \gamma^i = \frac{M}{1-\gamma}.$$

□

In the episodic case we only have finitely many rewards. Thus the return is always finite in this case. From now on we will simply write

$$G_t = \sum_{i=t+1}^{T} \gamma^{i-t-1} R_t$$

as return for both episodic and continuing tasks. We can let $T \to \infty$ as long as $\gamma < 1$. Or we can choose $\gamma = 1$ if $T < \infty$.

3.1.1 Value functions, action-value functions and policies

The value function is a key notion for this thesis and will be used intensively. Almost all RL algorithms use either the value function or the action function. Before we can introduce these concepts we must give the definition of a stochastic policy $\pi \in \Pi$. A stochastic policy is a mapping $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ and is defined at time t by $\pi = \pi(A_t = a_t | S_t = s_t)$. It denotes the probability for the learner to select action a_t, given that the learner finds itself in state s_t. The policy of an agent is often altered accordingly to the experience it has had. Now we can formally introduce the definition of the value function.

Definition 3.3. [Value function] A value function $v_\pi(s)$ is a mapping, $v_\pi : \mathcal{S} \to \mathbb{R}$ that maps a starting state s to the expected return by following a policy $\pi \in \Pi$. In other words,

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \Big[\sum_{i=0}^{\infty} \gamma^i R_{t+i+1} | S_t = s \Big] \ \ \forall s \in \mathcal{S}.$$

The definition of an action-value function, $q_\pi(s, a)$, is very similar to the definition of the value function:

Definition 3.4 (Action-value function)**.** An action-value function $q_\pi(s, a)$ is a mapping, $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that maps a starting state and starting action to the expected return by following a policy $\pi \in \Pi$ thereafter. In other words,

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \Big[\sum_{i=0}^{\infty} \gamma^i R_{t+i+1} | S_t = s, A_t = a \Big] \ \ \forall s \in \mathcal{S}, \forall a \in \mathcal{A}.$$

Now that we have introduced the value and action-value function we can introduce the definition of the optimal value and action-value function. However, we first introduce the optimal policy. The optimal policy π^* is defined as one would expect: $\pi^* = \arg\max_\pi v_\pi(s)$. The optimal value function and state action value are simply:

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s) \ \ \forall s \in \mathcal{S} \\ q_*(s, a) &= \max_\pi q_\pi(s, a) \ \ \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \end{aligned}$$

Note that $v_*(s) = \max_a q_*(s, a)$ holds by definition. We can use this fact to rewrite the optimal value function. This can be done as follows:

$$\begin{aligned} v_*(s) &= \max_a q_*(s, a) = \max_a \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma \Big(\sum_{i=0}^{\infty} \gamma^i R_{t+i+2} \Big) | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a \Big(\mathbb{E}_{\pi^*}[R_{t+1} | S_t = s, A_t = a] + \gamma \, \mathbb{E}_{\pi^*}[G_{t+1} | S_t = s, A_t = a] \Big) \\ &= \max_a \Big(r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s' | s, a) \, \mathbb{E}_{\pi^*}[G_{t+1} | S_{t+1} = s'] \Big) \\ &= \max_a \Big(r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s' | s, a) v_*(s') \Big). \end{aligned}$$

Now that we rewrote the value function a bit we can use this, to prove the following theorem.

Theorem 3.5. *For a finite MDP there exists a unique optimal value function v_*.*

Proof. The proof of this theorem relies on the Banach fixed point theorem (see appendix). We begin by constructing an operator for which we know that $\mathcal{B}v_* = v_*$ must hold. If we define the operator

$$(\mathcal{B}v)(s) = \max_a \Big(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v(s') \Big)$$

then we know that $\mathcal{B}v_* = v_*$ indeed holds. Thus, v_* is a fixed point. But the question remains: is it unique? And how can we approximate this value function? We can define the following value iteration algorithm. Choose an arbitrary starting value function v_0. The value function is then updated by: $v_{i+1} = \mathcal{B}v_i$. If we prove that $\mathcal{B} : \mathcal{V} \to \mathcal{V}$ is a contraction, then we have proven that the above-described algorithm converges to v_*. Then we have also shown that v_* is the unique fixed point. Here $\mathcal{V}$ is the space that consists of value functions $v : \mathcal{S} \to \mathbb{R}$. We can see that $(\mathcal{B}, \|\cdot\|_\infty)$ is a Banach space. Here $\|v\|_\infty = \max_s |v|$ denotes the standard max norm. Before we can apply Banach's fixed point theorem, we must show that $\mathcal{B}$ is a contraction. Thus we must show that $\|\mathcal{B}v - \mathcal{B}v'\|_\infty \leq \gamma \|v - v'\|_\infty$ holds for some $\gamma < 1$. This is indeed the case. We can deduce the following upper bound for $\|\mathcal{B}v - \mathcal{B}v'\|_\infty$:

$$\begin{aligned}
\|\mathcal{B}v - \mathcal{B}v'\|_\infty &\leq \gamma \max_a \| \sum_{s' \in \mathcal{S}} p(s'|s,a) \Big(v(s') - v'(s') \Big) \|_\infty \\
&\leq \gamma \max_{a,s} \sum_{s' \in \mathcal{S}} p(s'|s,a) \, |v(s') - v'(s')| \\
&\leq \gamma \|v - v'\|_\infty .
\end{aligned}$$

Hence, there exists a unique optimal value function v_* and the value iteration algorithm converges to this unique optimal value function. □

Several remarks are in order. First, why do we care about finding the optimal value function at all? This is because if we know the optimal value function (or a close approximation to it) then we also know the optimal policy. The optimal policy can then be found by

$$\pi^* = \arg\max_a \Big(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v_*(s') \Big).$$

If the agent has learned the optimal policy, then it can follow this policy to maximize the expected return. In general, an RL algorithm consists of two main components. The first component is called policy evaluation and the second component is called policy improvement. These two main components will be discussed in more detail in the upcoming section. The second remark is that the value iteration algorithm converges exponentially fast to the unique optimal value function v_*. This can be seen quite easily:

$$\|v_N - v_*\| = \|\mathcal{B}v_{N-1} - v_*\| \leq \gamma \|v_{N-1} - v_*\| \leq \cdots \leq \gamma^N \|v_0 - v_*\|.$$

Thus the error, $\|v_N - v_*\|$, decreases exponentially fast. We can use this observation to prove the following corollary.

Corollary 3.5.1. *The number of iterations (N) needed in a finite MDP for the value iteration algorithm to have an error of at most ϵ is given by:*

$$N = \left\lceil \frac{\log 2M/(\epsilon(1-\gamma))}{\log 1/\gamma} \right\rceil.$$

Proof. We already saw that we have

$$\|v_N - v_*\| \leq \gamma^N \|v_0 - v_*\|$$

as upper bound for $\|v_N - v_*\|$. Now we can use Definition 3.3 and Corollary 3.2.1 to deduce that the maximum bound for the expected return (= value function) is given by $\frac{M}{1-\gamma}$. Here M denotes the maximal return of the stochastic process $\{R_t\}_{t=1}^{\infty}$. Hence, we get:

$$\|v_N - v_*\| \leq \gamma^N \|v_0 - v_*\| \leq \gamma^N (\|v_0\| + \|v_*\|) \leq \frac{2M\gamma^N}{1-\gamma}$$

as bound for $\|v_N - v_*\|$. We want to have this bound smaller or equal to ϵ. If we now perform some basic algebraic operations we indeed get our result:

$$\frac{2M\gamma^N}{1-\gamma} \leq \epsilon \iff -N \log \gamma \geq -\log \epsilon(1-\gamma)/2M \iff N \geq \frac{\log 2M/(\epsilon(1-\gamma))}{\log 1/\gamma}.$$

Hence, if we take $N = \left\lceil \frac{\log 2M/(\epsilon(1-\gamma))}{\log 1/\gamma} \right\rceil$, then we indeed get our result. □

So far we have introduced some formal definitions in reinforcement learning and some results. We have briefly mentioned an algorithm, namely the value iteration algorithm, that can be used to find the optimal policy in an MDP. In the next section we will discuss this algorithm in more detail. Moreover, we will discuss the algorithm to improve on the policy that is followed.

3.2 Dynamic programming

In any MDP we want to find the optimal policy π^*. A way to do this is by utilizing algorithms from Dynamic programming (DP). There are two main algorithms that can be used to find the optimal policy. These two algorithms are presented below:

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

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $temp \leftarrow v(s)$
   $v(s) \leftarrow \sum_{s'} p(s'|s, \pi(s))\Big[r(s, \pi(s), s') + \gamma v(s')\Big]$
   $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $temp \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s, a)\Big[r(s, a, s') + \gamma v(s')\Big]$
   If $temp \neq \pi(s)$, then *policy-stable* $\leftarrow false$
   If *policy-stable*, then stop and return v and π; else go to 2

Initialize array v arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathcal{S}$:
$temp \leftarrow v(s)$
$v(s) \leftarrow \max_a \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')]$
$\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy π, such that
$\pi(s) = \arg\max_a \sum_{s'} p(s'|s, a)\Big[r(s, a, s') + \gamma v(s')\Big]$

Figure 3.2: The policy iteration (left) and value iteration algorithm (right). Adapted from Sutton and Barto [12].

As can be seen from the boxes above, there are significant differences between the two algorithms. Firstly, the policy iteration algorithm involves two main parts. The policy evaluation and the policy improvement. First the value function for each state $s \in \mathcal{S}$ is approximated with accuracy θ. After this is done, we update the policy for each state in the following way: at each state select the state that maximizes the value function. This process is repeated until the policy converges. The value iteration algorithm simply finds the optimal value function for each state and then extracts the optimal policy by taking the argument that maximizes the optimal value function. This algorithm does not contain a policy improvement part in its algorithm, because once the optimal value function is known, one can simply take the argument that maximizes this function, to find the optimal policy. Secondly, note that the policy iteration algorithm does not take the max operation whereas the value iteration algorithm does. Although both algorithms converge to the optimal value function and thus to the optimal policy, it must be noted that the computations for both algorithms are often tedious in practice. A general drawback of DP is that it requires complete knowledge of the environment. For both algorithms we presented, we need to have knowledge about the transition probabilities from one state to another. In practice this is often not known. An additional problem is that of the growing complexity of the problem. There are (luckily) also RL algorithms that do not have to deal with this problem. In the next section, we will introduce algorithms that do not need to have complete knowledge of the environment. We begin by examining Monte Carlo methods.

3.3 Monte Carlo methods

In this section we will estimate optimal value functions and policies without presuming total knowledge of the environment. As we will see, with Monte Carlo (MC) methods we only require experience from visiting a state, performing a certain action, and obtaining its reward. Our MC algorithms will be similar as in DP. That is, we first estimate the value/action-value function and then use it to improve on the policy. It is important to note that in this section we will consider episodic tasks (tasks that have a natural ending at some time T). There are two main MC methods that we will consider. The first visit MC algorithm and the every-visit MC algorithm. The latter mentioned algorithm will be discussed later on in chapter 4. We will begin with the first visit MC algorithm:

Algorithm 1: First-visit MC

input : A policy π to be evaluated
output: approximated value function $v(s)$
initialization:
$v(s) \in \mathbb{R}$ for each $s \in \mathcal{S}$
Returns(s) $\leftarrow$ empty list $\forall s \in \mathcal{S}$;
for *each episode* **do**
  Generate an episode following: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$;
  for *each time step* $t = T-1, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$;
    if S_t *appears in* $S_0, S_1, \ldots, S_{t-1}$ **then**
      Append G to Returns(s)
      $V(S_t) \leftarrow$ average Returns(s)
    end
  end
end

As can be seen from the algorithm above, the first-visit MC approximates the value function $v(s)$ by the average of the returns following the first visit to state s. The following result shows that the first-visit MC algorithm actually converges to the true value function.

Lemma 3.6. *The first visit MC algorithm converges to $v_\pi(s)$ as the number of first visits s goes to infinity.*

Proof. The proof relies on the law of large numbers (see appendix). Our returns G_i are i.i.d. estimates of $v_\pi(s)$ with finite variance, $\mathbb{V}(G_i) < \infty$. Furthermore, we have that $\mathbb{E}_\pi[G_t|S_t = s] = v_\pi(s)$ holds by definition. The law of large numbers tells us that

$$\lim_{n\to\infty} \mathbb{P}\Big(\left| \frac{1}{n}(G_1 + \cdots + G_n) - v_\pi(s) \right| < \epsilon \Big) = 1 \ \forall \epsilon > 0$$

must hold. □

3.4 Temporal difference learning

In this section we will discuss other major RL algorithms. These algorithms are known as TD (Temporal difference) learning algorithms. These algorithms have characteristics both from DP and MC. As in the case with MC methods, TD learning does not need a full model of the environment, in order to learn. It can learn from experience. As in DP, TD methods use estimates in its updating rule. It does not need to wait an entire episode for an update (as is the case with MC methods). In this section, we will mainly focus on TD(0) and Q-learning. These methods both fall under TD learning. We will also shortly mention SARSA, but we will not discuss this at great length. First we will discuss the TD(0) algorithm.

3.4.1 TD(0) learning

The box below discusses the steps that are taken in the TD(0) algorithm:

Algorithm 2: Tabular TD(0) algorithm

input : A policy π to be evaluated, learning parameter $\alpha \in (0, 1]$
output: approximated value function $V(s)$
Initialization:
$V(s) \in \mathbb{R}$ for each $s \in \mathcal{S}$ arbitrarily, except $V(s_{terminal}) = 0$;
for *each episode* **do**
  Initialize S ;
  for *each time step of episode* **do**
    $A \leftarrow$ action given by π 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 $S' = s_{terminal}$;
  end
end

First note that this is indeed a bootstrapping algorithm since it uses an existing estimate in its updating rule. Also note that the big difference of TD(0) learning with respect to MC methods is that the value function is updated at each time step. The algorithm described above is the tabular form of TD(0) learning. As the name implies, we could in principle write all our updates for each state and reward in a table. Tabular TD(0) learning is often only used for small MDPs. If the state space becomes too large, then it becomes infeasible to use this algorithm. A more general setting (linear function approximation) is then required for the learning algorithm. The name temporal difference learning stems from the updates rule: $V(S_t) \leftarrow V(S_t)+\alpha\big(R_{t+1}+\gamma V(S_{t+1})-V(S_t)\big)$. The error, $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, is the temporal difference between the target $R_{t+1} + \gamma V(S_{t+1})$ and the estimate $V(S_t)$. The other TD algorithms that we will examine are SARSA and Q-learning. Both of these algorithms have a similar update rule as TD(0) learning. That is, the update rule involves a temporal difference between the target and the estimate.

3.4.2 SARSA and Q-learning

The SARSA algorithm uses action-value functions rather than value function and can thus be used to estimate action value functions. The algorithm is depicted below and is similar in structure to what we have already seen. Its update depends on $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$. Hence, the name SARSA algorithm.

Algorithm 3: SARSA algorithm

input : A policy π to be evaluated, learning parameter $\alpha \in (0, 1]$, parameter ϵ (optional)
output: approximated action-value function $Q(s, a)$
Initialization:
$Q(s, a)$ $\forall s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily, except $Q(s_{terminal}, \cdot) = 0$;
for *each episode* **do**
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy) ;
  for *each time step of episode* **do**
    Take action A, observe R' and S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big(R + \gamma Q(S', A') - Q(S, A)\big)$
    $S \leftarrow S', A \leftarrow A'$
    Until $S' = s_{terminal}$;
  end
end

The policy ϵ-greedy mentioned in the box above means that we take a random action with probability ϵ and follow the policy π with probability $1 - \epsilon$. In this thesis we will not use the SARSA algorithm, but we still want to mention it here because it illustrates that there are different types of TD algorithms. SARSA is called an on-policy algorithm, because it uses a fixed policy (ϵ-greedy) to choose actions. Q-learning, another TD algorithm, is an off-policy algorithm, because it does not have a fixed policy that maps from states to actions (A_{t+1} is not used in the update rule). The box on the next page describes how Q-learning is executed.

Algorithm 4: Q-learning

input : Algorithm parameters: A policy π to be evaluated, learning parameter $\alpha \in (0, 1]$, small parameter $\epsilon > 0$

output: approximated action-value function $Q(s, a)$

Initialization:

$Q(s, a)$ $\forall s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily, except $Q(s_{terminal}, \cdot) = 0$;

for *each episode* **do**

  Initialize S ;

  for *each time step of episode* **do**

    Choose A from S using policy π derived from Q (e.g., ϵ-greedy)

    Take action A, observe R, S'

    $Q(S, A) \leftarrow Q(S, A) + \alpha\big(R + \gamma \max_a Q(S', a) - Q(S, A)\big)$

    $S \leftarrow S'$

    Until $S' = s_{terminal}$;

  end

end

Now that we have defined the Q-learning algorithm we can discuss whether or not this algorithm indeed converges to the actual action value function $q(s, a)$. It turns out that this is indeed the case (e.g., see [11]).

Now that we have established that both MC-methods and TD methods converge to the actual value function we can ask ourselves the question: which of these algorithms converges faster? It turns out that this is quite a difficult question to answer. The answer to the question depends very much on the nature of the problem. In the next chapter we will explore different environments and examine which algorithm will converge faster.

4 Rate of convergence of RL algorithms

In this chapter we will examine the influence of different environments on the rate of convergence of TD and MC methods. There exists a range of results that indicate that TD methods are generally speaking more efficient (converge faster) than MC methods. This has mainly been verified by using simulations. In this chapter, we will discuss the rate of convergence of TD methods and MC methods in Markov reward processes (MRPs). The definitions of this process will be given in a moment. During this section we will analyze both deterministic and stochastic MRPs. In particular we will investigate the rate of convergence of Q-learning and every-visit MC for these MRPs.

4.1 Markov reward processes

As was already mentioned, we begin by discussing the rate of convergence of Q-learning and MC methods for Markov reward processes. Before we can start with the analysis we must introduce some definitions.

Definition 4.1 (Discrete-time Markov chain)**.** A discrete-time Markov chain is a sequence of random variables $X_1, X_2, \dots$ with the Markov property:

$$\mathbb{P}(X_{n+1} = x_{n+1} | X_1 = x_1, X_2 = x_2, \cdots, X_n = x_n) = \mathbb{P}(X_{n+1} = x_{n+1} | X_n = x_n).$$

That is, the probability of moving to the next state depends only on the current state.

Definition 4.2 (Absorbing Markov chain)**.** A Markov chain is called absorbing if:

- There exists at least one state $s \in \mathcal{S}$ such that: $\mathbb{P}(S_{t+1} = s | S_t = s) = 1$ and $\mathbb{P}(S_{t+1} = \tilde{s} | S_t = s) = 0$ for all $\tilde{s} \neq s$ (such a state s is called an absorbing state)
- For every state $s \in \mathcal{S}$ it is possible to go to an absorbing state (in finitely many steps).

Definition 4.3 (Markov reward process)**.** A Markov Reward process (MRP) is an MDP without any actions.

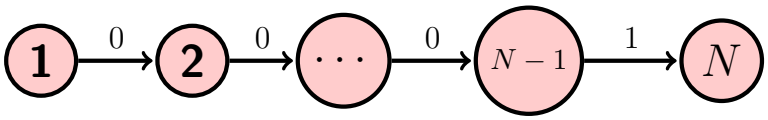
If we want to examine which algorithm will converge faster, then we also need a measure for the rate of convergence. We will use the following definition:

Definition 4.4 (Rate of convergence (action-value function))**.** We say that an algorithm $A : \Pi \to \mathcal{Q}$ converges with rate k if after k trials/episodes we have that: $|A(\pi) - q(s,a)| < \epsilon$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Here $\mathcal{Q}$ represents the space of action-value functions. $q(s,a)$ represents the true action-value function. $\epsilon > 0$ is an arbitrarily small number. A, represents an algorithm that follows a policy $\pi \in \Pi$ and returns an action-value function.

In other words if an algorithm A has a convergence rate of k, then the algorithm needs at least k samples/trials to approximate the true action-value function $q(s,a)$ with arbitrarily small accuracy. If the output of an algorithm is not an action-value function, but a value function for instance, then we simply replace the $q(s,a)$ by $v(s)$. Note, however, that in the case of a MRP there are no actions to be taken. Therefore, we have that $q(s,a) = v(s)$ for all $s \in \mathcal{S}$.

4.1.1 Q-learning vs MC methods in deterministic MRPs

The first example that we will consider is an undiscounted ($\gamma = 1$) Markov reward process. Moreover, the Markov chain is an absorbing one. In our example the agent starts at state S_1 and moves through state S_2 w.p. 1. The agent receives zero reward. This process is then repeated until the agent reaches state S_{N-1}. If the agent is in this state it moves to state S_N (terminal absorbing state) and receives a reward of 1. Thus we will analyze the following MRP.



Now that we have described the setting of our first MRP we can prove the following lemma.

Lemma 4.5. *The rate of convergence of Q-learning in an undiscounted MRP with an absorbing Markov chain that consists of N states and 1 absorbing state (N) and with $r(s,a) = \mathbb{1}_{N-1}$, scales exponentially with the number of states N.*

Proof. In this specific setting, Q-learning coincides with the TD(0) method (since there is only one action that can be taken). If we examine the algorithm of Q-learning (or TD(0) learning), then we see that the state prior to the terminal state is updated first. The next trial both $N-1$ and $N-2$ are updated. This process continues until the first state. The first state is updated last and this state needs the most trials k of any state to converge to the value of 1. Thus, we only have to consider the number of trials k it takes for $Q(1)$ to lie in an ϵ-neighborhood of $q(s) = 1$. This is because if $Q(1)$ lies in an ϵ-neighborhood of 1, then all other states also lie in an ϵ-neighborhood of 1. Note that we write $Q(s)$ instead of $Q(s,a)$, because we only have one action to take. We initialize

our Q-values all as zero for all states. Our Q-model reduces to:

$$\begin{aligned}
&Q_0(s) = 0 \quad \forall s \in \mathcal{S} \\
&Q_{t+1}(s_t) = Q_t(s_t) + \alpha_t(s_t)(r(s_t) + Q_t(s_{t+1}) - Q_t(s_t)) \ \text{ for } \ t \geq 0 \\
&Q_t(N) = 0 \ \text{ for } \ t \geq 0
\end{aligned}$$

Here we define our learning parameter $\alpha_t(s)$ as the reciprocal of the number of times state s has been visited up to time t. We can thus write $\alpha = \frac{1}{k}$ for each state s during the k^{th} trial. We can write our Q updating rule as follows:

$$\begin{aligned}
Q_k(s) &= Q_{k-1}(s) + \frac{1}{k}\Big(Q_{k-1}(s+1) - Q_{k-1}(s)\Big) \\
&= (1 - \frac{1}{k})Q_{k-1}(s) + \frac{1}{k}Q_{k-1}(s+1) \ \text{ for } \ s < N-1.
\end{aligned}$$

Before we continue with the proof, we must show that we can write the Q-value of state s at trial k as:

$$Q_k(s) = \frac{1}{k}\sum_{i=0}^{k-1} Q_i(s+1) \ \text{ for } \ s < N-1. \tag{4.1}$$

We will prove this by induction. First we see that this indeed holds for $k = 1$:
$Q_1(s) = 1 \cdot Q_0(s+1)$ for $s < N-1$ (see definition of Q_k above). Next we take

$$Q_{k-1}(s) = \frac{1}{k-1}\sum_{i=0}^{k-2} Q_i(s+1)$$

as induction hypothesis (IH). We can rewrite Q_k as follows:

$$\begin{aligned}
Q_k(s) &= (1 - \frac{1}{k})Q_{k-1}(s) + \frac{1}{k}Q_{k-1}(s+1) \\
&= \frac{1}{k}\Big(Q_{k-1}(s+1) + (k-1)Q_{k-1}(s)\Big) \\
&=^{(IH)} \frac{1}{k}\Big(Q_{k-1}(s+1) + \sum_{i=0}^{k-2} Q_i(s+1)\Big) \\
&= \frac{1}{k}\sum_{i=0}^{k-1} Q_i(s+1).
\end{aligned}$$

For $N-1$ and N we have that $Q(N-1) = 1$ and $Q(N) = 0$ for all $k \geq 1$. Next we will denote the average of the Q-values of state $s = 1$ till state $s = N-2$ at trial k with S_k. Thus, we write

$$S_k = \frac{Q_k(1) + \cdots + Q_k(N-2)}{N-2}.$$

We claim that we can rewrite this expression as:

$$S_k = S_{k-1} + \frac{Q_{k-1}(N-1) - Q_{k-1}(1)}{k(N-2)}.$$

To see this we write:

$$S_{k-1} + \frac{Q_{k-1}(N-1) - Q_{k-1}(1)}{k(N-2)} = \frac{Q_{k-1}(1) + \cdots + Q_{k-1}(N-2)}{N-2} + \frac{Q_{k-1}(N-1) - Q_{k-1}(1)}{k(N-2)}$$
$$= \frac{(k-1)Q_{k-1}(1) + \cdots + kQ_{k-1}(N-2)}{k(N-2)} + \frac{Q_{k-1}(N-1)}{k(N-2)}$$
$$= \frac{(k-1)Q_{k-1}(1) + Q_{k-1}(2)}{k(N-2)} + \frac{(k-1)Q_{k-1}(2) + Q_{k-1}(3)}{k(N-2)} + \cdots$$
$$+ \frac{(k-1)Q_{k-1}(N-2) + Q_{k-1}(N-1)}{k(N-2)} = \sum_{i=1}^{N-2} \frac{Q_k(i)}{N-2} = S_k.$$

Note that $Q_k(1) < S_k$ holds. Moreover, we obviously have that $S_0 = 0$. Now we can upper bound $Q_k(1)$:

$$Q_k(1) < S_k \leq S_{k-1} + \frac{1}{k(N-2)} \leq S_{k-2} + \frac{1}{(k-1)(N-2)} + \frac{1}{k(N-2)} \leq \cdots \leq S_0 + \frac{1}{N-2} \sum_{i=1}^{k} \frac{1}{i}.$$

We get

$$Q_k(1) < \frac{1}{N-2} \sum_{i=1}^{k} \frac{1}{i} \leq \frac{1 + \log k}{N-2}$$

as upper bound for $Q_k(1)$. To determine the rate of convergence (see definition) we must examine:

$$|q(1) - Q_k(1)| = |1 - Q_k(1)| = 1 - Q_k(1).$$

This quantity is smaller than $\epsilon > 0$ iff:

$$1 - \epsilon < Q_k(1) < \frac{1 + \log k}{N-2} \iff k > e^{(1-\epsilon)(N-2)-1}.$$

□

Thus, we need more than

$$e^{(1-\epsilon)(N-2)-1}$$

trials to get an error less than epsilon. Now that we know the rate of convergence of Q-learning we can discuss the rate of convergence of MC-methods. We will discuss the every-visit MC method.

Algorithm 5: Every-visit MC

input : A policy π to be evaluated
output: approximated value function $v(s)$
initialization:
$v(s) \in \mathbb{R}$ for each $s \in \mathcal{S}$
Returns(s) $\leftarrow$ empty list $\forall s \in \mathcal{S}$;
for *each episode* **do**
  Generate an episode following: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$;
  for *each time step* $t = T-1, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$;
    Append G to Returns(s)
    $V(S_t) \leftarrow$ average Returns(s)
  end
end

This algorithm only differs from the first-visit MC method, that we have seen in the previous chapter, in the sense that there is no check whether a state has been visited before during the episode. It is now easily seen that only 1 trial is required for every-visit MC to learn the true action-value function $q(s, a)$. This is because the total reward for each episode equals 1. If $k = 1$, then we have that every-visit MC algorithm A, satisfies $|A(\pi) - q(s)| < \epsilon$ for all $s \in \mathcal{S}$ after one trial. Hence, the every-visit MC algorithm (and also the first-visit MC algorithm) converges to the real action-value function after only 1 trial. We conclude that in this setting, MC algorithms converge faster than Q-learning. This is summarized in the figure on the next page. In this figure we plotted the number of trials (k) required to get an error less than ϵ, against the number of states in the chain for both MC-methods and Q-learning.

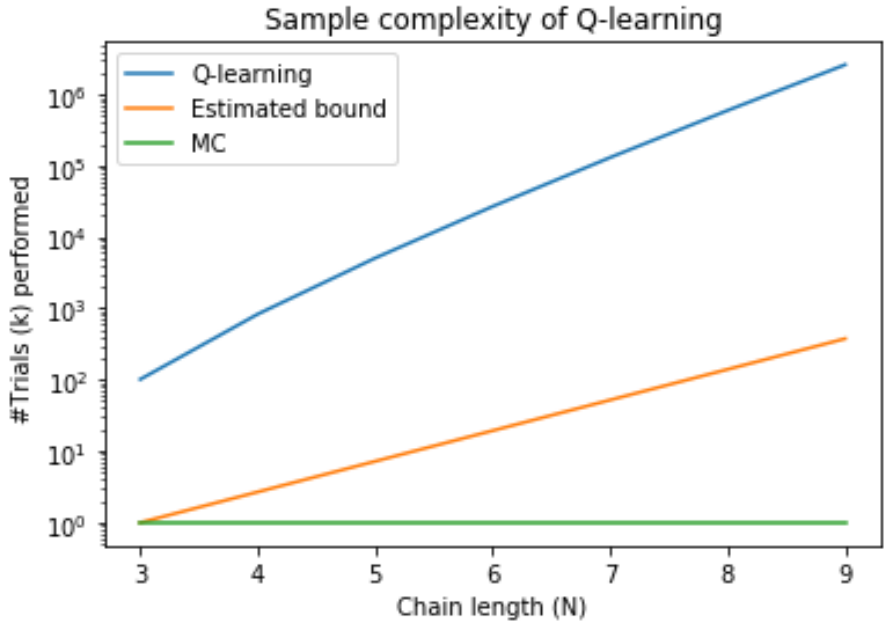


Figure 4.1: The number of trials (k) required to get an error less than $\epsilon = 0.01$.

The blue line (Q-learning) corresponds to the number of trials (k) required to get an error less than ϵ by following the Q-learning algorithm. The red line is given by $k = e^{(1-\epsilon)(N-2)-1}$ and denotes the lower bound for k in Q-learning. The green line represents the trials required to get an error less than ϵ. A few remarks are in order. First, note the y-axis has a logarithmic scale. Second, observe that the bound we computed earlier on is not very sharp (considering the y-axis is logarithmic). We also see that, for the MC algorithm, the number of trials required to get an error less than ϵ does not depend on the chain length N. We conclude that in this deterministic MRP, MC-methods are more efficient than TD methods (i.e., MC-methods converge faster). In the next subsection we will examine a stochastic MRP and we will ask ourself the same question: which algorithm (MC or Q-learning) converges faster?

4.1.2 Q-learning vs MC methods in stochastic MRPs

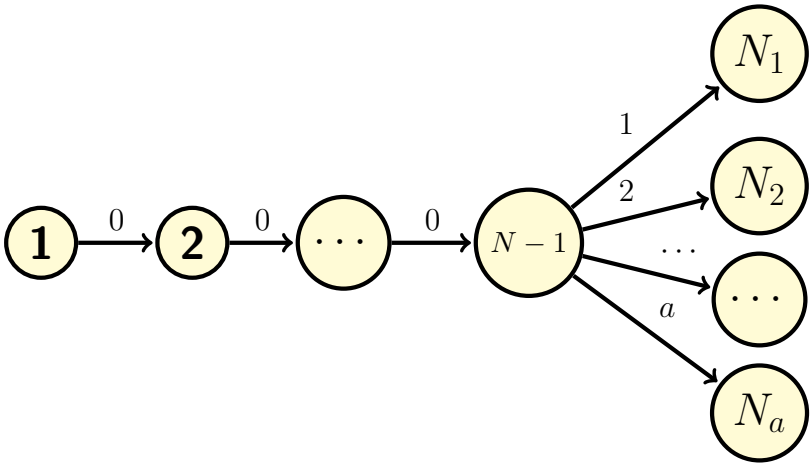
In this subsection, we will address the same question as before, but now we will slightly alter the MRP we discussed earlier. As the name of this subsection suggests, we now allow the rewards to be random variables instead of fixed constants. This makes the situation a bit more complex, because our action-value function depends on the rewards it receives during an episode. Thus, our action-value functions themselves become random variables. Therefore, we require a new notion of the rate of convergence of an algorithm. We introduce the following definition.

Definition 4.6 (Rate of convergence (action-value function))**.** We say that an algorithm $A : \Pi \to \mathcal{Q}$ converges with rate k if after k trials/episodes we have that:

$$\mathbb{P}\big(|Q_k(s) - q(s)| \geq \epsilon \big) \leq \delta$$

for all $s \in \mathcal{S}$. As before, $\mathcal{Q}$ represents the space of action-value functions. $q(s)$ represents the true action-value function and $Q_k(s)$ represents the estimated action value function after k trials. $\epsilon > 0$ is an arbitrarily small number and $\delta \in [0, 1]$.

Now that we introduced the definition of the rate of convergence that is suited for stochastic MRPs, we can extend the previous deterministic MRP with a stochastic component. The extension is depicted below.



The episode begins at state $S_0 = 1$ and ends when one of the terminal states (N_1, … , N_a) is reached. A reward of 0 is assigned for the transition from state i to state $i + 1$ with $1 \leq i \leq N - 2$. The reward that is received from the transition from state $N - 1$ to state N_j with $j \in \{1, 2, \ldots, a - 1, a\}$ is a random variable that follows the discrete uniform distribution. In particular, $R_i \sim \mathcal{U}\{1, a\}$ at each trial i for the transition from state $N - 1$ to state N. We thus have:

$$\mathbb{P}(S_{t+1} = i + 1 | S_t = i) = 1 \text{ for all } 1 \leq i \leq N - 2$$
$$\mathbb{P}(S_T = N_j | S_{T-1} = N - 1) = \frac{1}{a} \text{ for all } j \in \{1, 2, \ldots, a - 1, a\}$$

The goal for us is to explore whether Q-learning or MC-methods is best suited to learn the action-value function. This has become somewhat more difficult, because of the stochastic nature of the problem, but we can still prove that Q-learning will not perform better than MC-methods. To prove this, we begin by determining the rate of convergence for MC methods. This is done first.

Lemma 4.7. *The rate of convergence (def 4.6) of the MC algorithm in an undiscounted stochastic MRP with the following Markov chain*

$$\begin{aligned}
&\mathbb{P}(S_{t+1} = i+1 | S_t = i) = 1 \ \textit{for all} \ 1 \leq i \leq N-2 \\
&\mathbb{P}(S_T = N_j | S_{T-1} = N-1) = \frac{1}{a} \ \textit{for all} \ j \in \{1, 2, \ldots, a-1, a\} \\
&r(s) = 0 \ \textit{for all} \ 1 \leq s \leq N-2 \\
&r(N-1) \sim \mathcal{U}\{1, a\}
\end{aligned}$$

scales algebraically with $(a-1)$.

Proof. To prove this statement we first need to determine the rate of convergence, k_{MC}, of the MC algorithm. Here $k_{MC} \in \mathbb{N}$ denotes the number of trials needed such that

$$\mathbb{P}\Big(|Q_k(s) - q(s)| \geq \epsilon \Big) \leq \delta$$

holds. By following the MC algorithm (either first-visit or every-visit) we get

$$Q_k(s) = \frac{1}{k} \sum_{i=1}^{k} R_i = \bar{R} \ \text{ for all } \ 1 \leq s \leq N-1$$

as estimate for the action-value function after k trials. Here $R_i \sim \mathcal{U}\{1, a\}$ represent the reward observed at trial i. Note that the real action value function can be computed easily:

$$q(s) = \mathbb{E}[R_i] = \frac{1}{a} \sum_{j=1}^{a} j.$$

We also have that

$$\mathbb{E}[\bar{R}] = \mathbb{E}\left[\frac{1}{k} \sum_{i=1}^{k} R_i \right] = \frac{1}{k} \sum_{i=1}^{k} \mathbb{E}[R_i] = \mathbb{E}[R_i] = q(s)$$

holds. We are interested in an upper bound of the following probability

$$\mathbb{P}\Big(|Q_k(s) - q(s)| \geq \epsilon \Big).$$

We can rewrite this probability as

$$\mathbb{P}\Big(\big|\bar{R} - \mathbb{E}[\bar{R}]\big| \geq \epsilon \Big).$$

Now that we have written it in this form we can invoke Hoeffding's inequality (see appendix I) to get the following upper bound:

$$\mathbb{P}\Big(\big|\bar{R} - \mathbb{E}[\bar{R}]\big| \geq \epsilon \Big) \leq 2 \exp\Big(- \frac{2k\epsilon^2}{(a-1)^2} \Big) \leq \delta.$$

Hence, if we take

$$k_{MC} \geq \left\lceil \frac{(a-1)^2}{2\epsilon^2} \log \frac{2}{\delta} \right\rceil$$

then we have with probability less or equal to δ that the error, $|Q_k(s) - q(s)|$, equals or exceeds ϵ. Hence, the rate of converges scales algebraically with $(a-1)$.

□

Next we will draw our attention to Q-learning. We will prove the following.

Lemma 4.8. *In a undiscounted stochastic MRP with the following Markov chain*

$$\begin{aligned}
&\mathbb{P}(S_{t+1} = i+1 | S_t = i) = 1 \text{ for all } 1 \leq i \leq N-2 \\
&\mathbb{P}(S_T = N_j | S_{T-1} = N-1) = \frac{1}{a} \text{ for all } j \in \{1, 2, \ldots, a-1, a\} \\
&r(s) = 0 \text{ for all } 1 \leq s \leq N-2 \\
&r(N-1) \sim \mathcal{U}\{1, a\}
\end{aligned}$$

$Q_k(N-1)$ *requires*

$$k \geq \left\lceil \frac{(a-1)^2}{2\epsilon^2} \log \frac{2}{\delta} \right\rceil$$

trials so that

$$\mathbb{P}\Big(|Q_k(N-1) - q(N-1)| \geq \epsilon \Big) \leq \delta \tag{4.2}$$

holds for the Q-learning algorithm.

Proof. To prove the lemma we need to study how $Q_k(N-1)$ behaves. We already know that

$$q(s) = \mathbb{E}[R_i] = \frac{1}{a} \sum_{j=1}^{a} j$$

holds. Now by following the definition of the Q-learning algorithm we get

$$Q_k(N-1) = Q_{k-1}(N-1) + \frac{1}{k}\Big(R_k - Q_{k-1}(N-1) \Big) \tag{4.3}$$

as value for the approximated action-value function $Q_k(N-1)$ at trial k. We claim that

$$Q_k(N-1) = \frac{1}{k} \sum_{i=1}^{k} R_i \tag{4.4}$$

holds for all $k \geq 1$. We prove this by induction. First we check the base case $k = 1$:

$$Q_1(N-1) = R_1.$$

If we plug in $k = 1$ in (4.3), then we indeed see that $Q_1(N-1) = R_1$. This is because $Q_0(s)$ for all $s \in \mathcal{S}$ are initialized as 0. Next we take (4.4) as induction hypothesis and we need to show that

$$Q_{k+1}(N-1) = \frac{1}{k+1} \sum_{i=1}^{k+1} R_i$$

holds. We can show this by inserting (4.4) into (4.3):

$$\begin{aligned} Q_{k+1}(N-1) &= Q_k(N-1) + \frac{1}{k+1}\Big(R_{k+1} - Q_k(N-1)\Big) \\ &= \frac{(k+1)(R_1 + \ldots + R_k)}{k(k+1)} + \frac{kR_{k+1}}{k(k+1)} - \frac{R_1 + \ldots + R_k}{k(k+1)} \\ &= \frac{1}{k+1}\sum_{i=1}^{k+1} R_i. \end{aligned}$$

So now we have shown that (4.4) holds. Note that in Q-learning the approximated action-value function $Q_k(N-1)$ behaves the same as all the other MC approximated action-value functions. Therefore, we have that (4.2) holds for

$$k \geq \left\lceil \frac{(a-1)^2}{2\epsilon^2} \log \frac{2}{\delta} \right\rceil$$

as before. □

Now that we have proved Lemma 4.7 and Lemma 4.8 we can easily see that the following result follows.

Corollary 4.8.1. *The rate of convergence (def 4.6) of Q-learning in a undiscounted stochastic MRP with the following Markov chain*

$$\begin{aligned} &\mathbb{P}(S_{t+1} = i+1 | S_t = i) = 1 \ \textit{for all} \ 1 \leq i \leq N-2 \\ &\mathbb{P}(S_T = N_j | S_{T-1} = N-1) = \frac{1}{a} \ \textit{for all} \ j \in \{1, 2, \ldots, a-1, a\} \\ &r(s) = 0 \ \textit{for all} \ 1 \leq s \leq N-2 \\ &r(N-1) \sim \mathcal{U}\{1, a\} \end{aligned}$$

is not strictly smaller than the rate of convergence of the MC algorithm.

Proof. Apply Lemma 4.7 and Lemma 4.8 □

We conclude that in a stochastic MRP it is not necessarily the case that Q-learning performs better than MC methods. In the next subsection, we will discuss our last MRP. The MRP that we will consider is a random walk with rewards.

4.1.3 Random walk

In this section, we will explore the efficiency of Q-learning and MC methods in a random walk. Previously, we have seen that in our deterministic MRP, MC-methods only require one trial to learn the real underlying (action-)value function. When we added uncertainty for the rewards (discrete uniformly distributed rewards) we saw that the trials k required for convergence, scaled algebraically with the number of different rewards (see previous section). We also introduced uncertainty for the successive state of state $N-1$. In this section we will allow uncertainty for each state. That is, each state (except for terminal states) can transition to two different states. We begin by introducing some definitions.

Definition 4.9 (Random walk)**.** A random walk is a stochastic process $\{W_n\}$ with $W_0 = 0$ and $W_n = \sum_{i=1}^{n} X_i$, where the X_i are i.i.d random variables.

Definition 4.10 (Simple symmetric random walk)**.** We call a random walk simple and symmetric if $X_i \in \{-1, 1\}$ with $\mathbb{P}(X_i = 1) = \frac{1}{2}$ and $\mathbb{P}(X_i = -1) = \frac{1}{2}$

We will study a simple symmetric random walk where $W_0 = 3$ and where

$$\mathbb{P}(S_{t+1} = 6 | S_t = 6) = 1 \text{ for all } t$$
$$\mathbb{P}(S_{t+1} = 0 | S_t = 0) = 1 \text{ for all } t$$

In addition, we give a reward of 1 for the transition of state 5 to state 6. The MRP model we will study is depicted below.

Figure 4.2: A simple symmetric random walk.

As before we must know the true value function $v(s)$ for each state s before we can determine how efficient an RL algorithm is. It is clear that both $v(0)$ and $v(6)$ are zero. We can determine the value function of the other states as follow:

$$\begin{aligned} v(i) &= \mathbb{P}(S_T = 6 \text{ for some } T | s_t = i) \\ &= \mathbb{P}(S_T = 6 \text{ for some } T | s_t = i, s_{t+1} = i+1)\mathbb{P}(s_{t+1} = i+1) \\ &+ \mathbb{P}(S_T = 6 \text{ for some } T | s_t = i, s_{t+1} = i-1)\mathbb{P}(s_{t+1} = i-1) \\ &= \frac{1}{2}v(i+1) + \frac{1}{2}v(i-1) \quad \text{for all } i \in \{1, \ldots, 5\} \end{aligned}$$

We can solve this homogeneous difference equation by determining the zeroes of the characteristic polynomial:

$$\frac{1}{2}x^2 - x + \frac{1}{2} = 0 \implies x = 1$$

Thus we get

$$v(i) = A \cdot 1^i + B \cdot i \cdot 1^i$$

as expression for the value function. Now we can invoke the boundary conditions, $v(0) = v(6) = 0$ to determine the constants A and B. This yields

$$v(i) = \frac{i}{6} \quad \text{for } i \in \{1, \ldots, 5\}.$$

In the previous section we were able to derive bounds on the rate of convergence, for both the MC methods as Q-learning. In this situation it becomes too difficult to derive a bound for the rate of convergence for both learning algorithms. Therefore, we will use a

different measure to assess the performance of both learning algorithms. We will compute the root mean squared error (RMSE), between the approximated value function $\tilde{V}$ and the actual value function v, averaged over the five states. Thus we will use

$$\textbf{RMSE} = \Big(\frac{1}{5}\sum_{i=1}^{5}(\tilde{V}(i) - v(i))^2\Big)^{\frac{1}{2}}$$

as performance measure. We will use the first-visit variant of MC learning in our analysis (see first box in chapter 3). Recall, that for the first-visit MC learning algorithm, we have to wait for the end of the walk before we can update our value function. For each state s that was visited during the walk, we will update the value function of this state as follows:

$$\tilde{V}(s) \leftarrow \tilde{V}(s) + \alpha(G - \tilde{V}(s)). \tag{4.5}$$

This is the simple update rule for the Monte Carlo method. Here G denotes the return of the walk, after the first time state s has been visited. The intuition behind the above mentioned update rule is as follow. First, recall the definition of the estimated value function after k trials:

$$\tilde{V}_k(s) = \frac{1}{k}\sum_{i=1}^{k} G_i.$$

This expression can be rewritten as:

$$\begin{aligned}\tilde{V}_k(s) = \frac{1}{k}\sum_{i=1}^{k} G_i &= \frac{1}{k}\Big(G_k + \sum_{i=1}^{k-1} G_i\Big) = \frac{1}{k}\Big(G_k + (k-1)\tilde{V}_{k-1}(s)\Big) \\ &= \tilde{V}_{k-1}(s) + \frac{1}{k}\Big(G_k + \tilde{V}_{k-1}(s)\Big).\end{aligned}$$

If we now replace our factor, $\frac{1}{k}$, by a constant (learning) parameter α, we get (4.5) as update rule. We will initialize our value function $\tilde{V}(s) = 0.5$ for all $s \in \{1, \ldots, 5\}$. For Q-learning, we update our value function after each time step. We do not have to wait for the end of the walk, to update our value function. Instead, as was already mentioned in the previous chapter, we update our value function after each time step according to:

$$\tilde{V}(s_t) \leftarrow \tilde{V}(s_t) + \alpha(R_t + \gamma\tilde{V}(s_{t+1}) - \tilde{V}(s_t)).$$

We can use these learning algorithms, to approximate the value function and compute the corresponding RMSE. If we set $\alpha = 0.012$ and $\gamma = 0.9$, then we obtain the plot depicted on the next page.

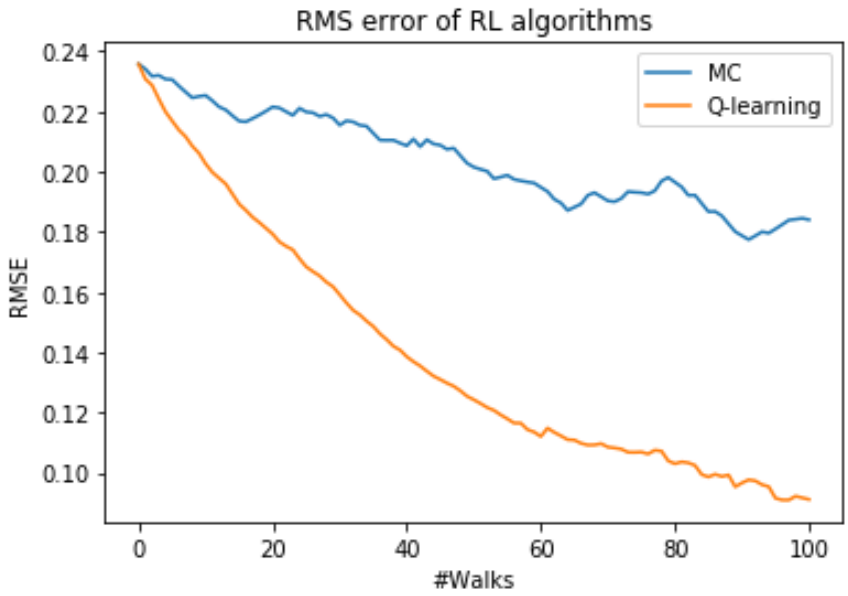


Figure 4.3: Plot of the RMSE of the first-visit MC method and Q-learning against the number of walks performed.

From the plot above we clearly see that the RMSE decreases faster under Q-learning, than under the first-visit MC method. The first-visit MC method results in a slower decrease of the RMSE, compared to Q-learning. We must note that if we repeat the procedure as described above, then our final plot will differ from the plot above. This is due to the stochastic nature of the problem. However, the plot will be qualitatively the same, as the plot above. In the preceding subsections we were able to derive bounds on the rate of convergence analytically. In these cases, we can therefore be certain which algorithm converges faster. For this particular MRP on the other hand, it is more difficult to derive a bound for the rate of convergence analytically. Hence, we are therefore resorted to simulations. From these simulations, we conclude that Q-learning is better suited for learning the true value function than the first-visit MC method in this symmetric random walk. So far we have focused on deterministic and stochastic MRPs. We explicitly chose MRPs as framework to study, because this allows, in some cases, for an analytical derivation of a bound for the rate of convergence. For MDPs and multi-armed bandit problems (this topic will be introduced in the next chapter) this is typically not the case. These types of problems become in some sense too complex for an analytical derivation of the rate of convergence. In these situations one has to resort to numerical methods. This is what we will do in the upcoming chapters.

5 RL algorithms and decision-making/learning

In this chapter, we will analyze the learning behavior of an agent (in our case that of a human). We will analyze the decisions the agent makes in a learning task that is known as the Iowa gambling task (IGT). We begin this chapter by introducing the IGT. After we have introduced the IGT we will introduce a mathematical framework, known as the multi-armed bandit (MAB) framework to analyze the decisions made by the learner. In the subsequent section, we will perform the so-called upper confidence bound algorithm (UCB) to the same learning task. Next we will model this task as a one-state MDP. Here we will perform the same RL algorithm as we did in the previous chapter. Furthermore, we will use a Bayesian framework for the IGT. This framework requires additional prior information that the other algorithms do not possess. Finally, in the last section we will compare all learning methods to each other and then we will conclude which method is most suited for this learning task. We begin this chapter by explaining what the IGT precisely entails.

5.1 The Iowa gambling task

The IGT is a learning task that has been quite popular in both psychology and in neuroscience. This task is designed to test the performance of both healthy and unhealthy individuals in decision-making. During this task, participants have the option to choose a card from four (virtual) decks. These are called deck A, B, C, and D. The participants are told that each deck (A, B, C, and D) contains cards that will either yield a reward or loss of virtual money. The participants are told that the objective of the task is to gain as much money as possible. After selecting a card, the participants are informed about their gain or loss of picking that card. What is unknown to the participants is that some decks are advantageous in the long run, whereas other decks are not. Thus the participants have to learn to choose cards from the advantageous decks. The table below summarizes the win/loss of picking a card from deck A, B, C, or D.

Deck	Win (per card)	loss (per card)	Expected value
A	100	X_A	-25
B	100	X_B	-25
C	50	X_C	25
D	50	X_D	25

The loss (per card) are random variables that follow these distributions:

$$X_A = \begin{cases} -150 & \text{w.p. } p = 1/10 \\ -200 & \text{w.p. } p = 1/10 \\ -250 & \text{w.p. } p = 1/10 \\ -300 & \text{w.p. } p = 1/10 \\ -350 & \text{w.p. } p = 1/10 \\ 0 & \text{w.p. } p = 1/2 \end{cases} \quad X_B = \begin{cases} -1250 & \text{w.p. } p = 1/10 \\ 0 & \text{w.p. } p = 9/10 \end{cases}$$

$$X_C = \begin{cases} -25 & \text{w.p. } p = 1/10 \\ -75 & \text{w.p. } p = 1/10 \\ -50 & \text{w.p. } p = 3/10 \\ 0 & \text{w.p. } p = 1/2 \end{cases} \quad X_D = \begin{cases} -250 & \text{w.p. } p = 1/10 \\ 0 & \text{w.p. } p = 9/10 \end{cases}$$

The expected value of picking a card from deck A, B, C, or D can now easily be computed. These values are presented in the table on the previous page. The net gain/loss of choosing a card from deck A, B, C, or D is a random variable that is distributed as $100 + X_A$, $100 + X_B$, $50 + X_C$, and $50 + X_D$, respectively. We conclude that decks C and D are 'advantageous', whereas decks A and B are considered 'disadvantageous'. Hence, the task of the learner is to learn to choose deck C or D. We must also note that a trial stops when 95 cards have been chosen. (This termination condition is unknown to the participants). Now that we have introduced the IGT we can formalize this task mathematically.

5.2 Multi-armed bandit problems

The multi-armed bandit problems (MABs) refer to a set of problems, in which an actor can perform several actions (from a set of action) in a sequential decision problem. At each time step, the actor must take an action which results in an observable pay-off. The goal of the agent is to maximize the long-term pay-off. MAB problems provide limited information to the agent regarding the pay-off. The agent just knows that the pay-off received upon selecting an action, follows an unknown distribution that might change over time. Because of the nature of MAB problems, we are confronted with the fundamental trade-off between exploration and exploitation. The agent must find a balance between the exploitation of an action that yielded a high pay-off, and the exploration of actions that could even give higher pay-offs in the future. The MABs can be formalized in three different ways. They can be distinguished from each other on the basis of the reward process. The reward process of an MAB problem can either be stochastic, adversarial or Markovian. We will only focus on the former, because our task the IGT can be modeled as a stochastic MAB problem. There are several bandit algorithms that can be performed to maximize the long-term pay-off of the learning task. We will focus on the upper confidence bound (UCB) algorithm and on Thompson sampling. First we will introduce some mathematical notation.

A multi-armed bandit problem consists of a set of unknown distributions

$$N = \{\nu_1, \ldots, \nu_k\}$$

where each distribution ν_i is associated with the reward delivered upon selecting selecting arm/action $i = 1, \ldots, K$. We denote the mean of ν_i (= mean reward of arm i) by μ_i. We will study the decisions of the agent during the IGT (which is a 4-armed bandit problem). That is, at each time step $t = 1, 2, \ldots$, the agent selects an arm $I_t \in \{1, \ldots, K\}$ and receives the associated reward $X_{I_t,t}$. Next we need a measure for the performance of the learner. The measure we will use is known as the regret:

Definition 5.1 (Regret)**.** The regret denoted by R_T is given by:

$$R_T = \max_{i=1,\ldots,K} \sum_{t=1}^{T} X_{i,t} - \sum_{t=1}^{T} X_{I_t,t}.$$

The regret after T rounds is thus the difference between the cumulative reward obtained by choosing the best arm T times, and the cumulative reward obtained by selecting arms according to some strategy S. Note that both rewards $X_{i,t}$ and $X_{I_t,t}$ are stochastic variables. We therefore introduce pseudo-regret:

Definition 5.2 (Pseudo-regret)**.** The pseudo-regret denoted by $\bar{R}_T$ is given by:

$$\bar{R}_T = T\mu^* - \mathbb{E}_{I_1,\ldots,I_T}\Big\{\sum_{t=1}^{T} X_{I_t,t}\Big\}$$

where

$$\mu^* = \max_{i=1,\ldots,K} \mu_i = \max_{i=1,\ldots,K} \mathbb{E}_{X\sim\nu_k}[X_i].$$

The box below summarizes the multi-armed bandit problem we have discussed so far.

Stochastic Bandit setting

Environment: distributions $(\nu_1, \ldots \nu_K)$ of arm rewards.
Protocol: For $t = 1, 2, \ldots$

- Learner picks arm I_t
- Learner observes and receives *reward* $X_{I_t,t} \sim \nu_{I_t}$

Objective: Minimize pseudo-Regret w.r.t. best expert after T rounds:

$$\bar{R}_T = T\mu^* - \mathbb{E}_{I_1,\ldots,I_T}\Big\{\sum_{t=1}^{T} X_{I_t,t}\Big\}.$$

Now that we have introduced the regret and pseudo-regret we can visualize how our 15 healthy participants performed on the IGT. This will be done in the next subsection. We will expand on the stochastic bandit setting and on bandit algorithms in Section 5.3.

5.2.1 Performance of healthy human participants

We have data of 15 healthy participants that performed the IGT. The data was originally obtained by Fridberg et al. This study also provides data on the performance on the IGT of chronic cannabis users. This additional data will be shown later on in this section. We will first turn our attention to the performance of the 15 healthy participants. These participants served as a control group in the original study of Fridberg et al. The group had an average age of 29 years and had an estimated average IQ of 110. Other interesting characteristics can be found in the study of Fridberg et al. The figure below displays the actions taken over time by four random participants.

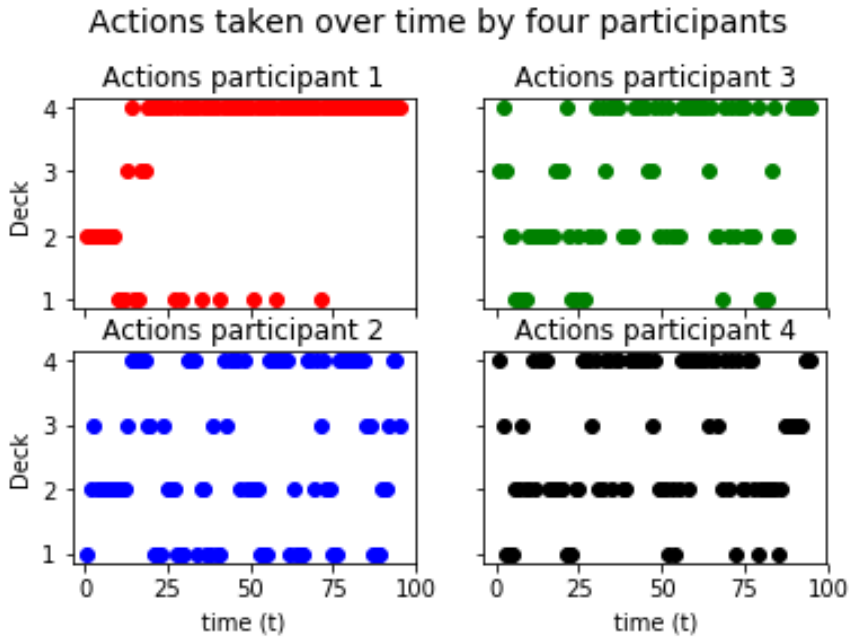


Figure 5.1: Time plotted against the action taken by participants.

In the plot above we identified decks A, B, C, and D with $1, 2, 3$, and 4, respectively. The behavior of participant 1 is striking. This participant learns the advantageous deck rather quickly and exploits its knowledge on the reward structure. Other participants seem to have some difficulty with finding the advantageous decks (at least to some degree). From the plot above we can already see that there is variation of the performance between participants on the IGT. Therefore, we will look at the average performance of the participants. We will compute the pseudo-regret of the participants for 5 blocks. The first block starts at $t = 1$ and ends at $t = 20$. The second block starts at $t = 21$ and ends at $t = 30$ and so on. The fifth block starts at $t = 80$ and ends at $T = 95$. We will compute the pseudo-regret at each time-block for the participants. This yields the plot on the next page.

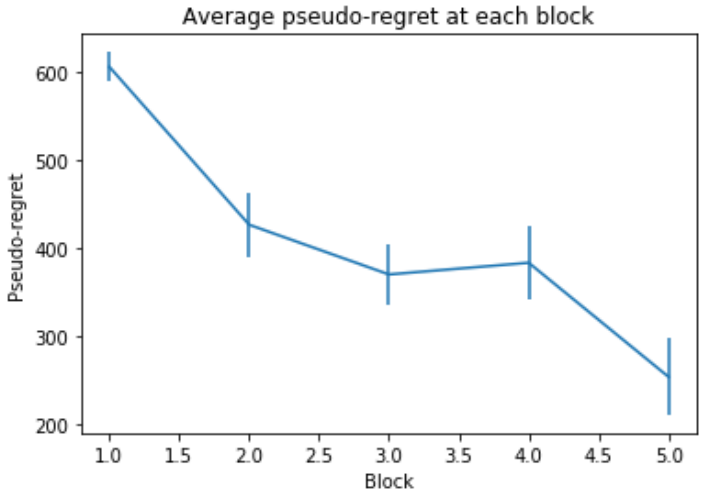


Figure 5.2: Plot of average pseudo-regret ± SEM at each block.

We see that indeed there exists a downward trend of the average pseudo-regret over time. This means that the participants **on average** learn to prefer the advantageous decks (C and D) over the disadvantageous decks (A and B). We could also use a different measure for the performance of the participants. Namely, the proportion of advantageous actions chosen. This is also a measure that Friedberg et al. [13] have chosen. Their results are depicted below. Here we also included the group that consists of chronic cannabis users. In our analysis we are not interested in their performance, nonetheless it is interesting to see how the performance of 'healthy' participants compares to a group that is not.

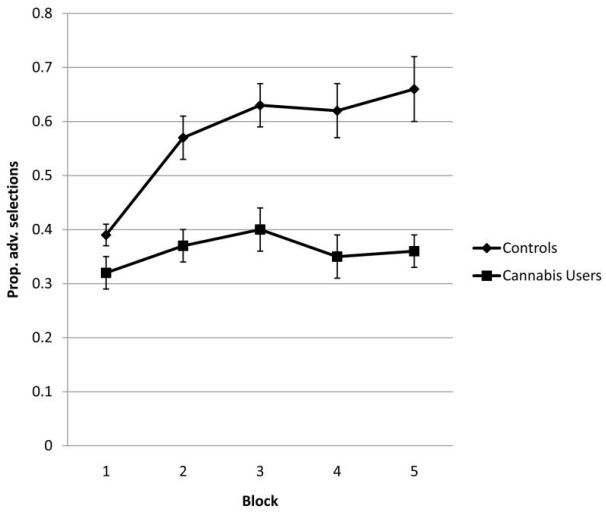


Figure 5.3: Plot of proportion of advantageous actions chosen ± SEM at each block adapted from Friedberg et al. [13]

From the plot above we can deduce that the control group performed significantly better than the group that consists of cannabis users. This is also a result that Friedberg

et al. [13] found. From a biological perspective it is not surprising that this is the case. In Section 2.2.3 we discussed the relevant brain regions for decision-making. It turns out that chronic cannabis users have an impaired mPFC. According to Section 2.2.3 this would lead to a poor assessment of the value of an action which would lead to choosing the disadvantageous decks more frequent. As already stated previously, our main interest is to compare the performance of the healthy group with the performance of bandit algorithms. In the next section we will introduce our first bandit algorithm for the IGT.

5.3 Bandit algorithms & the IGT

In the previous section, we have explored the pseudo-regret of our human participants at the IGT. Now we will see how an agent performs on the IGT that follows the so-called upper confidence bound (UCB) algorithm. In this section we will first introduce the UCB algorithm and then provide bounds on the pseudo-regret for this algorithm. Next, we will apply this algorithm to the IGT.

5.3.1 UCB algorithm

Previously, we introduced the stochastic multi-armed bandit setting. Here we will continue working in this setting. We adopt the same notation as before. Also, we use $T_i(n)$ to denote the number of times action/arm i has been performed during the first n time steps. That is

$$T_i(n) = \sum_{t=1}^{n} \mathbb{1}_{I_t=i}.$$

We will use

$$\Delta_i = \mu^* - \mu_i \quad \text{for } i = 1 \dots, K$$

to denote the difference between the expected value of the optimal action and the expected value of action i. In addition, we use

$$\hat{\mu}_{i,s} = \frac{1}{s} \sum_{t=1}^{s} X_{i,t}$$

to denote the estimated sample mean of action i. We can now present the UCB algorithm. The algorithm is described in the box below:

UCB algorithm

Protocol:
For $t = 1, \ldots, K$:
Initialize: $T_i(K) = 1, i = 1, 2, ..., K$.
For $t = K+1, K+2, \ldots, T$:

- Do:
$$I_t = \arg \max_{1 \leq i \leq K} \Big[\hat{\mu}_{i,T_i(t-1)} + (\psi^*)^{-1}\Big(\frac{\alpha \log t}{T_i(t-1)}\Big)\Big] \tag{5.1}$$
- Observe reward $X_{I_t,t}$

Several symbols in the box above are yet to be introduced. $\alpha > 0$, for example, is an input parameter for the UCB algorithm. The $(\psi^*)^{-1}$ term requires some additional explanation. We use $\psi(\lambda)$ with $\lambda > 0$ to denote a convex function such that the distribution of rewards X satisfies

$$\mathbb{E}\left[e^{\lambda(X - \mathbb{E}\,X)}\right] \leq e^{\psi(\lambda)} \text{ and } \mathbb{E}\left[e^{\lambda(\mathbb{E}\,X - X)}\right] \leq e^{\psi(\lambda)} \tag{5.2}$$

for all $\lambda > 0$. The star notation $\psi^*(\lambda)$ means that the Legendre-Fencher transformation has been applied to the convex function $\psi(\lambda)$. The Legendre-Fencher transformation is given by

$$\psi^*(\epsilon) = \sup_{\lambda \in \mathbb{R}} \Big(\lambda\epsilon - \psi(\lambda)\Big).$$

Thus, in order to perform the UCB algorithm on a stochastic multi-armed bandit problem, one has to find a convex function $\psi(\lambda)$ such that (5.2) holds for the distribution of rewards. Then one has to apply the Legendre-Fencher transformation on this convex function. Inverting this Legendre-Fencher transformation allows the learner to choose its next action according to (5.1).

5.3.2 UCB algorithm applied to IGT

We can apply the following transformation to each reward distribution:

$$\tilde{X} = \frac{X - \min X}{\max X - \min X}.$$

This ensures that each reward of each deck lies in the interval $[0, 1]$. If we apply this transformation to the IGT we get:

$$\tilde{X_A} = \begin{cases} 0.88 & \text{w.p. } p = 1/10 \\ 0.84 & \text{w.p. } p = 1/10 \\ 0.80 & \text{w.p. } p = 1/10 \\ 0.76 & \text{w.p. } p = 1/10 \\ 0.72 & \text{w.p. } p = 1/10 \\ 1.0 & \text{w.p. } p = 1/2 \end{cases} \quad \tilde{X_B} = \begin{cases} 0.0 & \text{w.p. } p = 1/10 \\ 1.0 & \text{w.p. } p = 9/10 \end{cases}$$

$$\tilde{X_C} = \begin{cases} 0.94 & \text{w.p. } p = 1/10 \\ 0.90 & \text{w.p. } p = 1/10 \\ 0.92 & \text{w.p. } p = 3/10 \\ 0.96 & \text{w.p. } p = 1/2 \end{cases} \quad \tilde{X_D} = \begin{cases} 0.76 & \text{w.p. } p = 1/10 \\ 0.96 & \text{w.p. } p = 9/10 \end{cases}$$

Note that we normalized the distribution of $100 + X_A$, $100 + X_B$, $50 + X_C$, and $50 + X_D$. Moreover, note that this transformation does not change the structure of the IGT. Decks A and B are still disadvantageous for the learner in the long-term. Indeed, we have

$$\mathbb{E}(\tilde{X_A}) = \mathbb{E}(\tilde{X_B}) = 0.90$$
$$\mathbb{E}(\tilde{X_C}) = \mathbb{E}(\tilde{X_D}) = 0.94$$

One might wonder why we would use this transformation for the rewards. The primary reason for this is that this transformation allows us to find a convex function $\psi(\lambda)$ that satisfies (5.2). Now we have that all our rewards (independent of the action) will lie in the interval $[0, 1]$. We can apply Hoeffding's lemma (see appendix) to obtain

$$\psi(\lambda) = \frac{\lambda^2}{8}$$

as convex function that satisfies (5.2). Next we must find the Legendre-Fencher transformation of $\psi(\lambda)$, before we can apply the UCB algorithm. Let $\epsilon > 0$ be given and define

$$f(\lambda) = \epsilon\lambda - \frac{\lambda^2}{8}.$$

Obviously we have

$$\frac{df}{d\lambda} = 0 \iff \lambda = 4\epsilon$$
$$\frac{df^2}{d^2\lambda} = -\frac{1}{4} < 0$$

Hence

$$\psi^*(\epsilon) = \sup_{\lambda \in \mathbb{R}} f(\lambda) = f(4\epsilon) = 4\epsilon^2 - 2\epsilon^2 = 2\epsilon^2$$

is our Legendre transform of $\psi(\lambda) = \frac{\lambda^2}{8}$. We also see that

$$(\psi^*)^{-1}(\lambda) = \sqrt{\frac{\lambda}{2}}.$$

Now we can perform the UCB algorithm for the IGT. We will thus perform the following algorithm:

UCB algorithm for the IGT

Protocol:
For $t = 1, \ldots, 4$:
Initialize: $T_i(4) = 1, i = 1, 2, 3, 4$.
For $t = 5, 6, \ldots, 95$:

- Do:

$$I_t = \arg \max_{1 \leq i \leq K} \left[\hat{\mu}_{i,T_i(t-1)} + \sqrt{\frac{\alpha \log t}{2T_i(t-1)}} \right]$$

- Observe reward $X_{I_t,t}$

The results of this learning algorithm are discussed in the next section.

5.3.3 Simulation of UCB strategy

In the previous section we derived the UCB algorithm for the IGT. Here we will simulate 15 independent trials, where the artificial agent that performs the IGT follows the UCB algorithm. The results are depicted below.

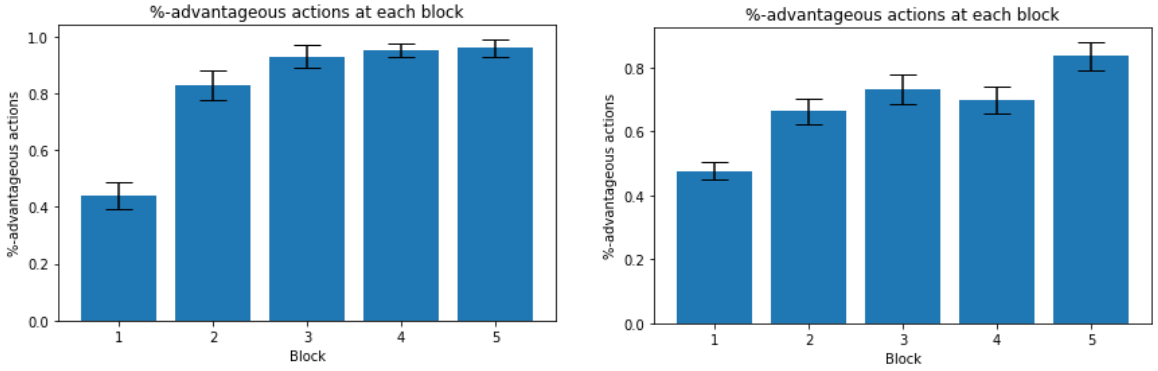


Figure 5.4: Bar graph of proportion of advantageous actions chosen ± SEM at each block.

As one can see from the plot above, the learner quickly learns to select deck C or D. Also, in the left bar graph we trained the agent with $\alpha = 0.01$ whereas in the right bar graph we trained our agent with $\alpha = 0.1$. We see that the former choice results in a higher proportion of advantageous actions chosen. This is also something we would expect. The parameter α in our UCB algorithm represents the exploration rate. By choosing a larger exploration rate we will also keep on choosing the disadvantageous decks with a higher frequency compared to the situation in which our exploration rate is lower. At a first glance we see that the UCB algorithm with an exploration rate of $\alpha = 0.01$ performs better than the cannabis and control group.

5.4 A one-state MDP

Another approach that we can take to address the IGT problem is to model the task as a one-state MDP. Since there is only one state we will drop the s notation in our analysis. We will use the same notation as before:

$$q_*(a) = \mathbb{E}(R_t | A_t = a).$$

This quantity is unknown to the learner, but is estimated by

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}.$$

It is easy to see that the estimated action-value function converges to q_*. Simply invoking the law of large numbers gives us the result. In the IGT we have four actions at our disposal. Let us fix an action a for the first n time steps. We can then write our estimated action function as:

$$\begin{aligned} Q_{n+1}(a) &= \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\Big(R_n + \sum_{i=1}^{n-1} R_i\Big) = \frac{1}{n}\Big(R_n + (n-1)Q_n(a)\Big) \\ &= Q_n(a) + \frac{1}{n}\Big(R_n + Q_n(a)\Big). \end{aligned}$$

Writing our estimated action-value function in this manner allows us to efficiently program the update rule. Thus for Q-learning we simply execute the following algorithm:

Q-learning for the IGT

Protocol:
Choose $\epsilon < 1$
Initialize for $a = 1, 2, 3, 4$:
$Q(a) \leftarrow 0$ and $N(a) \leftarrow 0$

- Loop until $t = T = 95$:

$$A = \begin{cases} \arg\max_a Q(a) & \text{w.p. } 1 - \epsilon \\ \sim \mathcal{U}\{1, 4\} & \text{w.p. } \epsilon \end{cases}$$

- Observe reward R
- $N(A) \leftarrow N(A) + 1$
- $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\Big(R - Q(A)\Big)$

If we train 15 agents independently, then we obtain the following plots.

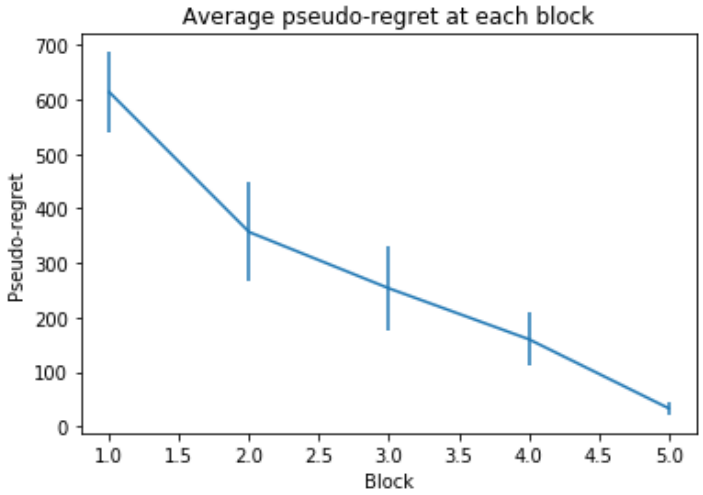


Figure 5.5: Plot of average pseudo-regret ± SEM at each block with $\epsilon = 0.1$.

In the plot above we set our exploration parameter $\epsilon = 0.1$. Again, at a first glance it appears to be the case that the one state MDP Q-learning algorithm performs better than the human participants. Note that if we increase our exploration parameter ϵ to 0.4 we mimic the behavior of the human participants more closely.

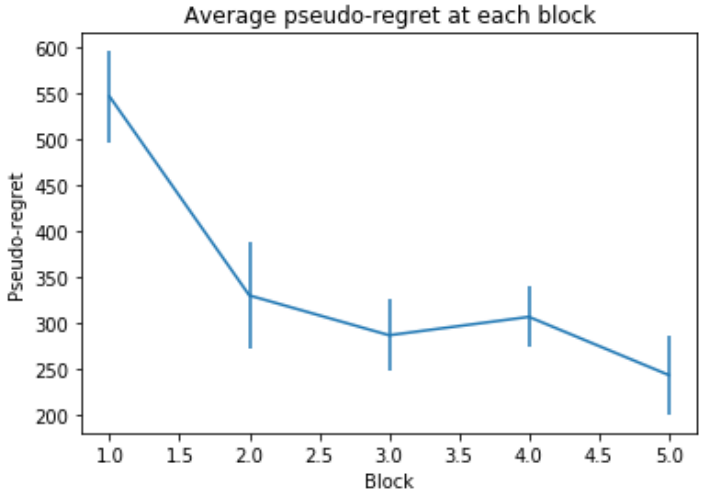


Figure 5.6: Plot of average pseudo-regret ± SEM at each block with $\epsilon = 0.4$.

If we compare the plot above with Figure 5.6, then we see that indeed this plot resembles the performance of human participants more closely compared to Figure 5.5. So far, we have implemented two learning algorithms and applied them to the IGT. In the discussion chapter, we will discuss which algorithm did best and if the results obtained from the learning algorithms differ significantly from the human participants. In the next section, we will use a Bayesian framework for the IGT. This framework, however, requires additional information before we can implement it. In this framework we need to know the possible rewards that can occur for each deck. We will therefore assume that we

know the reward structure. The probabilities with which these rewards occur, remain as before, unknown.

5.5 Bayesian inference for the IGT

As previously mentioned, we could also address the IGT from a Bayesian perspective. That is, we consider the $\boldsymbol{\theta}_i$ for $i \in \{1, 2, 3, 4\}$ as a random variable. Here $\boldsymbol{\theta}_i$ represents the probability vector associated with deck i. In this section we will build a Bayesian framework of the problem in order to 'solve' the IGT. We assume that we now know the reward structure of the learning task. Thus, we know that the reward distribution follows a categorical distribution. The main idea now is to treat $\boldsymbol{\theta}_i$ for $i \in \{1, 2, 3, 4\}$ as a random variable that follows the Dirichlet distribution. The reward of action i in turn follows the categorical distribution with $\boldsymbol{\theta}_i$ for $i \in \{1, 2, 3, 4\}$. By doing so we can model the distribution of the reward at each deck. Consequently, we can utilize an algorithm that uses this information properly. But, before we can perform this algorithm we will have to build up the Bayesian framework. Let us begin by introducing some definitions and theorems.

Definition 5.3. A categorical distribution (denoted as $\text{Cat}(\boldsymbol{\theta})$) is a discrete probability distribution that has a finite sample space of k items. The labeling of the k items is not important, but $\{1, \ldots, k\}$ is often taken as notation for the sample space. The probability mass function is given by:

$$\mathbb{P}(X = x|\boldsymbol{\theta}) = p(x|\boldsymbol{\theta}) = \theta_x$$

here $\boldsymbol{\theta} = (\theta_1 \ldots, \theta_k)$ and θ_i represents the probability of seeing item i. We also have:

$$\sum_{i=1}^{k} \theta_i = 1 \quad \text{and} \quad \theta_i \geq 0 \ \forall i. \tag{5.3}$$

In our case we have that $X_1 = 100 + X_A$, $X_2 = 100 + X_B$, $X_3 = 50 + X_C$, and $X_4 = 50 + X_D$ are all random variables that follow the categorical distribution with parameters $\boldsymbol{\theta_1}$, $\boldsymbol{\theta_2}$, $\boldsymbol{\theta_3}$, and $\boldsymbol{\theta_4}$, respectively. These parameters are unknown for the learning agent. Notice that if $k = 2$ as is the case for deck B and D, then we get that the net gain/loss is a random variable that follows the Bernoulli distribution. The categorical distribution can also be seen as a generalization of the Bernoulli distribution.

The idea of the Bayesian approach is to estimate a parameter $\theta \in \Theta$ (in our case $\boldsymbol{\theta_i}$), which is treated as a random variable. The parameter $\theta \in \Theta$ that needs to be estimated for a probability model, can be estimated with the Bayesian approach by considering the posterior. The posterior is given by

$$\mathbb{P}(\theta|X_i = x_1, \ldots, X_i = x_n, \lambda)$$

here $x_1, \ldots, x_n$ represent the net gain/losses received after selecting deck i. λ is another parameter which defines a quantity known as the prior. The prior, $\mathbb{P}(\theta|\lambda)$, is a probability distribution that one uses to model the distribution of the unknown parameter θ, before

any observations are made. The λ's are assumed to be known parameters. The prior is needed in our analysis in order to compute the posterior. Why do we want to compute the posterior? Well, later on in this section we will follow a bandit algorithm that requires the posterior. Furthermore, this algorithm also requires that a certain condition is satisfied. The condition is that both the prior and the posterior are in the same probability distribution family. Another condition that must be met is that the chosen prior must be the conjugate prior of the likelihood function. We will prove that both conditions are met if we take a Dirichlet distribution as prior for $\boldsymbol{\theta}_i$ and set

$$\mathbb{P}(X = x|\boldsymbol{\theta}) = p(x|\boldsymbol{\theta}) = \theta_x$$

as likelihood function. We will first introduce some definitions before we prove that these conditions are met for our choices of the prior and likelihood.

Definition 5.4 (prior conjugacy)**.** Suppose we have a distribution $p(x|\theta)$ and a prior distribution on θ denoted as $p(\theta|\lambda)$. Then we say that the parameter prior distribution $p(\theta|\lambda)$ is the prior conjugate of the distribution $p(x|\theta)$ if

$$p(x|\theta)p(\theta|\lambda) = \mathcal{C}\, p(\theta|\lambda')$$

holds. $\mathcal{C}$ denotes a constant that does not depend on θ. $p(\theta|\lambda)$ and $p(\theta|\lambda')$ have the same analytical expression, but may have a different parameter value.

In our learning task we replace θ with $\boldsymbol{\theta_i}$. The x represents a new unobserved reward. The term $p(x|\boldsymbol{\theta_i}) = \mathbb{P}(X = x|\boldsymbol{\theta_i})$ represents the probability mass function with $X \sim \text{Cat}(\boldsymbol{\theta_i})$. If we want to find the conjugate prior of the categorical distribution, then we must use a distribution that has support that satisfies (5.3). A distribution that satisfies (5.3) is the Dirichlet distribution. Recall the definition of the Dirichlet distribution:

Definition 5.5. The Dirichlet distribution denoted as $\text{Dir}(\boldsymbol{\alpha})$ with parameter $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K)$, $\alpha_i > 0$ for all $1 \leq i \leq K$, has a probability density function that is given by:

$$p(\boldsymbol{x}|\boldsymbol{\alpha}) = B(\boldsymbol{\alpha}) \prod_{i=1}^{K} x_i^{\alpha_i - 1}$$

here $\boldsymbol{x} = (x_1, \ldots, x_K)$ such that $\{x_i\}_{i=1}^K$ belongs to the $K-1$ simplex. This means that $\{x_i\}_{i=1}^K$ satisfies (5.3). The normalizing constant $B(\boldsymbol{\alpha})$ is given by:

$$B(\boldsymbol{\alpha}) = \frac{\Gamma(\sum_{i=1}^K \alpha_i)}{\prod_{i=1}^K \Gamma(\alpha_i)}. \tag{5.4}$$

Γ represents the gamma function and is given by $\int_0^\infty u^{z-1}e^{-u}du$ for $z > 0$.

Now that we have introduced the Dirichlet distribution and the notion of prior conjugacy we can prove the lemma on the next page.

Lemma 5.6. *The categorical distribution has the Dirichlet distribution as conjugate prior.*

Proof. We must show that

$$p(x|\boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{\alpha}) = \mathcal{C}\, p(\boldsymbol{\theta}|\boldsymbol{\alpha}')$$

holds for some constant $\mathcal{C}$ that does not depend on $\boldsymbol{\theta}$. This is indeed the case as can be seen by using the definitions of the probability distributions:

$$\begin{aligned} p(x|\boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{\alpha}) &= \theta_x B(\boldsymbol{\alpha}) \prod_{i=1}^{K} \theta_i^{\alpha_i - 1} = B(\boldsymbol{\alpha}) \prod_{i=1}^{K} \theta_i^{\alpha_i - 1 + \mathbb{1}_{x=i}} \\ &= B(\boldsymbol{\alpha}) \frac{B(\tilde{\boldsymbol{\alpha}})}{B(\tilde{\boldsymbol{\alpha}})} \prod_{i=1}^{K} \theta_i^{\tilde{\alpha_i} - 1} = \mathcal{C} p(\boldsymbol{\theta}|\tilde{\boldsymbol{\alpha}}) \end{aligned}$$

where $\mathcal{C} = \frac{B(\boldsymbol{\alpha})}{B(\tilde{\boldsymbol{\alpha}})}$ does not depend on $\boldsymbol{\theta}$ and where $\tilde{\boldsymbol{\alpha}} = (\tilde{\alpha_1}, \ldots, \tilde{\alpha_K})$ with $\tilde{\alpha_i} = \alpha_i + \mathbb{1}_{x=i}$. □

Thus we have shown that one of the required conditions holds. Next, we will prove that the other condition is also satisfied.

Theorem 5.7. *The posterior of $\boldsymbol{\theta}$ is the Dirichlet distribution, given that the prior of $\boldsymbol{\theta}$ is Dirichlet and that the likelihood function is the categorical distribution.*

Proof. We obtain after n observations, the dataset $\mathcal{D} = \{x_1, \ldots, x_n\}$. The posterior of $\boldsymbol{\theta}$ is given by:

$$p(\boldsymbol{\theta}|\mathcal{D}, \boldsymbol{\alpha}) = \frac{p(\boldsymbol{\theta}|\boldsymbol{\alpha}) \prod_{i=1}^{n} p(x_i|\boldsymbol{\theta})}{p(\mathcal{D}, \boldsymbol{\alpha})}. \tag{5.5}$$

We begin by rewriting the numerator:

$$\begin{aligned} p(\boldsymbol{\theta}|\boldsymbol{\alpha}) \prod_{i=1}^{n} p(x_i|\boldsymbol{\theta}) &= B(\boldsymbol{\alpha}) \prod_{i=1}^{K} \theta_i^{\alpha_i - 1} \prod_{i=1}^{n} \boldsymbol{\theta}_{x_i} \\ &= B(\boldsymbol{\alpha}) \prod_{i=1}^{K} \theta_i^{\alpha_i - 1} \prod_{j=1}^{K} \boldsymbol{\theta}_j^{\sum_{i=1}^{n} \mathbb{1}_{j=x_i}} \\ &= B(\boldsymbol{\alpha}) \prod_{j=1}^{K} \theta_j^{\alpha_j - 1 + \sum_{i=1}^{n} \mathbb{1}_{j=x_i}} \\ &= \mathcal{C} p(\boldsymbol{\theta}|\tilde{\boldsymbol{\alpha}}) \end{aligned}$$

where $\mathcal{C} = \frac{B(\boldsymbol{\alpha})}{B(\tilde{\boldsymbol{\alpha}})}$ does not depend on $\boldsymbol{\theta}$ and where $\tilde{\boldsymbol{\alpha}} = (\tilde{\alpha_1}, \ldots, \tilde{\alpha_K})$ with $\tilde{\alpha_j} = \alpha_j + \sum_{i=1}^{n} \mathbb{1}_{j=x_i}$. Next we compute the denominator of (5.5). A simple calculation reveals that:

$$p(\mathcal{D}, \boldsymbol{\alpha}) = \int_{\Theta} p(\boldsymbol{\theta}|\boldsymbol{\alpha}) \prod_{i=1}^{n} p(x_i|\boldsymbol{\theta}) d\boldsymbol{\theta} = \mathcal{C} \int_{\Theta} p(\boldsymbol{\theta}|\tilde{\boldsymbol{\alpha}}) d\boldsymbol{\theta} = \mathcal{C}.$$

Here we integrated over the $K-1$ simplex Θ, the support of the Dirichlet distribution. Thus, we indeed have that (5.5) reduces to the Dirichlet distribution. □

Note that from the proof above we see that the posterior has hyperparameter

$$\begin{aligned}\tilde{\boldsymbol{\alpha}} &= \left(\tilde{\alpha_1}, \ldots, \tilde{\alpha_K}\right) \\ &= \left(\alpha_1 + \sum_{i=1}^{n} \mathbb{1}_{x_i=1}, \ldots, \alpha_K + \sum_{i=1}^{n} \mathbb{1}_{x_i=K}\right).\end{aligned}$$

If we set the prior hyperparameters $\alpha_1, \ldots, \alpha_K$ all equal to one, then there are $\tilde{\alpha_1} - 1, \ldots, \tilde{\alpha_K} - 1$ occurrences of category $1, \ldots, K$ respectively. We can apply Theorem 5.7 for each deck in the IGT. This ensures that for each deck the posterior distribution remains in the same distribution family as the prior. We will use the following label of outcomes for the reward structure for the IGT:

$$X_1 = \begin{cases} -50 & \text{outcome 1 w.p. } p = 1/10 \\ -100 & \text{outcome 2 w.p. } p = 1/10 \\ -150 & \text{outcome 3 w.p. } p = 1/10 \\ -200 & \text{outcome 4 w.p. } p = 1/10 \\ -250 & \text{outcome 5 w.p. } p = 1/10 \\ 100 & \text{outcome 6 w.p. } p = 1/2 \end{cases} \quad X_2 = \begin{cases} -1150 & \text{outcome 1 w.p. } p = 1/10 \\ 100 & \text{outcome 2 w.p. } p = 9/10 \end{cases}$$

$$X_3 = \begin{cases} 25 & \text{outcome 1 w.p. } p = 1/10 \\ -25 & \text{outcome 2 w.p. } p = 1/10 \\ 0 & \text{outcome 3 w.p. } p = 3/10 \\ 50 & \text{outcome 4 w.p. } p = 1/2 \end{cases} \quad X_4 = \begin{cases} -200 & \text{outcome 1 w.p. } p = 1/10 \\ 50 & \text{outcome 2 w.p. } p = 9/10 \end{cases}$$

For deck $1, 2, 3, 4$ we thus have $K = 6, 2, 4, 2$ respectively. The algorithm we will now follow is known as Thompson sampling. The steps that are taken in Thompson sampling are given in the box below.

Thompson sampling for the IGT

Protocol:
For each $\boldsymbol{\theta_i}$ with $i \in \{1, 2, 3, 4\}$ set

- $\boldsymbol{\theta_i} \sim \text{Dir}(\boldsymbol{\alpha_i})$

as prior. Where $\boldsymbol{\alpha_1} = (1, 1, 1, 1, 1, 1)$, $\boldsymbol{\alpha_2} = (1, 1)$, $\boldsymbol{\alpha_3} = (1, 1, 1, 1)$ and $\boldsymbol{\alpha_4} = (1, 1)$
For each $t = 1, \ldots, 95$ do:

- Draw a sample $\boldsymbol{\theta_i} \sim \text{Dir}(\boldsymbol{\alpha_i})$ for each $i \in \{1, 2, 3, 4\}$
- Compute $\mathbb{E}_{\boldsymbol{\theta_i}}[X_i]$ for each $i \in \{1, 2, 3, 4\}$
- $a_t = \arg\max_i \mathbb{E}_{\boldsymbol{\theta_i}}[X_i]$
- Observe outcome in deck i
- Update parameter $\boldsymbol{\alpha_i}$

As one would expect, the algorithm performs quite well on the IGT. The result of one simulation is provided below.

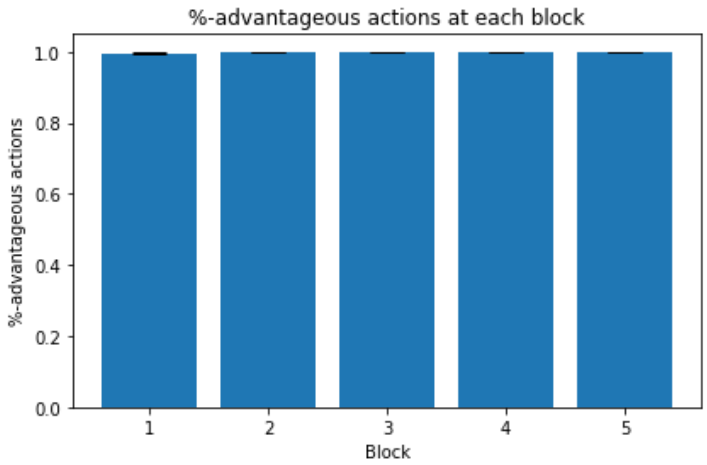


Figure 5.7: Bar graph of proportion of advantageous actions chosen ± SEM at each block.

From the bar graph above we see that Thompson sampling (TS) almost always chooses the advantageous decks and thus performs very well on the IGT. In the next section, we will compare all discussed bandit algorithms to each other and to the results of the human participants.

5.6 Comparing bandit algorithms with human decision-making

We have applied three algorithms to the IGT. As one would probably a priori expect, all three learning algorithms outperform (on average) the human participants. We say on average here, because in some instances Q-learning or the UCB algorithm **might** take a long time before the optimal arms are found. We will therefore not compare one simulation of an algorithm against the performance of the human participants. This might after all give misleading results. The performance measure we will use is the proportion of optimal actions chosen. We have 15 human participants their average proportion of optimal actions chosen equals 0.67. Thus, 0.67 is the threshold that the artificially trained agents must surpass. The advantage of training artificial agents is that we can train them as much as we want. This also enables us to estimate their performance accurately. We will use 100 simulations for each learning algorithm. Each simulation itself consists of 15 independently trained agents. The proportions of optimal actions taken by the 15 agents are then averaged out. This process is then repeated for another simulation. This repeated simulation yields another average of the proportions of optimal actions taken by the 15 agents. This newly obtained average together with the previously obtained average(s) are then averaged together to yield a new average. This process is repeated 100 times for each learning algorithm. The results are depicted in the graph below.

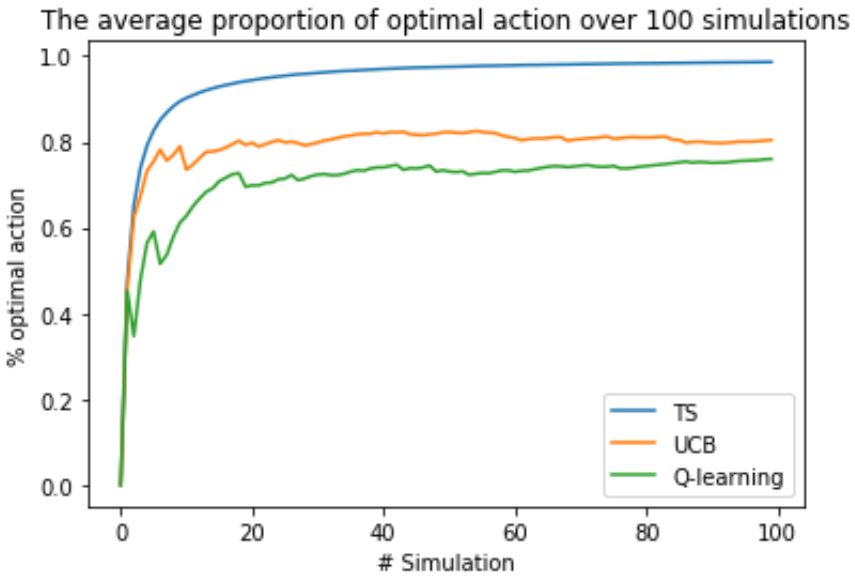


Figure 5.8: Average proportion of optimal actions chosen for different learning algorithms.

A few remarks are in order. Firstly, the TS algorithm performs best at this learning task. The average proportion of optimal actions chosen after 100 simulation equals 0.99 for TS. It is not surprising that this bandit algorithm performs best at this task. In fact, it would be surprising if it actually did not perform best at this learning task. The reason for this is that TS uses additional information about the reward structure. It takes into account that the reward at each deck follows a categorical distribution. The other learning algorithms do not take this into account. Secondly, we observe that the UCB algorithm seems to outperform the Q-learning algorithm, but as the number of simulation increases their performances do not deviate too much from each other. The average proportion of optimal actions chosen after 100 simulation for the UCB algorithm and Q-learning equals 0.81 and 0.77 respectively. The UCB algorithm and Q-learning both surpass the results of the participants. Thus, artificial learning outperforms biological learning in the IGT. Moreover, we conclude that TS exceeds both UCB and Q-learning as well as the participants. Motivated by the performance of TS in the IGT, we will now explore the performance of this Bayesian bandit algorithm in another biomedical setting. Namely, the setting of clinical trials. The framework and definitions we introduced during this chapter will also be used in the upcoming chapter.

6 Bandit algorithms applied to clinical trials

We have seen that TS performs quite well on the IGT. The IGT is in essence an MAB problem. As will become clear later on, designing a clinical trial can mathematically also be regarded as an MAB problem. This gives us therefore good reason to investigate the performance of TS on the design of clinical trials. The performance of TS together with other bandit algorithms, will be discussed in this chapter.

6.1 Design of clinical trials

The design of clinical trials can mathematically be viewed as an MAB problem. In a clinical trial, doctors often want to know if there is any difference in effectiveness between treatments. Say we have a population size of N patients that suffer from disease X. And let us assume that there are two available treatments (M_1 and M_2). The doctor that wants to find out if there is a difference in effectiveness between the treatments, has to design a clinical trial. In most cases a randomized clinical trial is used to test for the difference in effectiveness between the treatments. In fact, a randomized clinical trial is considered the golden standard. In a randomized clinical trial each patient is randomly assigned to either M_1 or M_2. Now let us assume that treatment M_1 is not very effective in treating disease X. The randomized clinical trial design then has as disadvantage that many patients may receive treatment M_1, which is not effective in treating the disease. This problem can be negated by modeling the clinical trial as an MAB problem. In the subsequent sections we will compare three possible designs. These designs include the already mentioned randomized clinical trial design. We will also examine the so-called 'explore-exploit' design. Moreover, we will discuss the Bayesian adaptive design. For each design we will address two key questions (Section 6.2). The first question is: what proportion of patients received an effective treatment? The second question is: did the design allow for detection of a difference in effectiveness between the treatments? Besides these questions we will also derive upper-bounds for the pseudo-regrets for each clinical design strategy. We begin with the currently most often used design, the randomized clinical trial design.

6.1.1 Randomized clinical trial design

In a randomized clinical trial each patient is randomly assigned to either M_1 or M_2. For each point in time $t = 1, \ldots, N$ the patient must either be allocated to M_1 or M_2. With θ_1 we will denote the probability that treatment M_1 is successful in treating disease X. Similarly, with θ_2 we will denote the probability that treatment M_2 is successful in treating disease X. We can thus model the design of a clinical trial (the allocation of patients to a treatment) as an MAB. At each point in time a patient must be allocated to either M_1 or M_2. After an action has been performed the agent (/doctor) is informed about the result of the treatment. A randomized clinical trial can also be viewed as an MAB. This strategy, however, does not take into consideration the outcome of each action (allocation). By following this design strategy we can explicitly compute the pseudo-regret (see Definition 5.2). To do this we first rewrite the pseudo-regret a bit:

$$\bar{R}_N = N\mu^* - \mathbb{E}_{I_1,\ldots,I_N}\Big\{\sum_{t=1}^{N} X_{I_t,t}\Big\} = \Big(\sum_{i=1}^{K} \mathbb{E}\, T_i(N)\Big)\mu^* - \mathbb{E}\sum_{i=1}^{K} T_i(N)\mu_i = \sum_{i=1}^{K} \Delta_i\, \mathbb{E}\, T_i(N).$$

Note that we now use N (population size) instead of T. We will assume w.l.o.g. that $\theta_1 \geq \theta_2$. Also, in our case we only consider two treatments. Hence $K = 2$. By writing the pseudo-regret in this form we can compute it directly.

$$\begin{aligned}
\bar{R}_N &= \sum_{i=1}^{K} \Delta_i\, \mathbb{E}\, T_i(N) = \Delta_2\, \mathbb{E}\, T_2(N) = (\theta_1 - \theta_2)\, \mathbb{E}\Big(\sum_{t=1}^{N} \mathbb{1}_{I_t=2}\Big) \\
&= (\theta_1 - \theta_2)\Big(\mathbb{E}\, \mathbb{1}_{I_1=2} + \ldots + \mathbb{E}\, \mathbb{1}_{I_N=2}\Big) \\
&= (\theta_1 - \theta_2)\Big(\mathbb{P}(I_1 = 2) + \ldots + \mathbb{P}(I_N = 2)\Big) = \frac{N(\theta_1 - \theta_2)}{2}.
\end{aligned}$$

In this case, since the strategy to follow was not that complicated we were able to compute the pseudo-regret directly. In the next subsection we will analyze the explore-exploit design strategy.

6.1.2 The explore-exploit design strategy

As the name already suggests, in the explore-exploit design we will first explore the effectiveness of the two treatments and then exploit our learned knowledge. In particular, n patients are allocated to treatment M_1 and n patients are allocated to treatment M_2. The remaining $N - 2n$ patients are allocated to the treatment with the highest empirical mean. We will denote

$$\hat{\mu}_i = \frac{1}{n}\sum_{j=1}^{n} X_{i,j}$$

as empirical mean for treatment $i = 1, 2$ at the end of the exploration phase. We have that $X_{1,j} \sim \text{Ber}(\theta_1)$ and $X_{2,j} \sim \text{Ber}(\theta_2)$. We can now prove the following lemma.

Lemma 6.1. *The pseudo-regret of the explore-exploit design strategy is upper-bounded by*

$$(\theta_1 - \theta_2)\Big(n + (N-2n)\exp\Big(-\frac{n(\theta_1-\theta_2)^2}{2}\Big)\Big)$$

assuming $\theta_1 \geq \theta_2$

Proof. As before we assume w.l.o.g. that $\theta_1 \geq \theta_2$ holds. We will first write out the pseudo-regret as follows:

$$\begin{aligned}
\bar{R}_N &= \sum_{i=1}^{2} \Delta_i \,\mathbb{E}\, T_i(N) = \Delta_2 \,\mathbb{E}\, T_2(N) = (\theta_1 - \theta_2)\,\mathbb{E}\Big(\sum_{t=1}^{N} \mathbb{1}_{I_t=2}\Big) \\
&= (\theta_1 - \theta_2)\Big(\mathbb{E}\,\mathbb{1}_{I_1=2} + \ldots + \mathbb{E}\,\mathbb{1}_{I_N=2}\Big) \\
&= (\theta_1 - \theta_2)\Big(\mathbb{P}(I_1 = 2) + \ldots + \mathbb{P}(I_N = 2)\Big) \\
&= (\theta_1 - \theta_2)\Big(n + \mathbb{P}(I_{2n+1} = 2) + \ldots + \mathbb{P}(I_N = 2)\Big) \\
&= (\theta_1 - \theta_2)\Big(n + (N-2n)\mathbb{P}(\hat{\mu}_2 \geq \hat{\mu}_1)\Big).
\end{aligned}$$

Now we have to find an expression for:

$$\mathbb{P}(\hat{\mu}_2 \geq \hat{\mu}_1) = \mathbb{P}\Big(\frac{1}{n}\sum_{j=1}^{n} X_{2,j} \geq \frac{1}{n}\sum_{j=n+1}^{2n} X_{1,j}\Big).$$

Recall that $X_{1,j} \sim \mathrm{Ber}(\theta_1)$ and $X_{2,j} \sim \mathrm{Ber}(\theta_2)$. Thus we rewrite the expression above in the following way:

$$\mathbb{P}\Big(\frac{1}{n}\sum_{j=1}^{n} X_{2,j} \geq \frac{1}{n}\sum_{j=n+1}^{2n} X_{1,j}\Big) = \mathbb{P}\Big(X \geq Y\Big) = \mathbb{P}\Big(X - Y \geq 0\Big) \tag{6.1}$$

with $X := \sum_{j=1}^{n} X_{2,j} \sim \mathrm{Bin}(n, \theta_2)$ and $Y := \sum_{j=n+1}^{2n} X_{1,j} \sim \mathrm{Bin}(n, \theta_1)$. Now we can define $Z_1, \ldots, Z_n$, with

$$Z_i = \begin{cases} 1 & \text{w.p. } (1-\theta_1)\theta_2 \\ 0 & \text{w.p. } \theta_1\theta_2 + (1-\theta_1)(1-\theta_2) \\ -1 & \text{w.p. } (1-\theta_2)\theta_1 \end{cases}$$

We can then use this r.v. to rewrite (6.1) as:

$$\begin{aligned}
\mathbb{P}\Big(X - Y \geq 0\Big) = \mathbb{P}\Big(\frac{1}{n}\sum_{i=1}^{n} Z_i \geq 0\Big) &= \mathbb{P}\Big(\frac{1}{n}\sum_{i=1}^{n} Z_i - (\theta_2 - \theta_1) \geq \theta_1 - \theta_2\Big) \\
&\leq \exp\Big(-\frac{n(\theta_1-\theta_2)^2}{2}\Big).
\end{aligned}$$

Here we again used Hoeffding's inequality. The result now follows by using this upper-bound. □

We will now continue with the last design strategy that we will consider.

6.1.3 Bayesian adaptive design

The Bayesian adaptive design strategy is based upon the same principles we encountered in Section 5.5. We can not follow the exact same algorithm, because our likelihood function is in our case not the categorical distribution. We are faced with a two-armed Bernoulli bandit problem, which is in some sense an easier problem than the IGT. Our likelihood function now becomes the Bernoulli distribution. We claim that the following holds:

Lemma 6.2. *The posterior of θ is the Beta distribution, given that the prior of θ is the Beta distribution and that the likelihood function is the Bernoulli distribution.*

Proof. We obtain after n observations, the dataset $\mathcal{D} = \{x_1, \ldots, x_n\}$. The posterior of θ is given by:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}. \tag{6.2}$$

We have to prove that (6.2) is the probability density function of the Beta distribution. First recall the definition of the Beta distribution. The Beta distribution is given by:

$$p(\theta) = f(\theta; \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} = \frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}.$$

Here θ lies in the interval $[0,1]$ and $\alpha > 0, \beta > 0$ are the shape parameters. We begin by rewriting the numerator of (6.2). First note that

$$p(\mathcal{D}|\theta) = p(x_1, \ldots, x_n|\theta) = \prod_{i=1}^{n} \theta^{x_i}(1-\theta)^{1-x_i} = \theta^{\sum_{i=1}^n x_i}(1-\theta)^{n-\sum_{i=1}^n x_i}$$

holds by independence. Thus we can write the numerator of (6.2) as:

$$p(\mathcal{D}|\theta)p(\theta) = \frac{\theta^{\sum_{i=1}^n x_i+\alpha-1}(1-\theta)^{n-\sum_{i=1}^n x_i+\beta-1}}{B(\alpha,\beta)}.$$

Next we will rewrite the denominator:

$$\begin{aligned} p(\mathcal{D}) &= \int_0^1 p(\mathcal{D}|\theta)p(\theta)d\theta = \int_0^1 \frac{\theta^{\sum_{i=1}^n x_i+\alpha-1}(1-\theta)^{n-\sum_{i=1}^n x_i+\beta-1}}{B(\alpha,\beta)}d\theta \\ &= \frac{B(\sum_{i=1}^n x_i + \alpha, n - \sum_{i=1}^n x_i + \beta)}{B(\alpha,\beta)}. \end{aligned}$$

Now we indeed see that (6.2) reduces to:

$$p(\theta|\mathcal{D}) = \frac{\theta^{\sum_{i=1}^n x_i+\alpha-1}(1-\theta)^{n-\sum_{i=1}^n x_i+\beta-1}}{B(\sum_{i=1}^n x_i + \alpha, n - \sum_{i=1}^n x_i + \beta)}.$$

Hence, the posterior is indeed the Beta distribution with parameters: $\sum_{i=1}^n x_i + \alpha$ and $n - \sum_{i=1}^n x_i + \beta$. □

Now that we have proved Lemma 6.2 we can again perform Thompson sampling.

Thompson sampling

Protocol:
For θ_1 and θ_2 set

- $\theta_i \sim \text{Beta}(\alpha_i, \beta_i)$ for $i \in \{1, 2\}$

as prior. Where $\alpha_1 = 1$, $\beta_1 = 1$ and $\alpha_2 = 1$, $\beta_2 = 1$
For $t = 1, \ldots, N$ do:

- Draw a sample $\theta_i \sim \text{Beta}(\alpha_i, \beta_i)$ for $i \in \{1, 2\}$
- $a_t = \arg\max_i\{\theta_i\}$ for $i \in \{1, 2\}$
- Observe outcome
- Update parameters α_{a_t} and β_{a_t}

In the previous sections we derived an upper-bound for the pseudo-regret for each designing strategy. For Thompson sampling one can derive an upper-bound of order:

$$\mathcal{O}\Big(\frac{\log N}{\theta_1 - \theta_2} + \frac{1}{(\theta_1 - \theta_2)^3}\Big).$$

Again, assuming w.l.o.g that $\theta \geq \theta_2$ holds. (For a proof see [14]) Now that we have introduced the relevant designing strategies we can see how they perform in practice.

6.2 Numerical results

In this section we will address the two questions we posed at the beginning of this chapter. That is: what proportion of patients received an effective treatment ? And did the design allow for detection of a difference in effectiveness between the treatments? Before we use simulations to answer these question, we must first pick values for our parameters. For all three algorithms we will set $N = 200$, $\theta_1 = 0.7$, and $\theta_2 = 0.3$. The explore-exploit design strategy, however, still requires an additional input. Namely, the parameter n. We can find the optimal value n by considering the upper-bound of the pseudo-regret we deduced in the previous section (see lemma 6.1). We want to minimize this expression as a function of n. This can not be done analytically. We are therefore resorted to numerical methods. We will use the secant method to find this optimal value for n. It turns out that the upper-bound for the pseudo-regret of the explore-exploit strategy is minimized for $n = 32$. Now that we have chosen our parameters, we can present the numerically obtained results. These are depicted in the table on the next page.

Strategy	proportion effective treatment ($\boldsymbol{p}$)	proportion rejected H_0
Randomization	0.500	1.00
Explore-exploit	0.636	0.986
Adaptive Bayesian	0.680	0.775

Table 6.1: Performance of the different strategies with $\theta_1 = 0.7$, $\theta_2 = 0.3$ and $N = 200$.

For each strategy we computed the proportion of patients that received an effective treatment and the proportion of rejected null hypotheses. Here we took

$$H_0 : \theta_1 = \theta_2$$
$$H_1 : \theta_1 \neq \theta_2$$

as null hypothesis and alternative hypothesis. We set $\alpha = 0.05$ as is usual in a clinical trial. We used 10^5 simulations for each strategy. At each simulation we computed the proportion of patients that received an effective treatment. Moreover, we used a Chi-squared test to test our null hypothesis at each simulation. We then took the average of the obtained results from the 10^5 simulations. The results of this method are depicted in the table above. Note that the simulations for the randomization indeed gave us the result we would a priori expect. By following the randomization strategy we were able to reject the null hypothesis in almost all cases. This is of course a desirable result for the researchers that conducts the clinical trial. Their aim is twofold. First they (preferably) want to demonstrate a difference in effectiveness between the two treatments. Secondly, they want as many patients as possible to benefit from an effective treatment. We can see that this goal is not met, if one follows the randomization strategy. In this situation, on average only half of the patients (100) receive an effective treatment. When we undertake a frequentist approach, we see that the proportion of patients that receives an effective treatment increases considerably. At the same time, the proportion of rejected null hypothesis decreases only slightly. In this particular setting, it is a better idea to adopt the explore-exploit design rather than the randomization design. We can also undertake the adaptive Bayesian design strategy. This design allows for an even bigger $\boldsymbol{p}$, but the proportion of rejected null hypothesis also decreases. Researchers can opt for the adaptive Bayesian design strategy if the main priority of the clinical trials is to get $\boldsymbol{p}$ as high as possible. If on the other hand, rejection of the null hypothesis is the main priority of the trial then the researchers can either opt for the randomization design or the explore-exploit design. Here the latter strategy is preferred above the former, because the explore-exploit in addition has a significant higher $\boldsymbol{p}$ than the randomization strategy.

We can also look at different settings. For instance, which strategy should be opted for if $\theta_1 - \theta_2$ is small/big. In the tables below we present the results.

Strategy	proportion effective treatment ($\boldsymbol{p}$)	proportion rejected H_0
Randomization	0.500	1.00
Explore-exploit	0.800	1.00
Adaptive Bayesian	0.837	0.955

Table 6.2: Performance of the different strategies with $\theta_1 = 0.85$, $\theta_2 = 0.15$ and $N = 200$.

Strategy	proportion effective treatment ($\boldsymbol{p}$)	proportion rejected H_0
Randomization	0.500	0.348
Explore-exploit	0.501	0.361
Adaptive Bayesian	0.532	0.179

Table 6.3: Performance of the different strategies with $\theta_1 = 0.56$, $\theta_2 = 0.44$ and $N = 200$.

The first table has $\theta_1 = 0.85$ and $\theta_2 = 0.15$ as parameter values, whereas the second table has $\theta_1 = 0.56$ and $\theta_2 = 0.44$ as parameter values. Because we changed the parameters θ_1 and θ_2 we must also compute n again for the explore-exploit strategy. In the first table we used $n = 98$ and in the second table we used $n = 15$ as value. Note that in the former case the explore-exploit strategy basically boils down to dividing the patient population into two groups. This is because the two parameter values θ_1 and θ_2 do not differ too much from each other. Hence, more patients are required in the exploration phase to determine the most beneficial treatment option. In the first table we increased the difference between θ_1 and θ_2. We see that the explore-exploit strategy is better suited in this situation than the randomization strategy (both have the same proportion rejected H_0, but the explore-exploit design has a higher $\boldsymbol{p}$). The adaptive Bayesian design slightly outperforms the explore-exploit strategy, but on the other hand it is slightly outperformed by the explore-exploit strategy with respect to the proportion rejected H_0. Both the explore-exploit and adaptive Bayesian design strategy seem to perform better than the randomization design.

In table 6.3 we have $\theta_1 = 0.56$ and $\theta_2 = 0.44$ as parameter values. In this situation we again have that the explore-exploit design should be preferred above randomization, since the proportion of rejected H_0 is higher for this design. It is also obvious that the $\boldsymbol{p}$ values for each strategy becomes closer to each other. This is because of the fact that $\theta_1 - \theta_2$ becomes smaller. We conclude that in all cases (Table 6.1, 6.2, and 6.3) both the explore-exploit design and the adaptive Bayesian design outperform the randomization design. Depending on the preference of the researchers, one can either opt for the Bayesian adaptive design (if the main priority is to maximize $\boldsymbol{p}$) or for the explore-exploit design (if both $\boldsymbol{p}$ and the proportion of rejected H_0 is important). In the next chapter we will discuss the limitations of viewing a clinical trial design as an MAB. Moreover, we will discuss all of the results we obtained during this thesis.

7 Discussion

In this thesis we looked at both reinforcement and bandit algorithms. The first section of the thesis dealt with RL algorithms. In the literature, there is still debate about whether MC methods or Q-learning converges faster. To address this issue we considered MRPs. Our goal was to examine different types of MRPs and find out which algorithm converges faster. We specifically chose MRPs as framework (rather than MDPs), because providing bounds on the rate of convergence of these learning algorithms analytically, becomes difficult very rapidly. The first MRP we studied was a deterministic linear Markov Chain. In this MRP we found that the MC methods converge faster than Q-learning does. We then proceeded on to a similar MRP. The only difference with the previous MRP was that the transition from state $N-1$ to state N was stochastic. In this MRP, we proved that the rate of converges of Q-learning is at least as big as that of MC methods. The final MRP we studied was a symmetric random walk. Here we found (contrary to the previously found results) that Q-learning converges faster. We found this result by means of simulations rather than an analytical derivation. Overall, we conclude that determining the best RL algorithm very much depends on the nature of the MRP. The structure of the MRPs for which Q-learning should be preferred above MC methods and vice versa, remains unclear. Future research could try to resolve this question.

In the second part of the thesis we compared human decision making with artificial decision making. The task we examined was the IGT. The IGT is a famous test that is often used in psychology/neuroscience. Surprisingly, there is no study that compared human decision making with artificial decision making in this particular well-known task. We compared three learning algorithms to the human decision maker. Two of the learning algorithms were bandit algorithms (UCB and Thompson sampling). The other learning algorithm was an RL algorithm (Q-learning). The simulations showed that all the artificial learning agents outperformed the human decision makers. This result shows that artificial learning agents can outperform their human counterparts in this particular decision task. This observation is also something we would expect a priori. In recent years, artificially learning agents were able to outperform humans in complex decision tasks such as chess, poker and the game of Go. Therefore, it would be surprising if the human subjects were able to outperform the AI. When we compared the learning algorithms between themselves we found that the Bayesian approach performed better than the frequentist approach. This is also what we would expect, because the Bayesian approach uses more information in its algorithm than the frequentist approach. The Bayesian approach (Thompson sampling) models the reward distribution as a categorical distribution whereas the frequentist approach models it as an unknown distribution. It is also interesting to note that in psychology participants are in some cases asked to perform the modified IGT. The modified IGT has in contrast to the regular IGT a non-stationary

reward distribution. That is, the reward distribution of the different decks changes over time. It would be interesting to see if the human participants could outperform the AIs on this task. Future research could be devoted to see if this is the case.

In the final part of this thesis we looked at another biomedical application of the bandit algorithms. Inspired by the tremendous performance of Thompson sampling in the IGT we looked at how this algorithm would perform at the task of designing a clinical trial. This task boils down to allocating patients to a treatment. Mathematically speaking the problem can be regarded as an MAB. As we recently discussed, both the frequentist (explore-exploit design) and the Bayesian (Thompson sampling) strategy outperformed the randomization design of a clinical trial. Outperformed here means that more people received an effective treatment while at the same time the researchers were able to establish a significant difference between the effectiveness of the treatments. Does this entail that physicians/biomedical researchers should abandon the randomization strategy (which is considered to be the golden standard) and adopt either the explore-exploit or adaptive Bayesian strategy? Well, not exactly. There are some limitations to the approach we took that we have not yet discussed. We viewed the design of the clinical trial as an MAB. This means that once we have allocated a patient to a treatment we also observe the outcome of the treatment of the patient immediately. This is of course not the case in practice. For some types of medication it could easily take a couple of weeks/months before one can observe the outcome. In these situations one still has to resort to the randomization design. If on the other hand the population size of the patients is not too large (say $N \leq 80$) and the outcome of the treatment can be observed rather quickly (within a couple of hours/days), then one can use the explore-exploit or adaptive Bayesian strategy to design the clinical trial. Future research could for instance focus on constructing bandit algorithms that take the delay of the outcome into account.

In conclusion, we find that RL algorithms together with bandit algorithms can be helpful in solving many real-world problems, and in particular biomedical problems. There are, however, still open questions in the field of RL that need to be addressed. We hope that this thesis will encourage future researchers to start addressing these types of questions.

8 Bibliography

- (1) Singh S., and Dayan P. (1998). Analytical mean squared error curves for temporal difference learning. Machine Learning.
- (2) Beleznay F., Gröbler T., and Szepesvari C. (1999). Comparing value function estimation algorithms in undiscounted problems. Technical Report TR-99-02, Mindmaker, Ltd.
- (3) Komorowski M, Celi L.A., Badawi O., Gordon A.C. and Faisal A.A. (2018). The Artificial Intelligence Clinician learns optimal treatment strategies for sepsis in intensive care. Nature Medicine, Volume 24, Number 11, Page 1716
- (4) Sahba F., Tizhoosh H.R., and Salama M.M. (2008). Application of reinforcement learning for segmentation of transrectal ultrasound images. BMC Med Imaging.
- (5) Purves, D. (2012) Neuroscience. 5th Edition, Sinauer Associates, Inc., Sunderland.
- (6) Glimcher P.W., and Fehr E. (2013). Neuroeconomics: Decision Making and the Brain. Academic Press
- (7) Rangel A., Camerer C., and Montague P.R. (2008). A framework for studying the neurobiology of valuebased decision making. Nat Rev Neurosci. 9(7):545–56
- (8) Balleine B.W., Daw N.D., and O'Doherty J.P. (2008). Multiple forms of value learning and the function of dopamine. Neuroeconomics: decision making and the brain. 36:7–385
- (9) Bechara A. (2004). Disturbances of emotion regulation after focal brain lesions. Int Rev Neurobiol.2004;62:159–193.
- (10) Wallis J.D. (2007). Orbitofrontal Cortex and Its Contribution to Decision-Making. Annu. Rev. Neurosci. 2007. 30:31–56
- (11) Watkins J.C.H., and Dayan P. (1992). Q-learning. Machine Learning 8, 279-292 (1992)
- (12) Sutton R.S., and Barto A.G. (2018). Reinforcement Learning: An Introduction. Second Edition. MIT Press, Cambridge, MA, 2018

- (13) Fridberg D.J., Queller S., Ahn W., Kim W., Bishara A.J., Busemeyer J.R., Porrino L., and Stout J.C. (2010) . Cognitive Mechanisms Underlying Risky Decision-Making in Chronic Cannabis Users. J Math Psychol, 2010. 54(1): p. 28-38.

- (14) Agrawal S., and Goyal N. (2012). Analysis of Thompson sampling for the multi-armed bandit problem. In COLT.

9 Appendix I

Theorem 9.1 (Banach's fixed point theorem)**.** *Let X be a Banach space and $T : X \to X$ a contraction mapping. Then T has a unique fixed point x. Moreover, for any $x_0 \in X$, if $x_{n+1} = Tx_n$ then $x_n \to_{\|\cdot\|} x$. In addition we have that the convergence is geometric,*

$$\|x_n - x\| \leq \gamma^n \|x_0 - x\|$$

where $\gamma \in [0, 1)$.

Theorem 9.2 (Law of large numbers (weak))**.** *Let $X_1, X_2, \ldots$ be a sequence of independent and identically distributed random variables, each having finite mean $\mathbb{E}[X_i] = \mu$. Then, for any $\epsilon > 0$,*

$$\lim_{n\to\infty} \mathbb{P}\Big(\left| \frac{1}{n}(X_1 + \cdots + X_n) - \mu \right| < \epsilon \Big) = 1.$$

Lemma 9.3 (Chebyshev's inequality)**.** *If X is a random variable with finite mean μ and variance σ^2, then for any value $k > 0$,*

$$\mathbb{P}\Big(|X - \mu| \geq k \Big) \leq \frac{\sigma^2}{k^2}.$$

Theorem 9.4 (Hoeffding's inequality)**.** *Let $X_1, \ldots, X_n$ be independent bounded random variables with $X_i \in [a, b]$ for all i. Here $-\infty < a \leq b < \infty$. Let $\bar{X}$ denote the empirical mean (i.e. $\bar{X} = \frac{1}{n}\sum_{i=1}^n X_i$). Then,*

$$\mathbb{P}\Big(\left| \bar{X} - \mathbb{E}[\bar{X}] \right| \geq \epsilon \Big) \leq 2 \exp\Big(- \frac{2n\epsilon^2}{(b-a)^2} \Big).$$

Lemma 9.5 (Hoeffding's lemma)**.** *If X is a random variable with support in $[0, 1]$, then*

$$\mathbb{E}\left[e^{\lambda(X - \mathbb{E}\,X)} \right] \leq e^{\frac{\lambda^2}{8}} \quad \textit{and} \quad \mathbb{E}\left[e^{\lambda(\mathbb{E}\,X - X)} \right] \leq e^{\frac{\lambda^2}{8}}$$

hold for all $\lambda > 0$.

10 Appendix II

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math

#file I
def Q_learning(epsilon, N):
    "Returns the number of trials needed for the value of the first state Q(1)"
    ".to be in a epsilon nghbd of the true Q(1) value. "
    "epsilon = accuracy parameter"
    "N = chain_size = length of the linear markov chain process in which only
    "one action can be performed"
    #Last state N has value zero
    Q_values = np.zeros(N)
    error  = 1 - Q_values[0]
    k=0
    while error > epsilon:
        k+=1
        for s in range(N-2):
            Q_values[s] += 1/k * (Q_values[s+1]-Q_values[s])
        Q_values[N-2] += 1/k*(1-Q_values[N-2] )   #this is the action-value function
        error  = 1 - Q_values[0]
    return(k)


X=np.arange(3,10,1)
empty_list = []
for x in X:
    empty_list.append(Q_learning(0.01,x))

plt.semilogy(X, empty_list, label='Q-learning')
plt.semilogy(X,np.exp((1-0.01)*(X-2)-1),label = 'Estimated bound')
plt.semilogy(X,np.ones(len(X)), label = 'MC')
plt.title('Sample complexity of Q-learning')
plt.xlabel('Chain length (N)')
plt.ylabel('#Trials (k) performed ')
plt.legend()
plt.show()
```

```
#File II

import numpy as np
import matplotlib.pyplot as plt
#Compute RMSE of random walk
#initialization of value function
v_approx = np.zeros(7)
v_approx[1:6] = 0.5
v_approx[6] = 1

#Real value function:
v = np.zeros(7)
for i in range(1,6):
    v[i] = i/6.0
v[6] = 1


actions = [-1,1]
N_state = np.zeros(7)
states = [1,2,3,4,5]
alpha = 0.012

#MC learning algorithm
def MC(walks,gamma):
    'Returns the RMS error after following the Monte Carlo method'
    'Walks represents the number of trials will be performed and gamma is'
    'the discount factor'
    RMSE = [np.sqrt(sum((np.array(v_approx)-np.array(v))**2)/5)]
    for i in range(walks):
        k=0
        path= [3]
        state = 3
        done = False
        reward_walk = 0.0
        while not done:
            state += np.random.choice(actions, p=[float(1/2)]*(2))
            if state == 0:
                done = True
                reward =0.0
            elif state ==6:
                done = True
```

```
                reward =1.0
            else:
                done = False
                reward =0.0
            reward_walk+=reward*gamma**k
            k+=1
            path.append(state)
        for state in states:
            if state in path:
                v_approx[state] += alpha *(reward_walk - v_approx[state])
            else:
                v_approx[state] = v_approx[state]
        RMSE.append(np.sqrt(sum((np.array(v_approx)-np.array(v))**2)/5))
    return(RMSE)

v_approx1 = np.zeros(7)
v_approx1[1:6] = 0.5
v_approx1[6] = 1

def Q_learning(walks,gamma):
    'Returns the RMS error after following the Q-learning algorithm'
    'Walks represents the number of trials will be performed and gamma is'
    'the discount factor'
    RMSE1=[np.sqrt(sum((np.array(v_approx1)-np.array(v))**2)/5)]
    for i in range(walks):
        state =3
        done =False
        while not done:
            new_state = state + np.random.choice(actions, p=[float(1/2)]*(2))
            if new_state == 0:
                done = True
                reward =0.0
            elif new_state ==6:
                done = True
                reward =1.0
            else:
                done = False
                reward =0.0
            v_approx1[state] += alpha*(reward+gamma*v_approx1[new_state]-v_approx1[st
            state = new_state
        RMSE1.append(np.sqrt(sum((np.array(v_approx1)-np.array(v))**2)/5))
    return(RMSE1)

plt.plot(MC(100,0.9),label = 'MC')
```

```
plt.plot(Q_learning(100,0.9), label = 'Q-learning')
plt.title('RMS error of RL algorithms')
plt.xlabel('#Walks')
plt.ylabel('RMSE')
plt.legend()


#file III
Xa = np.array([-50,-100,-150, -200, -250,100])
Xb = [-1150,100]
Xc = [25,-50,0,50]
Xd = [-200,50]
alpha_1 = [1]*6
alpha_2 = [1]*2
alpha_3 = [1]*4
alpha_4 = [1]*2
alpha = []
alpha.append(alpha_1)
alpha.append(alpha_2)
alpha.append(alpha_3)
alpha.append(alpha_4)


def Thompson_sampling():
    'Returns a list of action taken by the agent in the IGT'
    'by following the Thompson Sampling algorithm.'
    actions_taken = []
    Expected_value= np.zeros(4)
    for k in range(95):
        sample_1 = np.random.dirichlet(alpha[0], size=1)
        sample_2 = np.random.dirichlet(alpha[1], size=1)
        sample_3 = np.random.dirichlet(alpha[2], size=1)
        sample_4 = np.random.dirichlet(alpha[3], size=1)
        Expected_value[0] = np.dot(sample_1,Xa)
        Expected_value[1] = np.dot(sample_2,Xb)
        Expected_value[2] = np.dot(sample_3,Xc)
        Expected_value[3] = np.dot(sample_4,Xd)
        best_estimated_arm = np.argmax(Expected_value)
        if best_estimated_arm == 0:
            actions_taken.append(1)
            reward_arm_1 = np.random.choice(Xa,p=[1/10,1/10,1/10,1/10,1/10,1/2])
            if reward_arm_1 == -50:
                alpha[0][0] +=1
            elif reward_arm_1 == -100:
```

```
                alpha[0][1] +=1
            elif reward_arm_1 == -150:
                alpha[0][2] +=1
            elif reward_arm_1 == -200:
                alpha[0][3] +=1
            elif reward_arm_1 == -250:
                alpha[0][4] +=1
            else:
                alpha[0][5] +=1
        elif best_estimated_arm == 1:
            actions_taken.append(2)
            reward_arm_2 = np.random.choice(Xb,p=[1/10,9/10])
            if reward_arm_2 == -1150:
                alpha[1][0] +=1
            else:
                alpha[1][1] +=1
        elif best_estimated_arm ==2:
            actions_taken.append(3)
            reward_arm_3 = np.random.choice(Xc,p=[1/10,1/10,3/10,1/2])
            if reward_arm_3 == 25:
                alpha[2][0] +=1
            elif reward_arm_3 == -50:
                alpha[2][1] +=1
            elif reward_arm_3 == 0:
                alpha[2][2] +=1
            else:
                alpha[2][3] +=1
        else:
            actions_taken.append(4)
            reward_arm_4 = np.random.choice(Xd, p = [1/10,9/10])
            if reward_arm_4 == -200:
                alpha[3][0] +=1
            else:
                alpha[3][1] +=1
    return(actions_taken)


def Q_IGT():
    'Returns a list of action taken by the agent in the IGT'
    'by following the Q-learning algorithm.'
    "Epsilon denotes the probability of taking a random action"
    epsilon = 0.1
    Xa = [-50,-100,-150, -200, -250,100]
    Xb = [-1150,100]
```

```
    Xc = [25,-50,0,50]
    Xd = [-200,50]
    Q_values = np.zeros(4)
    N_values = np.zeros(4)
    t=0
    history_of_actions = []
    action_taken =['argmax','random']
    actions = [0,1,2,3]
    while t < 95:
        first_action = np.random.choice(action_taken, p=[1-epsilon, epsilon])
        if first_action == 'argmax':
            second_action = np.argmax(Q_values)
        else:
            second_action = np.random.choice(actions, p=[1/4]*(4))
        if second_action == 0:
           reward = np.random.choice(Xa,p=[1/10,1/10,1/10,1/10,1/10,1/2])
        elif second_action ==1:
            reward = np.random.choice(Xb,p=[1/10,9/10])
        elif second_action ==2:
            reward = np.random.choice(Xc,p=[1/10,1/10,3/10,1/2])
        else:
            reward = np.random.choice(Xd, p = [1/10,9/10])
        N_values[second_action] += 1
        Q_values[second_action] += (1/N_values[second_action])*(reward - Q_values[sec
        history_of_actions.append(second_action)
        t+=1
    return(history_of_actions)


def UCB_IGT():
    'Returns a list of action taken by the agent in the IGT'
    'by following the UCB algorithm.'
    Xa = [0.88,0.84,0.80,0.76,0.72,1.0]
    Xb = [0.0,1.0]
    Xc = [0.94,0.90,0.92,0.96]
    Xd = [0.76,0.96]
    estimated_sample_mean = [0,0,0,0]
    rewards_deck_1= []
    rewards_deck_2= []
    rewards_deck_3= []
    rewards_deck_4= []
    alpha =  0.01
    "Play each deck once"
    history_of_actions = [0,1,2,3]
```

```
    Ti=np.array([1,1,1,1])

    rewards_deck_1.append(np.random.choice(Xa,p=[1/10,1/10,1/10,1/10,1/10,1/2]))
    rewards_deck_2.append(np.random.choice(Xb,p=[1/10,9/10]))
    rewards_deck_3.append(np.random.choice(Xc,p=[1/10,1/10,3/10,1/2]))
    rewards_deck_4.append(np.random.choice(Xd, p = [1/10,9/10]))
    estimated_sample_mean[0] = np.mean(rewards_deck_1)
    estimated_sample_mean[1] = np.mean(rewards_deck_2)
    estimated_sample_mean[2] = np.mean(rewards_deck_3)
    estimated_sample_mean[3] = np.mean(rewards_deck_4)
    estimated_sample_mean = np.array(estimated_sample_mean)
    for t in list(range(5,96)):
        action_taken = np.argmax(estimated_sample_mean+np.sqrt((alpha*np.log(t))/(2*T
        if action_taken == 0:
           rewards_deck_1.append(np.random.choice(Xa,p=[1/10,1/10,1/10,1/10,1/10,1/2]
           estimated_sample_mean = list(estimated_sample_mean)
           estimated_sample_mean[0] = np.mean(rewards_deck_1)
           estimated_sample_mean = np.array(estimated_sample_mean)
           Ti[0] +=1
        elif action_taken ==1:
            rewards_deck_2.append(np.random.choice(Xb,p=[1/10,9/10]))
            estimated_sample_mean = list(estimated_sample_mean)
            estimated_sample_mean[1] = np.mean(rewards_deck_2)
            estimated_sample_mean = np.array(estimated_sample_mean)
            Ti[1] +=1
        elif action_taken ==2:
            rewards_deck_3.append(np.random.choice(Xc,p=[1/10,1/10,3/10,1/2]))
            estimated_sample_mean = list(estimated_sample_mean)
            estimated_sample_mean[2] = np.mean(rewards_deck_3)
            estimated_sample_mean = np.array(estimated_sample_mean)
            Ti[2] +=1
        else:
            rewards_deck_4.append(np.random.choice(Xd, p = [1/10,9/10]))
            estimated_sample_mean = list(estimated_sample_mean)
            estimated_sample_mean[3] = np.mean(rewards_deck_4)
            estimated_sample_mean = np.array(estimated_sample_mean)
            Ti[3] +=1
        history_of_actions.append(action_taken)
    return(history_of_actions)

df_action = pd.read_csv('choice_95.csv',index_col=None)

def make_df(algorithm):
    'Makes a dataframe (15x95) of the actions taken by the agent '
```

```
    'that follows a learning algorithm'
    list1 = []
    for i in range(15):
        list1.append(algorithm())
        df1=pd.DataFrame(list1)
    return(df1)

def compute_prop_adv1(df):
    for i in range(15):
        empty_list = []
        for j in list(range(95)):
            if df.iloc[i,j] == 0 or df.iloc[i,j] == 1:
                empty_list.append(0)
            else:
                empty_list.append(1)
    prop_adv_deck=sum(empty_list)/len(empty_list)
    return(prop_adv_deck)


def compute_prop_adv2(df):
    for i in range(15):
        empty_list = []
        for j in list(range(95)):
            if df.iloc[i,j] == 1 or df.iloc[i,j] == 2:
                empty_list.append(0)
            else:
                empty_list.append(1)
    prop_adv_deck=sum(empty_list)/len(empty_list)
    return(prop_adv_deck)



def simulate1(algorithm):
    empty_list = []
    for k in range(100):
        empty_list.append(compute_prop_adv1(make_df(algorithm)))
    return(np.mean(empty_list),empty_list)

def simulate2():
    empty_list = []
    for k in range(100):
        empty_list.append(compute_prop_adv2(make_df(Thompson_sampling)))
    return(np.mean(empty_list),empty_list)
```

```
def average(list1):
    empty_list = []
    for k in range(1,101):
        empty_list.append(np.mean(list1[:k]))
    return(empty_list)

TS = simulate2()
UCB =simulate1(UCB_IGT)
Q = simulate1(Q_IGT)
TS[1].insert(0,0)
UCB[1].insert(0,0)
Q[1].insert(0,0)

plt.plot(average(TS[1]),label = 'TS')
plt.plot(average(UCB[1]),label='UCB')
plt.plot(average(Q[1]),label='Q-learning')
plt.title('The average proportion of optimal action over 100 simulations')
plt.xlabel('# Simulation')
plt.ylabel('% optimal action')
plt.legend()

#File IV


def Thompson_sampling(theta1,theta2,N):
    'Performs Thompson sampling on a two-armed Bernoulli bandit problem'
    S1 = 0; F1 = 0
    S2 = 0; F2 = 0
    alpha = 1.0; beta = 1.0
    actions_taken = []
    for k in range(N):
        p1 = np.random.beta(S1 + alpha, F1 + beta)
        p2 = np.random.beta(S2 + alpha, F2 + beta)
        if p1 > p2:
            actions_taken.append(1)
            reward_arm_1 = np.random.binomial(n=1,p=theta1)
            if reward_arm_1 ==1:
                S1+=1
            else:
                F1+=1
        else:
            actions_taken.append(2)
            reward_arm_2 = np.random.binomial(n=1,p=theta2)
            if reward_arm_2 ==1:
```

```
                S2+=1
            else:
                F2+=1
    return(actions_taken,S1,F1,S2,F2)


def exploitation_exploration(k,theta1,theta2,N):
    'Performs the exploration-exploitation algorithm on a two-armed Bernoulli bandit'
    'problem'
    actions_taken = [1,2]*k
    a = np.random.binomial(n=1,p=theta1,size=k)
    b = np.random.binomial(n=1,p=theta2,size=k)
    a=list(a)
    b=list(b)
    empirical_average_arm1 = np.mean(a)
    empirical_average_arm2 = np.mean(b)
    if empirical_average_arm1 > empirical_average_arm2:
        actions_taken.extend([1]*(N-2*k))
        c = np.random.binomial(n=1,p=theta1,size=(N-2*k))
        a.extend(c)
    else:
        actions_taken.extend([2]*(N-2*k))
        d = np.random.binomial(n=1,p=theta2,size=(N-2*k))
        b.extend(d)
    a = np.array(a)
    b = np.array(b)
    return(actions_taken,(a == 1).sum(),(a == 0).sum(),(b == 1).sum(),(b == 0).sum())

def randomization(theta1,theta2,N):
    'Follows the randomization design for a two-armed Bernoulli bandit problem'
    actions_taken = []
    S1 =0; F1=0
    S2=0;  F2=0
    for k in range(N):
        action = np.random.choice([1,2], p=[1/2,1/2])
        actions_taken.append(action)
        if action == 1:
            reward = np.random.binomial(n=1,p=theta1)
            if reward ==1:
                S1+=1
            else:
                F1+=1
        else:
            reward = np.random.binomial(n=1,p=theta2)
```

```
            if reward ==1:
                S2+=1
            else:
                F2+=1
    return(actions_taken,S1,F1,S2,F2)



def compute_prop_healthy():
    'Computes the proportion of patients that received an effective treatment'
    'the local variable, A, can changed to any bandit algorithm '
    empty_list = []
    for k in range(100000):
        A = Thompson_sampling(0.85,0.15,50)
        empty_list.append((A[1]+A[3])/len(A[0]))
    average=np.mean(empty_list)
    return(average)


def compute_chi_sq():
    'Computes the proportion of rejected null hypotheses.'
    'The local variable, A, can changed to any bandit algorithm.'
    empty_list = []
    list1 = []
    for k in range(100000):
        A = Thompson_sampling(0.85,0.15,30)
        b=np.zeros((2,2))
        b[0][0] = A[1]
        b[0][1] = A[2]
        b[1][0] = A[3]
        b[1][1] = A[4]
        pvalue = stats.chi2_contingency(b)[1]
        if pvalue < 0.05:
            list1.append(pvalue)
            empty_list.append(True)
        else:
            empty_list.append(False)
    prop_rejected_pvalue = sum(empty_list)/len(empty_list)
    return(prop_rejected_pvalue)
```