UTRECHT UNIVERSITY

MASTER THESIS

Applying reinforcement learning to argument-based inquiry

Author:
Nienke Uijlen

Supervisor:
dr. G.A.W. (Gerard) Vreeswijk
dr. B.J.G (Bas) Testerink

Intelligent Systems group
Department of Information and Computing Sciences

May 23, 2019

Contents

Chapter 1

Introduction

Nowadays, life without internet is unimaginable. More and more we rely on the internet for activities that we would previously have realised offline. Shopping is no exception: an increasing percentage of purchases is bought online, often fairly anonymously. Criminals see this as an opportunity to mislead people online, resulting in a continuously increasing number of online fraud cases.

Currently, the Netherlands National Police receives roughly 40,000 complaints about online fraud every year (Bex, Peters, and Testerink, 2016). These complaints cover several types of online fraud, of which the most common one is probably the type of fraud that is committed via second-hand auction websites such as marktplaats.nl and ebay.com. Another example of online fraud is the kind where fake webshops are used to deceive people. Often, these webshops try to imitate an existing webshop. Criminals then create a website almost identical to an existing one, hoping that people will not notice that it is fake. In other cases, the fake webshop is not an imitation of a real webshop, but looks like an ordinary webshop. Often, on these websites a lot of (fake) contact information can be found, which makes them appear trustworthy.

Victims of the above described or other types of online fraud can report the fraud at LMIO (Landelijk Meldpunt Internetoplichting), which is the section of the National Police that focusses on online fraud. One possibility to report the crime is at the police station. Victims can then state what happened to a police officer, who can ask further questions to make sure the information is complete enough to determine whether the case is a true case of fraud. However, because of the high volume and relatively low detail (when compared to, for example, burglary) of such cases, they are very suitable for online complaints and further automated processing (Bex, Peters, and Testerink, 2016). Therefore, the Netherlands National Police, in collaboration with the UU Intelligent Systems group, has recently designed an online interface in which citizens can file complaints about online fraud cases. The interface contains a form where citizens must fill out some basic information and where they can explain what happened. After the complainant completed and submitted the form, the information is processed by a police officer. Often, the information turns out to be incomplete. This is because usually the complainant does not know exactly which information the police needs to determine whether the complaint forms a sufficient basis for further investigation. The complainant must then be contacted to obtain the missing information. Since waiting for the response of the complainant can take a long time, and often the complainant needs to be contacted multiple times, this method of processing online filed complaints is inefficient.

The processing of online filed complaints is currently being automated to make it more efficient. A software agent processes the complaint immediately after it is submitted. It can then determine whether the complaint is complete and, if not, ask the complainant questions through the interface, such that the complainant can answer right away. The process of submitting a complaint then becomes a dialogue between the agent and complainant. This way, the time it takes to wait for the complainant's response is minimised. However, a difficulty in this method is that often the agent has a large number of questions to ask. The order of the questions influences the length of the dialogue, since some questions are more logical to ask than others. Humans do not think about which questions are logical explicitly, due to their common sense. When the complainant ordered a product at a fake webshop for example, he or she probably did not have physical contact with the counterparty. Humans then intuitively know that it is logical and efficient to ask the question whether physical contact took place late in the dialogue. The agent has no common sense and therefore does not know these nuances intuitively. The dialogue can then become long and inefficient, which is undesirable. Therefore, it is important that the agent learns which questions to ask in certain situations. Since every complaint is different and therefore a lot of different situations are possible, learning the right questions for every situation is a difficult task. This task can be modelled as a Markov Decision Process (MDP). The solution to the task can then be found by solving the MDP.

In general, solving an MDP can be done through a variety of methods. Two of the most common ones are dynamic programming and Q-learning, which we both apply in this thesis. However, the state space of the MDP for the task described above is very large. Dynamic programming and Q-learning are then not applicable anymore, due to their large amount of required memory and processing time. A possible solution to this problem is to use a variant of Q-learning, which represents Q-values by an approximation function rather than saving them in a Q-table. Ormoneit and Sen (2002) developed kernel-based reinforcement learning, in which every iteration an extra kernel, or basis function, is added to the approximation function. Furthermore, Riedmiller (2005) and Van Hasselt, Guez, and Silver (2016) proposed a variant of Q-learning where the Q-values are represented by a neural network.

In this thesis, we apply three different methods to solve the MDP corresponding to the task of learning the right questions during the intake of a complaint. The first two are Q-learning and dynamic programming, which are, as explained above, practically inapplicable when the state space of the MDP becomes too large. However, it is proved that, under certain circumstances, both methods provide an optimal solution. Therefore they can be used as a comparison for the third method we discuss in this thesis: a Q-learning algorithm that uses a neural network to represent Q-values, which we call neural fitted Q iteration (NFQ). The aim of this thesis is to answer the question whether the NFQ algorithm could be a solution to the problem of the large state space.

From experiments, it follows that NFQ's performance is comparable to that of Q-learning. Moreover, the amount of required memory and processing time is significantly reduced by this method, when comparing it to traditional Q-learning. Therefore we can conclude that the NFQ algorithm could indeed be a solution to the problem of the large state space of the MDP corresponding to the task of learning the right questions during the intake of a complaint.

The structure of this thesis is as follows. In Chapter 2, we first discuss the automated intake process of complaints at LMIO and the problems that are faced during this process in more detail. Chapter 3 provides background information about reinforcement learning and in particular Q-learning. In Chapters 4, 5 and 6 we present Q-learning, dynamic programming and neural fitted Q iteration applied to argument-based inquiry. In Chapter 7, we test the validity and efficiency of the presented methods. Finally, we explore related work in current literature in Chapter 8.

Chapter 2

Processing complaints at LMIO

At LMIO, complaints about online fraud cases are often filed in an online interface that the Netherlands National Police has recently designed in collaboration with the UU Intelligent Systems group. During the processing of these complaints, the following steps are taken.

First, during the intake of the complaint, it is investigated whether the complaint forms a sufficient basis for further investigation. This is only the case when, based on the information the complainant provides, there are reasons to believe that the complainant is indeed misled. When further investigation turns out to be needed, police officers try to determine whether it is likely that the counterparty misled the complainant deliberately. If that is the case, the next step is to decide whether the counterparty should be arrested. This decision is made by the Public Prosecution Service (Dutch: Openbaar Ministerie, OM). A possible arrest is followed by an interrogation, where it is determined whether the case is a true case of fraud, and possibly by a prosecution.

In this thesis, we focus on the first step of the above described process, namely the intake of a complaint. This step is being automated, to make it more efficient. In this chapter, we first globally describe how the intake of complaints that are filed at LMIO currently proceeds. We then explain the most important and interesting steps in this intake process in more detail.

2.1 The pipeline for the intake process of complaints

The intake process of a complaint can be seen as a pipeline where the input is a filed complaint and the output is the conclusion whether the complaint forms a sufficient basis for further investigation. The first step in this process is to convert the complaint to a text file (an XML file) that contains the information of the complaint in a structured manner. From this text file, it is determined which *observables* are present in the complaint and therefore what has been observed. By observables, we mean everything that can be observed and that can be used to build arguments, such as "something has been delivered" or "the contact took place via marktplaats.nl".

The observables that are found in the complaint are used by an argumentation agent in an argumentation dialogue, which we will explain in more detail in Section 2.2. In this dialogue, the agent tries to build arguments for its *topic*. In this context, a topic is a question for which we try to find a decisive answer. When considering fraud detection at LMIO, the topic of the agent is almost always the question whether or not the complaint is worth further investigation. The agent's goal is to *stabilise* its argumentation with respect to its topic. A stabilised argumentation means that the opinion of the agent about the topic will not change anymore when more information becomes available. In Section 2.2, we will elaborate on when an argumentation is stabilised in this context.

Sometimes the agent is unable to stabilise its argumentation with respect to its topic, due to a lack of information. If that is the case, it is necessary to obtain more information by asking the complainant questions. Currently, this is done manually by employees of the Netherlands National Police, but this process is intended to be automated. The agent will then be asking questions to obtain the right information, that is the information that it needs to make its argumentation stable. This is done by performing an *argument-based inquiry*. To make this inquiry as efficient as possible, the agent needs to learn which questions to ask in certain situations. In Section 2.3, we will look at this inquiry in more detail.

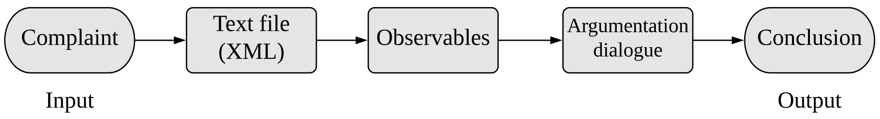


FIGURE 2.1: The pipeline of processing a complaint.

When the agent has formed a stable argumentation with respect to its topic, the end of the pipeline is reached. The conclusion is then sent to an employee who verifies the correctness of the agent's conclusion. In the future, this check may no longer be necessary.

2.2 Argumentation dialogue

As explained in Section 2.1, the observables that are observed in a complaint are used by the agent in an argumentation dialogue. During this argumentation dialogue, certain concepts of the ASPIC+ framework (Prakken, 2010) are used. That is, the agent has access to a set of rules and a *knowledge base* consisting of observations, which it can use together to build arguments for and against its topic. The observations in the knowledge base are always observables that the agent obtains from a complaint. These observables form a subset of the set of all observables. This fixed set is made up manually, and contains all observables that could be relevant for detecting fraud. Each observable o can be present in a complaint C in two different forms: positive, which we denote by $o \in C$ and which means that o is true in that complaint, or negative, denoted by $\neg o \in C$, which states that o is not true. Note that there is a difference between a complaint that contains the observable $\neg o$, and one that contains neither o nor $\neg o$. In the first situation, the agent knows for sure that o is not true, while in the second case it has no information about o.

In addition to the observables from the complaint, the agent uses rules to build arguments in order to obtain new information. An example of such a rule is that if a buyer has paid and the product has not been sent, then the seller has probably committed fraud. This rule can be expressed in a more mathematical way by using the following notation: "paid, $\neg$sent $\Rightarrow$ fraud". We call the observables on the left-hand side the *antecedents* of the rule and the one on the right-hand side is called the *consequent*. Another example of a rule is "waited, ¬received ⇒ ¬sent", which states that if the buyer has waited a reasonable amount of time and didn't receive anything, the seller probably did not send the product. In general, when the complaint contains observables o_1 and $\neg o_2$, and there exists a rule "$o_1, \neg o_2 \Rightarrow o_3$, then the agent can use the rule to build an argument for o_3 and therefore add o_3 to its knowledge base. Conclusion o_3 can then also be used to build arguments for other statements. The rules that are used in the argumentation dialogue of the agent are obtained through knowledge acquisition with experts. They model laws, regulations and the grounding of judicial terms with real-world observations.

In the argumentation dialogue, the agent uses the rules and available observables to build arguments in the way explained above. When it has built all possible arguments, a graph that shows the arguments and their *attack relations* is created. For these attack relations, we say that arguments A_1 and A_2 attack each other when their conclusions are contradictory (for example o and $\neg o$). Furthermore, A_1 attacks A_2 one-sided when A_1's conclusion contradicts a premise of A_2. An important exception to these attacks is that premises or conclusions that are in the agents' knowledge base at the start of the argumentation cannot be attacked. We will now provide an abstract example to illustrate how arguments and their attack relations can be found.

Example 2.2.1. Suppose that the knowledge base of the agent is $\{a, \neg b, c, d\}$ and that its set of rules is:

1. $\neg b \Rightarrow \neg e$
2. $\neg b, c \Rightarrow \neg a$
3. $\neg a, d \Rightarrow e$

The arguments that can be built with these rules and known observables and their attack relations are shown in Figure 2.2. As can be seen, A_5 and A_7 attack each other, since their conclusions ($\neg e$ and e) are contradictory. Furthermore, A_1 attacks A_6 on its conclusion and A_7 on one of its premises. Note that, while the conclusion of A_6 contradicts that of A_1, A_6 does not attack A_1. This is because the conclusion of A_1, which is a, is in the agents' knowledge base and can therefore never be attacked. ◇

$A_1 : \varnothing \Rightarrow a$
$A_2 : \varnothing \Rightarrow \neg b$
$A_3 : \varnothing \Rightarrow c$
$A_4 : \varnothing \Rightarrow d$
$A_5 : A_2 \Rightarrow \neg e$ (rule 1)
$A_6 : A_2, A_3 \Rightarrow \neg a$ (rule 2)
$A_7 : A_4, A_6 \Rightarrow e$ (rule 3)

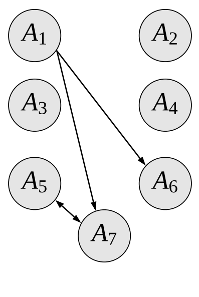


FIGURE 2.2: The arguments that can be built with the rules and knowledge base of example 2.2.1 (left) and a graphical representation of their attack relations (right). In arguments A_1, A_2, A_3 and A_4, $\varnothing$ is used as left-hand side to point out that their conclusions are observed facts rather than concluded from other observables and rules.

The graph with attack relations is used to determine which arguments are *'in'* and which are *'out'*, in the following sense:

- An argument is 'in' if all arguments that attack it are 'out'. Note that an argument with no incoming attacks is always 'in'.
- An argument is 'out' if at least one of the arguments that attack it is 'in'.

When we use this notion of 'in' and 'out' to examine the arguments of Example 2.2.1, we obtain a coloured graph, which is shown in Figure 2.3. In this graph, green arguments are 'in' and red arguments are 'out'.

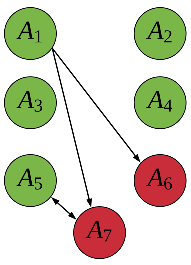


FIGURE 2.3: The coloured graph for the arguments of Example 2.2.1. Green arguments are 'in' and red arguments are 'out'.

Subsequently, the agent verifies whether its argumentation with respect to its topic is stable, or, in other words, whether additional information would not change its opinion about its topic. This is the case if one of the following four situations occurs (Testerink and Bex, 2019):

1. There exists no argument with the topic as conclusion and even with extra information, such an argument is impossible to build. This situation is called the *unsatisfiable* case.
2. There exists an argument with the topic as conclusion that is 'in' and that will always stay 'in', even when more information is added. We call this the *defended* case.
3. There exists at least one argument with the topic as conclusion, but it is (or they are) not 'in' and will never become 'in' when additional information becomes available. This situation is divided in two cases:
    - The *out* case: in this situation, all arguments for the topic are attacked by an argument that is 'in'.
    - The *blocked* case: here, at least one argument for the topic is not attacked by an argument that is 'in'.

Since recognising a stabilised argumentation is a complex task, a heuristic is used to predict this. This heuristic tells, when given a set of observables, whether it is possible to build a stabilised argumentation with respect to the topic with these observables or not.

When the agent has stabilised its argumentation, the intake process of the complaint is completed. Depending on which of the above three situations occurred, its conclusion will be that there is possibly a case of fraud (situation 1) or that fraud has not been proven (situation 2 or 3). However, when the agent is unable to stabilise its argumentation with the currently available observables, more information is needed to continue the process.

2.3 Argument-based inquiry

Sometimes the argumentation agent does not have enough information to stabilise its argumentation with respect to its topic. A reason for this can be that the complaint is incomplete and does not contain enough relevant observations. In that case, more information is needed. One possibility is that this information is gathered from the complainant. Currently, an agent is under development that will be used to gain this information automatically through an argument-based inquiry. In this inquiry, the agent can always choose from a fixed set of questions. To each question, a number of observables are linked that could be learned by asking that particular question. The agent determines which observables it is missing, based on the argumentation dialogue. Subsequently, it searches its set of questions for relevant ones (that is, for questions that could yield missing observables). Often, multiple questions turn out to be relevant for the agent. This can be because the agent still needs several observables, or because a certain observable can be obtained with multiple questions. In such situations, the agent must choose one of the relevant questions. The question is then how the agent should make this choice.

Since a long dialogue between agent and user is undesirable, the agent should attempt to gather the information it needs as soon as possible, so by asking as few questions as possible. For this purpose, the agent must learn to ask the right questions at the right moment. The goal is to find the optimal question for each possible set of observables. Then the agent knows which question it should ask in every situation it can encounter during the inquiry. However, the number of different sets of observables is very large. Recall from Section 2.2 that every observable can be absent, or present in two ways (positive or negative). Therefore the number of different sets is 3^n, where n is the number of observables. Because in the case of fraud detection the number of observables is around 20, a few billion different sets of observables are possible. It is therefore not trivial to find the optimal question for each set. In this thesis, we will explore three different methods to address this problem. In Chapter 4, we will show how reinforcement learning, especially Q-learning, can be applied to argument-based inquiry, or in other words, how it can be used to teach the agent to ask the right questions. Then, in Chapter 5, another method is shown, namely dynamic programming. Lastly, in Chapter 6 we will consider a variant of Q-learning, called neural fitted Q iteration. Before diving into these three methods, we start by discussing the most important concepts of reinforcement learning in Chapter 3.

Chapter 3

Reinforcement learning and Markov decision processes

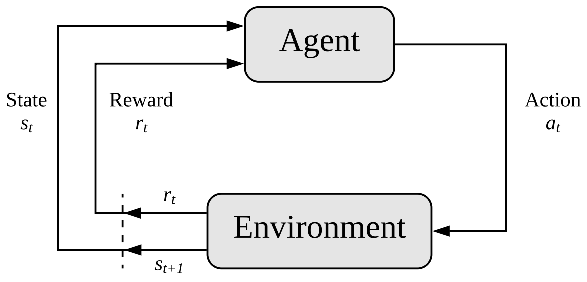
In this chapter, we provide an introduction to reinforcement learning and Markov decision processes (MDPs), based on Sutton and Barto (1998). Readers that are already familiar with these concepts may skip this chapter.

3.1 The agent and the environment

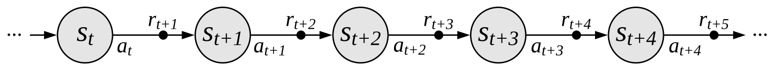
Reinforcement learning is a way of learning from interaction to achieve a goal. The learner is called the *agent*. The agent is a decision-maker and is able to select actions, based on what *state* it is in. Everything outside the agent is called the *environment*. The agent and the environment interact continually in the following way.

In reinforcement learning, S denotes the set of all possible states that the agent can be in and $A(s_t)$ is the set of all actions that it can select when it is in state s_t. At each step of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$, the agent and the environment interact. At every time step t, the agent receives its state $s_t \in S$ from the environment. Based on this state, it selects an action $a_t \in A(s_t)$. This action changes the environment, such that one time step later, the agent is in a new state s_{t+1}. In addition, the agent receives a numerical reward $r_t \in \mathbb{R}$ as a consequence of its action. With this reward, the agent can update its *policy* for choosing actions. A policy π is a function from S and A to probabilities of choosing an action when in a specific state. In other words, the policy of an agent describes in each state with what probability the agent should choose available actions to achieve its goal. Roughly speaking, the agent's goal is to maximise the total amount of reward it receives in the long run. A schematic representation of the process of interaction between the agent and the environment can be found in Figure 3.1a. Figure 3.1b schematically shows the movement of the agent through states for a number of time steps. To make the concept of reinforcement learning more clear, we now provide a simple example of a situation in which reinforcement learning can be applied.

Example 3.1.1. *The fraud detection robot*
Consider a robot that has the job to find out whether a filed complaint is (possibly) a case of fraud or not. In order to achieve this goal, the robot can ask the person that filed the complaint, the complainant, questions about what happened. For illustrative purposes, a very simplified situation is used in this example. Suppose the robot can only choose among two different questions, namely 1) "Did you Pay the



(A) A schematic representation of the interaction between the agent and the environment. The agent observes the current state (s_t) and chooses an action (a_t) based on that state. Then it moves to the next state (s_{t+1}) and receives a reward (r_{t+1}) from the environment, which it uses to update its policy.



(B) A schematic representation of the movement of the agent through states for a number of time steps.

FIGURE 3.1: Schematic representations of how the agent moves (b) and interacts with the environment (a).

agreed amount of money?", denoted by P, and 2) "Has anything been Delivered?", denoted by D. The robot's task in this situation is to learn a strategy that minimises the expected amount of questions it has to ask before it is able to draw a conclusion about the complaint. To teach the robot such an optimal strategy, a reinforcement learning agent can be used.

This reinforcement learning agent will be used to make decisions about which of the two questions the robot should ask, on the basis of the information that the robot already knows. This information can be empty (denoted by $\emptyset$), or one of the following: the complainant has paid ($\{p\}$), the complainant has not paid ($\{\neg p\}$), something has been delivered ($\{d\}$), nothing has been delivered ($\{\neg d\}$), or a combination of these $\Big(\{p,d\},\{p,\neg d\},\{\neg p,d\},\{\neg p,\neg d\}\Big)$. The agent therefore has nine states. When the robot's information is empty, both questions are relevant to ask. Therefore, if the reinforcement learning agent is in state $\emptyset$, its possible actions are P and D. However, when the robot already knows the answer to P, only question D is relevant, and vice versa. Once the robot knows the answers to both P and D, asking a question will never yield additional information and is therefore never rewarding, such that in this example

$$S = \Big\{\emptyset, \{p\}, \{\neg p\}, \{d\}, \{\neg d\}, \{p,d\}, \{p,\neg d\}, \{\neg p,d\}, \{\neg p,\neg d\}\Big\},$$

$$A(\emptyset) = \{P, D\},$$

$$A(\{p\}) = A(\{\neg p\}) = \{D\},$$

$$A(\{d\}) = A(\{\neg d\}) = \{P\}.$$

In order to let the agent learn, we must define rewards. This can be done in multiple ways, but one possibility is the following: the reward of asking a question is always negative, since the agent should learn to ask as few questions as possible. Whenever the agent has enough information to draw a conclusion about the complaint, it receives an additional positive reward. In Example 3.3.1 we will elaborate on these rewards. In this way, the agent will eventually update its policy such that it will try to find a conclusion as soon as possible. $\diamondsuit$

One comment should be made about Example 3.1.1: we did not explain how the agent updates its policy and thus how the agent actually learns. The reason for this is that the way of updating the policy differs among different reinforcement learning methods. In section 3.5, we will explain one of these methods, namely Q-learning.

3.2 Rewards

In Section 3.1, we already described the goal of a reinforcement learning agent, namely to maximise its 'reward in the long run'. This is a rather imprecise definition that we formalise in this section.

Suppose the sequence of rewards received after time step t is denoted by r_t, r_{t+1}, r_{t+2}, ... We denote the total reward from time t by R_t. The agent's goal is to maximise the expected total reward in every time step t. In the simplest case the reward from time t is just the sum of all future rewards:

$$R_t = r_t + r_{t+1} + r_{t+2} + ... + r_{T-1},$$

where T is the final time step. However, this is not the function that is usually used, since many reinforcement problems do not necessarily have a final time step. In that case we would have $T = \infty$ and therefore the total reward, which the agent is trying to maximise, could become infinite as well. To cover this problem, we need the concept of discounting. The idea of this concept is that rewards received later are worth less than those received sooner. The function for the total reward then becomes:

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k},$$

where $0 \leq \gamma \leq 1$ is called the *discount rate*. Using this discount rate ensures that R_t is finite, even if the sequence of individual rewards is infinite. A proof for this is given in Watkins and Dayan (1992). Informally, the discount rate determines the present value of future rewards. That is, a reward received k time steps in the future is worth only γ^{k-1} times the value it would have if it was received immediately.

3.3 Markov decision processes

Reinforcement learning problems are often modelled in a structured way as a *Markov decision process*. This is a way to represent the states of the reinforcement learning problem and the transitions between the states. In this section, we explain these Markov decision processes.

Before we show how reinforcement learning tasks can be modelled as Markov decision processes, we should first explain the *Markov property* of a process. Informally, this property means that in a process, the next step is only dependent of the current state and independent of the past. For example, the process of the fraud detection robot from Example 3.1.1 has the Markov property: no information of the past is required, since the only information that the agent uses to select a question is that of the current state (what information is already known).

In the context of reinforcement learning problems, the Markov property can be seen as follows: a reinforcement learning task has the Markov property if the probability that the agent moves to a state s' when it was in state s and chose action a is independent of the history. Hence, the transition probabilities between states should be the same throughout the entire process. The same holds for the probabilities of getting reward r when choosing action a in state s. When we define this property mathematically, we get:

$$Pr(s_{t+1} = s', r_t = r | s_t, a_t, r_{t-1}, s_{t-1}, a_{t-1}, ..., r_0, s_0, a_0) = Pr(s_{t+1} = s', r_t = r | s_t, a_t).$$

If a reinforcement learning task satisfies this Markov property, it is called a Markov decision process (MDP). Usually, the set of states and the set of actions are finite, in which case it is called a finite MDP. Any finite MDP is defined by its set of states, its sets of actions and two quantities:

- The probability that the agent moves to state s' when it performs action a in state s: $\mathcal{P}^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a)$. This is sometimes called the 'transition function', since it determines the probabilities of which transitions occur.
- The reward that the agent receives when it moves from state s to state s' by performing action a, denoted by $\mathcal{R}^a_{ss'}$.

These quantities specify the most important aspects of the dynamics of a finite MDP.

To make the concept of MDPs more clear, we now model the reinforcement learning task of the fraud detection robot from Example 3.1.1 as an MDP.

Example 3.3.1. *The fraud detection robot continued*
Recall that the reinforcement learning agent in this example makes decisions about which question the robot should ask, based on the current information that the robot has. We saw that the set of states and the sets of actions were as follows:

$$S = \Big\{\emptyset, \{p\}, \{\neg p\}, \{d\}, \{\neg d\}, \{p, d\}, \{p, \neg d\}, \{\neg p, d\}, \{\neg p, \neg d\}\Big\},$$

$$\begin{aligned} A(\emptyset) &= \{P, D\}, \\ A(\{p\}) = A(\{\neg p\}) &= \{D\}, \\ A(\{d\}) = A(\{\neg d\}) &= \{P\}. \end{aligned}$$

To define the MDP for this task, we now only need to specify the quantities $\mathcal{P}^a_{s,s'}$ and $\mathcal{R}^a_{s,s'}$ for all s, a, s'. Let us start with the rewards $\mathcal{R}^a_{s,s'}$: the reward for asking a question is always negative, let us say -1. This ensures that the agent will try to ask as few questions as possible, to keep the dialogue short. Whenever the agent arrives in a state in which it has enough information to draw a conclusion about the complaint, the dialogue is finished. We call these states *final states*. Since the agent's goal is to draw a conclusion, it should be rewarded when reaching such a final state. Therefore it receives an additional reward of $+3$ when it arrives in a final state, such that its total reward for that time step is $+2$. All that remains now is to specify which states are final, or in other words, in which states the agent has enough information to draw a conclusion.

In this example, we assume the following rules, which are of course a simplification of reality:

1. "When something has been delivered, no fraud has been committed". This rule can be written down more mathematically as "$d \Rightarrow \neg f$", where d stands for "something has been delivered" and f for "it is not a case of fraud".
2. "When the agreed amount of money has been paid and nothing has been delivered, we consider it a case of fraud." This can be notated as "$p, \neg d \Rightarrow f$".

With these rules, three different conclusions are possible. The first conclusion is "The case is a case of fraud". Because of rule number 2, this conclusion can be drawn when the agent knows that there has been paid but not delivered. Or, in other words, when the agent arrives in state $\{p, \neg d\}$. Therefore $\{p, \neg d\}$ is a final state. Another conclusion is "The case is not a case of fraud". According to rule number 1, this can be concluded when the agent knows that something has been delivered (d), thus when it is in one of the (final) states $\{d\}$, $\{p, d\}$ or $\{\neg p, d\}$. The third conclusion is "It is impossible to tell if the case is a case of fraud". This conclusion is drawn when all possible information about the case is known, but none of the two above rules apply. This is the case if the agent arrives in state $\{\neg p, \neg d\}$, which is therefore also a final state.

For the transition probabilities $\mathcal{P}^a_{ss'}$, the following holds: when the robot's information is empty (which is the case at the beginning), the agent has two possible questions to choose from. When it chooses to ask P, it will get an answer to the question "Did you pay the agreed amount of money?". Therefore it will either end up in state $\{p\}$ or in state $\{\neg p\}$. In this example, we assume the probability that it moves to state $\{p\}$ when asking P in state $\emptyset$, denoted as $\mathcal{P}^P_{\emptyset,\{p\}}$, is 0.7 and that $\mathcal{P}^P_{\emptyset,\{\neg p\}} = 0.3$. When the agent instead chooses to ask D, "Has anything been delivered?", in state $\emptyset$, it will move to state $\{d\}$ or $\{\neg d\}$ with the following probabilities: $\mathcal{P}^D_{\emptyset,\{d\}} = 0.4$ and $\mathcal{P}^D_{\emptyset,\{\neg d\}} = 0.6$. When in state $\{p\}$ or $\{\neg p\}$, the agent can only choose action D. We assume that starting in state $\{p\}$, it will end up in state $\{p, d\}$ with probability 0.43 and in state $\{p, \neg d\}$ with probability 0.57. When it is in state $\{\neg p\}$, the probabilities that it moves to state $\{\neg p, d\}$ or $\{\neg p, \neg d\}$ are 0.33 and 0.67 respectively. If the agent asks P in state $\{d\}$, its next state will be $\{p, d\}$ with probability 0.75 and $\{\neg p, d\}$ with probability 0.25. However, since we already saw that state $\{d\}$ is a final state, we can consider these probabilities to be 0, because the process ends in that state. Finally, the probabilities of moving to states $\{p, \neg d\}$ and $\{\neg p, \neg d\}$ when starting in state $\{\neg d\}$ are 0.67 and 0.33 respectively.

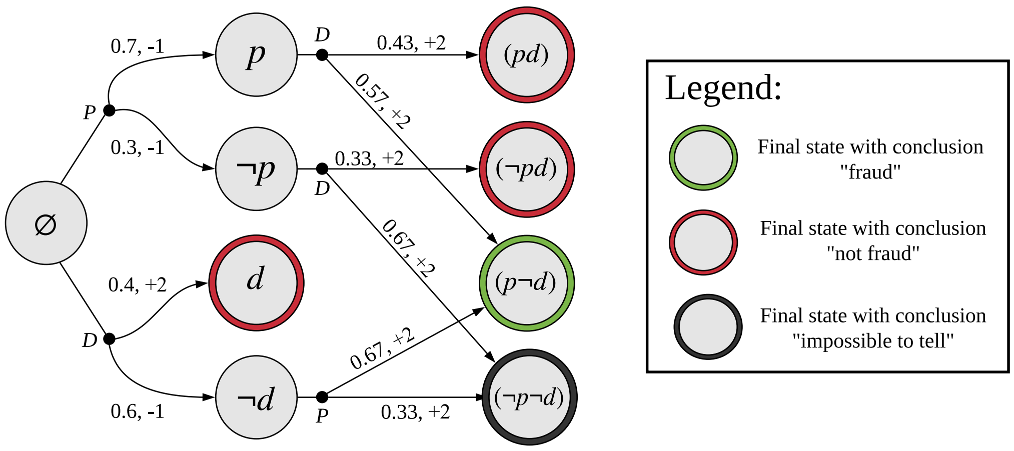
We have now specified $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ for all s, a, s', which determine the MDP. We can write down the transition probabilities and rewards in a table, like in Table 3.1.

Another, more visual way to summarise the values from Table 3.1 is in a *transition graph*. In such a graph, there are two kind of nodes: *state nodes* for each possible state s and *action nodes* for each state-action pair (s, a). One should read the graph as follows: starting in state s and performing action a moves you along the line from state s to the action node (s, a). Every arrow corresponds to a triple (s, s', a) and is

$s = s_t$	$a = a_t$	$s' = s_{t+1}$	$\mathcal{P}^a_{ss'}$	$\mathcal{R}^a_{ss'}$
$\varnothing$	P	$\{p\}$	0.7	-1
$\varnothing$	P	$\{\neg p\}$	0.3	-1
$\varnothing$	D	$\{d\}$	0.4	$+2$
$\varnothing$	D	$\{\neg d\}$	0.6	-1
$\{p\}$	D	$\{p, d\}$	0.43	$+2$
$\{p\}$	D	$\{p, \neg d\}$	0.57	$+2$
$\{\neg p\}$	D	$\{\neg p, d\}$	0.33	$+2$
$\{\neg p\}$	D	$\{\neg p, \neg d\}$	0.67	$+2$
$\{d\}$	P	$\{p, d\}$	0	$+2$
$\{d\}$	P	$\{\neg p, d\}$	0	$+2$
$\{\neg d\}$	P	$\{p, \neg d\}$	0.67	$+2$
$\{\neg d\}$	P	$\{\neg p, \neg d\}$	0.33	$+2$

TABLE 3.1: The transition probabilities ($\mathcal{P}^a_{ss'}$) and rewards ($\mathcal{R}^a_{ss'}$) for the finite MDP of the fraud detection robot. There is a row for each possible combination of the current state s, the next state s' and the actions possible in the current state $a \in A(s)$.

labelled with the probability and the reward for that transition. To keep transition graphs as simple as possible, transitions with probability 0 are usually left out of it. The transition graph for the fraud detection robot is shown in Figure 3.2. $\diamondsuit$



∅
P
D
p
¬p
d
¬d
(pd)
(¬pd)
(p¬d)
(¬p¬d)
0.7, -1
0.3, -1
0.4, +2
0.6, -1
0.43, +2
0.57, +2
0.33, +2
0.67, +2
0.67, +2
0.33, +2
Legend:
Final state with conclusion "fraud"
Final state with conclusion "not fraud"
Final state with conclusion "impossible to tell"

FIGURE 3.2: The transition graph for the finite MDP of the fraud detection robot. The large grey circles each represent a state and the small black dots stand for state-action pairs. Each transition, represented by an arrow, is labelled with its probability and reward.

3.4 Value functions

One way to solve an MDP is by applying reinforcement learning. Solving an MDP means finding the *optimal policy*: a policy that maximises the expected total reward. Almost all reinforcement learning algorithms that solve MDPs use *value functions*. These are functions that estimate the *value* of states or of state-action pairs. These values are defined in terms of future rewards that the agent can expect. Since the expected reward depends on what actions the agent will take, the value functions are defined with respect to particular policies.

Recall that a policy π is a function from S and A to probabilities. To be more precise, it is a mapping from states $s \in S$ and actions $a \in A$ to a probability $\pi(s, a)$ of taking action a when in state s. Now, the value of a state s under a policy π, denoted $V^{\pi}(s)$, is the expected reward when starting in state s and then following π. For MDPs, this can be formally defined as:

$$V^{\pi}(s) = E_{\pi}[R_t|s_t = s] = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s],$$

where t can be any time step and where $E_{\pi}[...]$ denotes the expected value given that the agent follows policy π.

For state-action pairs (s, a), the value of a pair under policy π, denoted $Q^{\pi}(s, a)$, is defined as the expected total reward starting from s, executing action a and following policy π thereafter. Or more formally:

$$Q^{\pi}(s, a) = E_{\pi}[R_t|s_t = s, a_t = a] = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s, a_t = a].$$

As mentioned, these value functions are often used in reinforcement learning algorithms to solve MDPs. In Section 3.5, we will show an algorithm that uses Q-values, namely Q-learning.

3.5 Q-Learning

According to Watkins (1989), Q-learning is a form of reinforcement learning that is *model-free*. This means that in this method, the reinforcement learning agent does not have to learn a model of the environment to learn an optimal policy. The idea behind this algorithm is that the agent continuously updates Q-values, hence the values of state-action pairs, depending on its experiences. When this is done until the Q-values converge, the optimal policy is easily found by choosing in each state the action that has the highest Q-value. For the proof that the Q-values will indeed converge with probability 1, see Watkins and Dayan (1992).

In Q-learning, the agent's experience consists of a sequence of episodes (Watkins and Dayan, 1992). During each of these episodes, the agent moves from the initial state to a final state. Every time the agent arrives at a final state, the program enters the next episode. An episode can contain several steps. In the t^{th} step, the agent performs the following sub-steps:

1. It observes its current state s_t.
2. It selects and executes an action $a_t \in A(s_t)$.
3. It receives the immediate reward r_t.
4. It observes the next state s_{t+1}.
5. It updates its value for the state-action pair (s_t, a_t).

The selection of an action in sub-step 2 can be done in multiple ways. One possibility is to always choose a random action. Another option is to explore (choosing a random action) with probability ϵ and to exploit (choosing the action with the highest Q-value) with probability $1 - \epsilon$, where $0 \leq \epsilon \leq 1$. This way of action selection is called ϵ-greedy. Setting the probabilities for selecting actions proportional to their Q-values is sometimes done as well. In that way, every action has a chance to be taken, but actions that were successful in the past have a higher chance.

The updating in the last step is done according to the following formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_{a \in A(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a)\Big),$$

or, formulated differently:

$$Q(s_t, a_t) \leftarrow \alpha\Big(r_t + \gamma \max_{a \in A(s_{t+1})} Q(s_{t+1}, a)\Big) + (1 - \alpha)Q(s_t, a_t), \tag{3.1}$$

where t can be any time step and where $0 \leq \alpha \leq 1$, called the *learning rate*, is a parameter that determines how fast the agent learns.

In words, the updating rule works as follows: every time the agent executes an action a_t in a state s_t, it tries to estimate the value of taking that action in that state, hence the value of the state-action pair (a_t, s_t). This estimation is based on the immediate reward that the agent receives (r_t) and on the discounted future rewards that it will receive if it chooses optimal actions in the successor states ($\gamma \max_{a \in A(s_{t+1})} Q(s_{t+1}, a)$). Subsequently, it shifts its old Q-value slightly towards its estimate by taking a weighted average of its old Q-value and the estimation as its new Q-value. The parameter α determines the rate between the old value and the estimate in this weighted average.

The pseudocode of the Q-learning algorithm is shown in Listing 3.1.

```
Initialise $Q(s,a)$ arbitrarily for all $s,a$
Repeat (for each episode):
    Initialise $s$ randomly
    Repeat (for each step of the episode):
        Select $a \in A(s)$
        Perform action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow \alpha\big(r + \gamma \max_{a' \in A(s')} Q(s',a')\big) + (1-\alpha)Q(s,a)$
        $s \leftarrow s'$
    until $s$ is a final state
```

LISTING 3.1: Pseudocode of the Q-learning algorithm

To illustrate the idea of Q-learning, we now show how the MDP of the fraud detection robot from Example 3.3.1 can be solved with Q-learning.

Example 3.5.1. *The fraud detection robot continued*
Recall that in this example, the agent's goal is to reach a final state (a state in which it has enough information to draw a conclusion) by asking as few questions as possible. We will now apply Q-learning to find out which question should be asked in each state, hence to find the optimal policy.

Before we can start the learning process, we first need to set the discount factor γ and the learning rate α. In this example, the agent always ends up in a final state, which makes the process finite. Therefore there is no need to discount in order to ensure that the total reward is bounded. For this reason and for simplicity, we choose $\gamma = 1$. For the learning rate, we take $\alpha = 0.2$.

According to the Q-learning algorithm from Listing 3.1, the first step is to initialise all Q-values arbitrarily. Let us initialise the Q-values of all state-action pairs to 0.00. We define Q to be the matrix in which all Q-values are shown. At the start of the process, this matrix looks as follows:

$$Q = \begin{array}{r} \\ \emptyset \\ \{p\} \\ \{\neg p\} \\ \{d\} \\ \{\neg d\} \\ \{p,d\} \\ \{p,\neg d\} \\ \{\neg p,d\} \\ \{\neg p,\neg d\} \end{array} \begin{array}{c} \begin{array}{cc} P & D \end{array} \\ \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \end{bmatrix} \end{array}$$

We now demonstrate the first two episodes of the Q-learning in this example.

Episode 1
As shown in Listing 3.1, the first episode starts by randomly choosing an initial state s. Because it is useless to choose a final state as initial state (since the episode is then immediately finished), we will only select among the non-final states ($\emptyset, \{p\}, \{\neg p\}, \{\neg d\}$). In this first episode, the initial state is $s = \{\neg p\}$. Subsequently, an action $a \in A(\{\neg p\})$ needs to be selected. As said above, this selection can be done in multiple ways. For simplicity, we assume random selection in this example. Since $A(\{\neg p\})$ contains only one element, namely D, we have $a = D$. Now D is performed and the resulting next state, s', and reward, r, are observed. In this case, $s' = \{\neg p, \neg d\}$ and $r = +2$.
The first episode is now almost finished, we just have to update the Q-value of the state-action pair that was chosen, namely the pair $(\{\neg p\}, D)$. For this, we need to know $\max\limits_{a' \in A(s')} Q(s', a')$, hence the highest of all Q-values of the next state. As we have seen, the next state is $\{\neg p, \neg d\}$, which has, according to the matrix Q, only Q-values of 0. Therefore $\max\limits_{a' \in A(s')} Q(s', a') = 0$.

Now, the Q-value of the state-action pair $(\{\neg p\}, D)$ can be updated according to the updating rule:

$$\begin{aligned} Q(\{\neg p\}, D) &\leftarrow 0.2\Big(2 + \max_{a' \in A(s')} Q(s', a')\Big) + 0.8 \cdot Q(\{\neg p\}, D) \\ &= 0.2\big(2 + 0\big) + 0.8 \cdot 0 \\ &= 0.4. \end{aligned}$$

The matrix Q therefore changes to:

$$Q = \begin{array}{r} \\ \emptyset \\ \{p\} \\ \{\neg p\} \\ \{d\} \\ \{\neg d\} \\ \{p,d\} \\ \{p,\neg d\} \\ \{\neg p, d\} \\ \{\neg p, \neg d\} \end{array} \begin{array}{c} \begin{array}{cc} P & D \end{array} \\ \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.40 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \end{bmatrix} \end{array}$$

Since the next state $\{\neg p, \neg d\}$ is a final state, the episode ends here.

Episode 2

Again, an initial state s needs to be randomly selected. In this episode, $s = \emptyset$. An action is now randomly selected from $A(\emptyset)$. In this case, $a = P$. When this action is executed, we observe $s' = \{\neg p\}$ and $r = -1$. Before updating $Q(\emptyset, P)$, we need to find $\max_{a' \in A(s')} Q(s', a')$ with the help of the Q matrix. When we look at Q, we see that state $\{\neg p\}$ has Q-values $Q(\{\neg p\}, P) = 0$ and $Q(\{\neg p\}, D) = 0.4$, such that the maximum of these Q-values is 0.4. Therefore

$$\begin{aligned} Q(\emptyset, P) &\leftarrow 0.2\Big(-1 + \max_{a' \in A(s')} Q(s', a')\Big) + 0.8 \cdot Q(\emptyset, P) \\ &= 0.2\big(-1 + 0.4\big) + 0.8 \cdot 0 \\ &= -0.12, \end{aligned}$$

and we can update the Q-values in the matrix to:

$$Q = \begin{array}{r} \\ \emptyset \\ \{p\} \\ \{\neg p\} \\ \{d\} \\ \{\neg d\} \\ \{p,d\} \\ \{p,\neg d\} \\ \{\neg p, d\} \\ \{\neg p, \neg d\} \end{array} \begin{array}{c} \begin{array}{cc} P & D \end{array} \\ \begin{bmatrix} -0.12 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.40 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \\ 0.00 & 0.00 \end{bmatrix} \end{array}$$

Since the next state, $\{\neg p\}$, is now a non-final state, the episode continues. We update s such that $s = \{\neg p\}$ and perform the only possible action $a = D$. Then we observe that $s' = \{\neg p, d\}$ and $r = +2$. To find $\max\limits_{a' \in A(s')} Q(s', a')$, we look at the Q-values of state $\{\neg p, d\}$ in Q and see that those are both 0. Hence

$$\begin{aligned} Q(\{\neg p\}, D) &\leftarrow 0.2\Big(2 + \max_{a' \in A(s')} Q(s', a')\Big) + 0.8 \cdot Q(\{\neg p\}, D) \\ &= 0.2\big(2 + 0.0\big) + 0.8 \cdot 0.4 \\ &= 0.72, \end{aligned}$$

such that

$$Q = \begin{array}{r|cc|} & P & D \\ \emptyset & -0.12 & 0.00 \\ \{p\} & 0.00 & 0.00 \\ \{\neg p\} & 0.00 & 0.72 \\ \{d\} & 0.00 & 0.00 \\ \{\neg d\} & 0.00 & 0.00 \\ \{p, d\} & 0.00 & 0.00 \\ \{p, \neg d\} & 0.00 & 0.00 \\ \{\neg p, d\} & 0.00 & 0.00 \\ \{\neg p, \neg d\} & 0.00 & 0.00 \end{array}$$

Now the next state is final, which makes the episode ends here.

In the above two episodes, the idea of updating Q-values is made clear. After running another 1000 episodes, the Q-values look as follows:

$$Q = \begin{array}{r|cc|} & P & D \\ \emptyset & 1.00 & 1.54 \\ \{p\} & 0.00 & 2.00 \\ \{\neg p\} & 0.00 & 2.00 \\ \{d\} & 0.00 & 0.00 \\ \{\neg d\} & 2.00 & 0.00 \\ \{p, d\} & 0.00 & 0.00 \\ \{p, \neg d\} & 0.00 & 0.00 \\ \{\neg p, d\} & 0.00 & 0.00 \\ \{\neg p, \neg d\} & 0.00 & 0.00 \end{array}$$

From this, we can extract the optimal policy by performing those actions with the highest Q-value. In this case, the optimal policy is therefore the following: ask question D in state $\emptyset, \{p\}, \{\neg p\}$ and question P in state $\{\neg d\}$. In all other states, no actions are possible, since they are final states. Note that in this example the only interesting state to find a policy for is the empty state, since that is the only state where multiple actions are possible. Intuitively, the result we got for this state makes sense. Indeed, in state $\emptyset$ there is a chance to reach a final state ($\{d\}$) in only 1 question when selecting D, while selecting action P always results in two required questions. Asking D first is therefore better than asking P first. $\diamondsuit$

Chapter 4

Q-learning applied to argument-based inquiry

Now that we have covered the basic concepts of reinforcement learning, and in particular Q-learning, in this chapter we will show how this learning method can be applied to optimize the argument-based inquiry in the context of fraud detection.

4.1 Optimizing argument-based inquiry modelled as a reinforcement learning task

As explained in Section 2.3, the argumentation agent does sometimes not have enough information to stabilise its argumentation with respect to its topic. It must then perform an argument-based inquiry to gather the information it needs to reach a stable argumentation. This inquiry should be kept as efficient as possible. Therefore the agent has to learn the optimal question (that is, the question that will help it to stabilise its argumentation with as few questions as possible) in every possible situation. This is attempted through reinforcement learning in the following way.

The agent practices with simulated complaints, generated by a user agent, which will be explained in more depth in 4.4. These 'complaints' consist of a set of observables. Every complaint generated by the user agent contains every possible observable, in its positive or negative form. That is, for every generated complaint C and for each observable o, either $o \in C$ or $\neg o \in C$ holds.

Each episode in the reinforcement learning process, the agent uses a new simulated complaint. At the beginning of such an episode, the agent's knowledge base is always empty (that is, the agent has not observed any observables yet). To achieve its goal, which is stabilising its argumentation with respect to its topic, it needs to gather observables by asking questions. Recall from Section 2.3 that the agent can figure out which questions are relevant based on its knowledge base and the argumentation dialogue. Subsequently, it chooses one such relevant question randomly. After that, it receives an answer that consists of a set of observables. This set is a random subset of the intersection of the observables that are asked (that is, that are linked to the question asked), and the observables that are present in the complaint. The idea behind giving back a subset of the intersection is that in reality, the complainant's answer does not always cover everything that is asked. After the agent has received the answer, it gets a reward for its action (asking the question). This reward depends on the answer that the agent received in the following way: it is positive if the agent can, after asking the question and receiving the new observables, stabilise its argumentation with respect to its topic. However, the agent also gets a negative reward for each question it asks. This negative reward is given because a long dialogue between agent and user is undesirable.

4.2 Optimizing argument-based inquiry modelled as a Markov decision process

The reinforcement learning task described above can be modelled more formally as a Markov decision process (MDP), which we explained in section 3.3. We saw that any MDP is defined by its set of states, its sets of actions, the transition function and the reward function. We now will describe each of these for this particular MDP. For a visual representation of the MDP, see the transition graph shown in Figure 4.1

States
The states of this MDP are all sets of observables. The smallest state is therefore the state in which the agent has no information and the largest states are those which contain every observable in its positive or negative form. The notation of states is just the summation of all observables that represent the state. For example, the state with no observables is notated as $\emptyset$ and for the state that contains observables k and l in their positive and m in its negative form, we write $\{k, l, \neg m\}$.

Final states
In this context, a final state is a state in which the agent has enough information (enough observables) to stabilise its argumentation with respect to its topic. That means that a state is final if the agent cannot change its opinion about its topic anymore when it receives more information. Obviously, a state that contains all observables is final, since in such a state all information is already known, and no extra information will be added anymore. Therefore the agent can never change its opinion about its topic in such states. However, those states are not the only final states, since there are also smaller sets of observables that can produce a stabilised argumentation. Note that in Example 3.3.1 we indeed saw that state $\{d\}$ is a final state, while it does not contain all observables. In Section 2.2, we discussed in what kind of situations this is the case in general. Using the it heuristic that is explained there, the final states of this MDP can be found.

Actions
In this reinforcement learning task, all actions are questions that the agent can ask. Which questions are possible to ask, and thus which actions are available, depends on the state the agent is in, or on which observables the agent knows and which it is still missing. Since the agent is asking questions to gain more information, or more observables, it is useless to ask questions that can only yield observables that are already known. Moreover, it may be the case that some missing observables are irrelevant for its argumentation. Therefore, in each state the only questions, or actions, available are those that could yield observables that the agent is still missing and that are relevant based on its argumentation.

Transition probabilities
The transition probabilities in this MDP are defined by the way that the simulated complaints are generated, explained in section 4.4. The agent does not know these

probabilities at the start of the learning process, but they are implicitly incorporated in the Q-values.

Rewards
We already saw that arriving in a final state (hence, being able to build a stable argumentation), should be positively rewarded, since the agent has then achieved its goal. On the other hand, we want the agent to reach such a state by asking as few questions as possible. Therefore every time the agent asks a question, it receives a small negative reward. In this MDP, we use $r = +n$ whenever the agent reaches a final state, where n is the number of observables. In all other cases we use $r = -1$.

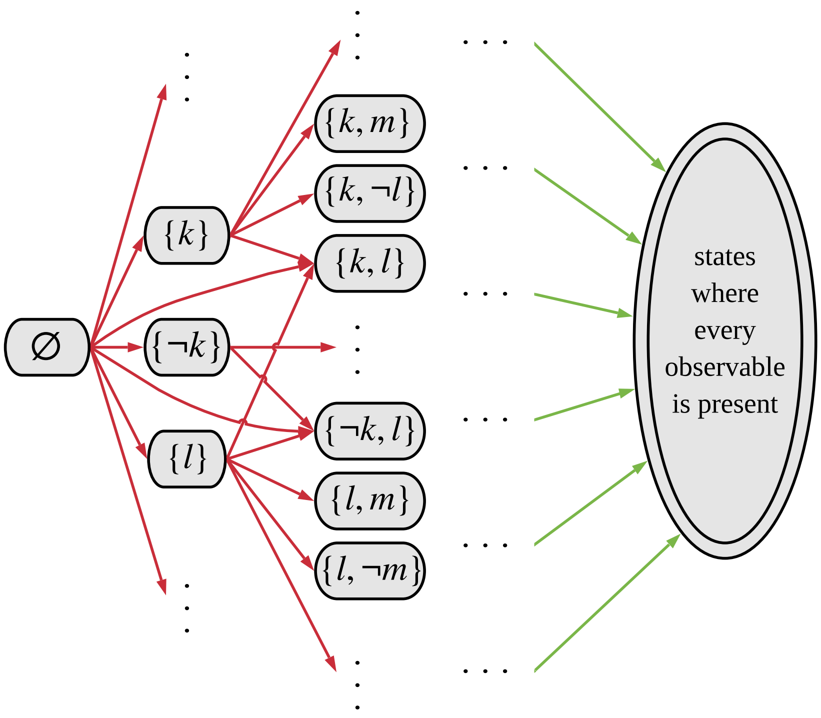


FIGURE 4.1: The transition graph for the MDP that is used to teach the agent which question to ask in each state. Every red arrow indicates a transition that yields a reward of -1 and green arrows stand for rewards of $+n$, where n is equal to the number of observables.

4.3 Solving the MDP with Q-learning

The MDP described in the previous section can be solved by using Q-learning, a method that we explained in Section 3.5. Each episode, the agent's initial state is the empty state. Then, the ϵ-greedy algorithm is used to select an action. That is, with probability ϵ it selects the action that has at that moment the highest Q-value. However, with probability $1 - \epsilon$ it explores and chooses a random action. After observing its successor state and its reward, it updates the Q-value for the state-action pair it has executed. Since the agent always ends up in a final state, there is no need for discounting in order to bound the total reward. Therefore, we set the discount rate to $\gamma = 1$. All in all, the pseudocode of the algorithm that is executed during the Q-learning process is listed in Listing 4.1.

```
Initialise $Q(s,a) = 0$ for all $s,a$
Repeat (for each episode):
    Initialise $s = \emptyset$
    Repeat (for each step of the episode):
        Select action $a \in A(s)$ with the $\epsilon$-greedy algorithm
        Perform action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow \alpha\big(r + \max_{a' \in A(s')} Q(s',a')\big) + (1-\alpha)Q(s,a)$
        $s \leftarrow s'$;
    until $s$ is a final state
```

LISTING 4.1: Pseudocode of the Q-learning algorithm ask

4.4 User agent

To learn which question to ask in a particular situation, the agent should be trained. As we saw in the previous sections, Q-learning can be used for that training. However, the agent then needs a large number of complaints to practice with. Since real complaints are not available in a form that the agent can work with, complaints need to be simulated. A *user agent* is used for this. We define a user agent to be an agent that represents the user, which is in this case the complainant. To simulate real complaints, a so-called *scenario tree* is built. In such a tree, the properties of each possible scenario are encoded by splitting branches, where every level corresponds to a particular property. For example, the root could be split in the following two situations: one where the (possible) fraud has occurred via an official trading website and one where it occurred via a fraudulent webshop. Other properties are for example the kind of product that was sold or the way in which contact between buyer and seller took place (e-mail, Whatsapp, etc.). In each split, probabilities are assigned to each situation that indicate how often that situation occurs. Figure 4.2 shows the root of an example of a scenario tree. Currently, the probabilities in the scenario tree are assigned by humans, based on experience with real complaints. In the future, they might be learned from available data, namely real complaints.

From the scenario tree, complaints can be simulated. This is done by creating a set of observables, using the probabilities of the properties from the scenario tree. These observables then represent a complaint. As said in Section 4.1, the complaints generated by the user agent are used during the reinforcement learning process.

4.5 Practical drawbacks of Q-learning applied to argument-based inquiry

Q-learning is an intuitive and often used method to solve MDPs. However, it has its drawbacks, especially in problems with large state spaces. Because the initialisation of the Q-values is random and every time a state is visited its Q-value is only slightly shifted towards its converged value, Q-learning requires to visit every state many times in order to obtain representative Q-values. Therefore it requires a large amount

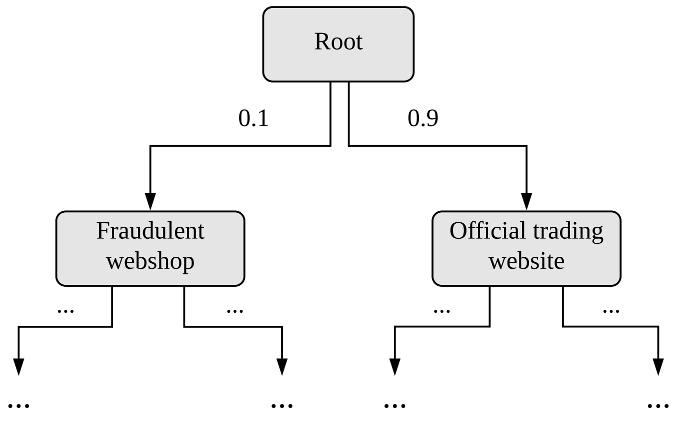


FIGURE 4.2: An example of a scenario tree.

of processing time. Moreover, for every state-action pair a Q-value must be saved. This requires a large number of Q-values, in the order of magnitude[1] of the number of states multiplied by the number of actions. Therefore the memory usage of Q-learning does not scale well.

Besides the large amounts of memory and processing time that Q-learning requires, there exists another drawback. We already explained that, if there are n different observables, there are 3^n different subsets of observables possible. Because in the case of fraud detection the number of observables is very large, the state space of the agent is enormous. Therefore it can easily occur that the agent at some point arrives in a state where it has never been before. In that case, the agent has no policy for that particular state. This could be a problem, because the agent's behaviour then becomes random.

In the remainder of this thesis, we will consider two alternative methods to solve the MDP of the argument-based inquiry. Each of them resolves at least one of the drawbacks that we discussed above.

[1]If m is the number of actions and n the number of possible actions, the total number of state-action pairs is not always equal to $m \times n$. In fact, this is only the case when each action is available in every state.

Chapter 5

Dynamic programming

As explained previously, the main goal of this thesis is to explore various methods that can be used to optimize argument-based inquiry. The method that we present in this chapter is called dynamic programming (Bellman, 1954). With dynamic programming, or DP, every state only needs to be visited once, requiring significantly less time than traditional reinforcement learning algorithms such as Q-learning. Moreover, this method yields a global optimal solution, allowing us to check the quality of Q-learning results. However, memory usage remains problematic, since it scales linearly with the number of states.

5.1 Dynamic programming as a method to solve MDPs

In the Q-learning algorithm discussed in Section 3.5, due to the use of the ϵ-greedy algorithm, most of the times the actions are chosen such that they optimise the reward. That reward depends on the number of questions asked in order to reach a final state: every time the agent asks an extra question, it receives a negative reward, until it reaches a final state, which yields a positive reward. The total reward is therefore higher when less questions are asked. This way, the agent is trained to reach a final state with a minimal expected number of questions. An equivalent approach of this problem would be to forget about rewards and just minimise the expected number of questions required to reach a final state. That is precisely what is done in the dynamic programming method.

The idea behind DP is to divide a large problem into sub-problems that are easier to solve. In the argument-based inquiry described in Section 2.3, the problem is to find an optimal policy for each state. As explained above, this can be done by selecting in each state the action that minimises the expected number of questions required to reach a final (hence, stable) state. In order to find the optimal policy, we therefore need to calculate for each state-action pair (s, a) the expected number of questions that are going to be asked if the agent performs action a in state s. From now on, we will call these the expected values of state-action pairs and denote them by $\mathbb{E}(s, a)$. Calculating the expected values can be difficult, since for most states they depend on the expected values of their successor states, which might be unknown.

For final states however, it is trivial to find an optimal policy: in such states the agent is finished, so no action is required. In these final states we cannot speak of the expected value of state-action pairs, since there are no available actions. Therefore we introduce a new notion, namely the expected value of a state s, for which we will use the notation $\mathbb{E}(s)$. The expected value of all final states is equal to 0. For non-final states, the expected value can be determined when all expected values of the state-action pairs of that state are known: because we are trying to find the optimal policy, we can assume that the agent will select the action for which the state-action pair has the lowest expected value. Therefore the expected value of a state can be found by minimising over the expected values of all the state-action pairs:

$$\mathbb{E}(s) = \min_{a \in A(s)} \mathbb{E}(s, a), \tag{5.1}$$

where $A(s)$ stands for all available actions in state s.

Now, when for a state-action pair (s, a) the expected values of all possible next states are known, we can calculate the expected number of questions that are asked after performing a in state s. This is done by taking a weighted average of the expected values of the successors. The weighted average is based on the transition probabilities between states. Moreover, the action a is in itself also an extra question. Therefore, the expected value of a state-action pair (s, a) can be found with the following formula:

$$\mathbb{E}(s, a) = 1 + \sum_{s' \in S(s,a)} \mathcal{P}^{a}_{ss'} \mathbb{E}(s'), \tag{5.2}$$

where $S(s, a)$ is the set of all possible successor states of s when performing action a and $\mathcal{P}^{a}_{ss'}$ is the probability that the agent moves to state s' when it performs action a in state s. In this formula, the 1 represents the cost of the question of action a itself, while the $\sum_{s' \in S(s,a)} \mathcal{P}^{a}_{ss'} \mathbb{E}(s')$ part represents the number of questions that are asked in the future.

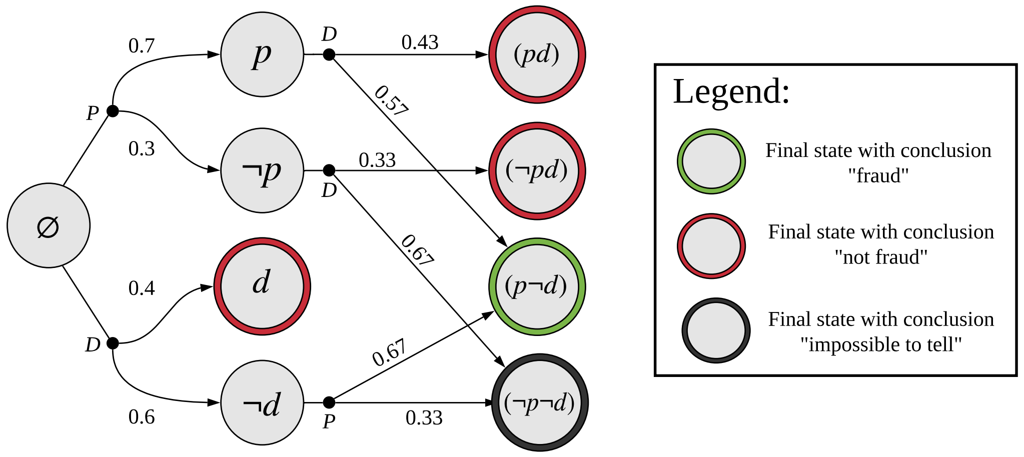
Now, we have divided the large problem (to find the expected value of each state-action pair) into sub-problems (to find the expected value of a state-action pair when the expected values of successors are known). As explained above, those sub-problems are solvable. We can therefore solve the MDP by using dynamic programming as follows. We start at the states with the maximal number of observables. These are all final states, since no more information can be added, and have therefore expected values of 0. Then we work our way back through the MDP, by considering in every step those states that contain one observable less than in the previous step. This way, it is guaranteed that when calculating the expected value of a state-action pair, the expected values of the successor states are already available. This is because in this particular problem, it is only possible to gain information, and not lose. Therefore there exist only transitions to states with more observables than the current state. Because the expected values of the successor states are always known, the expected values of the state-action pairs can be found by using formula 5.2. When all state-action pairs of a certain state are found, the expected value of that state can be determined as shown above in formula 5.1. When we repeat this process until we reach the empty state, we can find the expected value of all states in the MDP. An important thing to keep in mind however, is that in the end we are not interested in the expected values, but in the optimal policy. The expected values are just means to find that policy. When performing the DP algorithm, we should therefore not only store the expected value of states, but also the action with the lowest expected value. The pseudocode of the DP algorithm is shown in Listing 5.1.

```
Initialise $n \leftarrow nr\_observables$
Repeat (for each step):
    For each state $s$ with $n$ observables:
        if $s$ is a final state:
            $\mathbb{E}(s) \leftarrow 0$
        else:
            For each action $a \in A(s)$:
                $\mathbb{E}(s,a) \leftarrow 1 + \sum_{s' \in S(s,a)} \mathcal{P}^a_{ss'} \mathbb{E}(s')$
            $\mathbb{E}(s) \leftarrow \min_{a \in A(s)} \mathbb{E}(s,a)$
            $a^* \leftarrow \arg\min_{a \in A(s)} \mathbb{E}(s,a)$
            save $a^*$ as best action for state $s$
     $n \leftarrow n - 1$
until n < 0
```

LISTING 5.1: Pseudocode of the Dynamic Programming algorithm

To illustrate how DP works in practice, we now show how the MDP of the fraud detection robot from Example 3.3.1 can be solved with DP.

Example 5.1.1. *The Fraud Detection Robot revisited*
Recall that in this MDP, which is shown again in Figure 5.1, the goal is to find a policy for each state that minimises the number of questions required to reach a final state. We will now use dynamic programming to find that policy.



0.7
P
0.3
∅
0.4
D
0.6
p
¬p
d
¬d
D
0.43
0.57
0.33
D
0.67
0.67
P
0.33
(pd)
(¬pd)
(p¬d)
(¬p¬d)
Legend:
Final state with conclusion "fraud"
Final state with conclusion "not fraud"
Final state with conclusion "impossible to tell"

FIGURE 5.1: The transition graph for the finite MDP of the fraud detection robot. The large grey circles each represent a state and the small black dots stand for state-action pairs. Each transition, represented by an arrow, is labelled with its probability.

According to the DP algorithm in Listing 5.1, the first thing to do is to initialise n as the number of observables in the problem. In this case, there are 2 different observables, namely p (paid) and d (delivered). Therefore we set $n = 2$. We now show all steps of the algorithm.

Step n = 2
In this step, we consider all states which contain two observables: $\{p, d\}$, $\{\neg p, d\}$, $\{p, \neg d\}$ and $\{\neg p, \neg d\}$. Let us start with the first one, $\{p, d\}$. Because this is a final state, the expected value of this state should be equal to 0. Therefore we store $\mathbb{E}(\{p, d\}) = 0$. The same holds for all other states with 2 observables, which means that $\mathbb{E}(\{p, d\}) = \mathbb{E}(\{\neg p, d\}) = \mathbb{E}(\{p, \neg d\}) = \mathbb{E}(\{\neg p, \neg d\}) = 0$. According to Listing 5.1, we now must lower n by 1, such that $n = 1$, before we continue with the next step.

Step n = 1
In this step, we focus on all states with 1 observable: $\{p\}$, $\{\neg p\}$, $\{d\}$ and $\{\neg d\}$. Let us consider them one by one.

- State $\{p\}$ is a non final state, which means that we must calculate the expected values of all state-action pairs with state $\{p\}$. In Figure 5.1, we can see that $A(\{p\}) = D$, such that the only state-action pair is $(\{p\}, D)$. The successor states of this state-action pair are $\{p, d\}$ and $\{p, \neg d\}$. Its expected value can now be found with the given formula:

$$\begin{aligned}
\mathbb{E}(\{p\}, D) &= 1 + \sum_{s' \in S(\{p\}, D)} \mathcal{P}^{D}_{\{p\}s'} \mathbb{E}(s') \\
&= 1 + \mathcal{P}^{D}_{\{p\}\{p,d\}} \mathbb{E}(\{p, d\}) + \mathcal{P}^{D}_{\{p\}\{p,\neg d\}} \mathbb{E}(\{p, \neg d\}) \\
&= 1 + 0.43 \cdot 0 + 0.57 \cdot 0 \\
&= 1.
\end{aligned}$$

  Since there was only one available action, we can now immediately set $\mathbb{E}(\{p\}) = 1$ and save the action D as the best (and only) action in this state.

- For state $\{\neg p\}$, we can calculate with the same reasoning the expected value of the only state-action pair with state $\{\neg p\}$:

$$\begin{aligned}
\mathbb{E}(\{\neg p\}, D) &= 1 + \sum_{s' \in S(\{\neg p\}, D)} \mathcal{P}^{D}_{\{\neg p\}s'} \mathbb{E}(s') \\
&= 1 + \mathcal{P}^{D}_{\{\neg p\}\{\neg p,d\}} \mathbb{E}(\{\neg p, d\}) + \mathcal{P}^{D}_{\{\neg p\}\{\neg p,\neg d\}} \mathbb{E}(\{\neg p, \neg d\}) \\
&= 1 + 0.33 \cdot 0 + 0.67 \cdot 0 \\
&= 1.
\end{aligned}$$

  Again, we immediately see that $\mathbb{E}(\{\neg p\}) = 1$ and that the best action in state $\{p\}$ is D.

- State $\{d\}$ is a final state, which means that it gets expected value 0: $\mathbb{E}(\{d\}) = 0$.

- State $\{\neg d\}$ is similar to state $\{p\}$ and $\{\neg p\}$ in the sense that it is no final state and that there is only one available action (in this case P). Therefore, the expected value of state $\{\neg d\}$ can be found in the same way. First, we calculate the expected value of the state-action pair $(\{\neg d\}, P)$:

$$\begin{aligned}
\mathbb{E}(\{\neg d\}, P) &= 1 + \sum_{s' \in S(\{\neg d\}, P)} \mathcal{P}^P_{\{\neg d\}s'} \mathbb{E}(s') \\
&= 1 + \mathcal{P}^P_{\{\neg d\}\{p,\neg d\}} \mathbb{E}(\{p, \neg d\}) + \mathcal{P}^P_{\{\neg d\}\{\neg p,\neg d\}} \mathbb{E}(\{\neg p, \neg d\}) \\
&= 1 + 0.67 \cdot 0 + 0.33 \cdot 0 \\
&= 1.
\end{aligned}$$

It now follows that $\mathbb{E}(\{\neg d\}) = 1$ and the the optimal action in state $\{\neg d\}$ is P.

We have now found the expected value of all states which contain 1 observable and can proceed with the next step, where $n = 0$.

Step n = 0

The only state to consider in this step is the empty state, $\emptyset$. As can be seen in Figure 5.1, there are two available actions in this state: $A(\emptyset) = \{P, D\}$. Following the algorithm from Listing 5.1, we now calculate $\mathbb{E}(\emptyset, a)$ for all $a \in A(\emptyset)$.

- $$\begin{aligned}
\mathbb{E}(\emptyset, P) &= 1 + \sum_{s' \in S(\emptyset, P)} \mathcal{P}^P_{\emptyset s'} \mathbb{E}(s') \\
&= 1 + \mathcal{P}^P_{\emptyset\{p\}} \mathbb{E}(\{p\}) + \mathcal{P}^P_{\emptyset\{\neg p\}} \mathbb{E}(\{\neg p\}) \\
&= 1 + 0.7 \cdot 1 + 0.3 \cdot 1 \\
&= 2.
\end{aligned}$$

- $$\begin{aligned}
\mathbb{E}(\emptyset, D) &= 1 + \sum_{s' \in S(\emptyset, D)} \mathcal{P}^D_{\emptyset s'} \mathbb{E}(s') \\
&= 1 + \mathcal{P}^D_{\emptyset\{d\}} \mathbb{E}(\{d\}) + \mathcal{P}^D_{\emptyset\{\neg d\}} \mathbb{E}(\{\neg d\}) \\
&= 1 + 0.4 \cdot 0 + 0.6 \cdot 1 \\
&= 1.6.
\end{aligned}$$

Now that the expected value of all state-action pairs is known, we must minimise over these values to find $\mathbb{E}(\emptyset)$. We see that

$$\mathbb{E}(\emptyset) = \min_{a \in A(\emptyset)} \mathbb{E}(\emptyset, a) = \min\{\mathbb{E}(\emptyset, P), \mathbb{E}(\emptyset, D)\} = \min\{2, 1.6\} = 1.6\ .$$

Moreover, for the optimal action a^* in state $\emptyset$ it holds that

$$a^* = \arg\min_{a \in A(\emptyset)} \mathbb{E}(\emptyset, a) = \arg\min_{\{P,D\}} \{\mathbb{E}(\emptyset, P), \mathbb{E}(\emptyset, D)\} = D.$$

Therefore we save D as best action in state $\emptyset$. The algorithm has now come to an end.

The optimal policy can now be determined by selecting in each state the action that was found to be the best action during the dynamic programming algorithm. In this case, the optimal policy is to ask question D in the states $\emptyset$, $\{p\}$ and $\{\neg p\}$ and to ask question P in state $\{\neg d\}$. All other states are final and do therefore not require a policy. Note that this is the same result as we obtained with Q-learning in Example 3.5.1. ♢

5.2 Dynamic programming applied to argument-based inquiry

The method of dynamic programming can be used to optimize the argument-based inquiry that is performed during the processing of complaints at LMIO. The MDP that is used for this is almost the same as the MDP described in Section 4.2, which was shown in Figure 4.1. However, they differ on the rewards and the transition probabilities:

- In the MDP from Section 4.2, which was designed to be solved by Q-learning, we defined rewards for every transition. Since DP uses the notion of expected values instead of rewards, these rewards are irrelevant in an MDP that is solved by DP. Therefore, the rewards of the MDP for Q-learning are removed in the one for DP.
- As for the transition probabilities, in Section 4.2 we already mentioned that they are defined by the way the user agent from Section 4.4 generates the simulated complaints. For Q-learning, it is not required to know the specific probabilities, since the agent does not use them during the learning process. In DP however, the transition probabilities play a crucial role, since they are needed to calculate the expected values of states and state-action pairs. Therefore we need to extract the probabilities from the user agent. Although this is possible, it requires a lot of computation and therefore takes a long time. Another possibility is to assume that the transition probabilities are unconditional, thus that the probability of receiving observable $\neg p$ when asking P is always the same, independent of the observables that are already known.

The (final) states and actions are defined in exactly the same way as in Section 4.2. This results in an MDP that can be solved by the algorithm shown in Listing 5.1. That is, by starting to calculate the expected value of states with the maximum number of observables and work back to the empty state, while saving for each state the action for which the state-action pair has the highest expected value.

Chapter 6

Neural fitted Q iteration

In this chapter, we discuss a third method to optimise argument-based inquiry, namely neural fitted Q iteration, or NFQ (Riedmiller, 2005). This is a version of the traditional Q-learning algorithm explained in Section 3.5. The main advantage of this method is that it significantly reduces memory usage in comparison to traditional Q-learning. Moreover, NFQ might have another advantage, namely generalisation of states. In Section 6.4, we will tell more about this possible advantage.

In traditional Q-learning, a Q-value is saved in a Q-table for every state-action pair. As we explained in Section 4.5, this requires a number of Q-values in the order of magnitude of the number of states multiplied by the number of actions. Saving all those Q-values explicitly requires a large amount of memory. This required memory could be significantly reduced by using an approximation function with a bounded number of parameters, which, given a state, returns the Q-values of each possible action in this state. Then, the required memory does not depend on the number of states and actions, but only on the number of parameters that is used. This approximation function needs to be updated along with the Q-learning process. That is, the Q-values returned by this function must change as long as the agent is learning, just like the Q-values are updated in traditional Q-learning. Regression models like linear regression or random forest can be used to represent such a function, thus to approximate the Q-function. Another possibility is to represent the Q-function by a neural network. In the following section, we first discuss the basic concepts of neural networks. After that, in Sections 6.2 and 6.3 we show how a neural network can be used in the NFQ algorithm applied to optimising the argument-based inquiry in the context of fraud detection.

6.1 Basic concepts of neural networks

A neural network can be defined as a model of reasoning based on the human brain. That means that neural networks imitate the structure of the brain. This structure consists of a densely interconnected set of basic information-processing units, which are called neurons. A neural network, too, consists of a number of simple processors, or neurons, that are connected to each other. In fact, a typical neural network is made up of different layers of neurons: an input layer, one or more hidden layers, and an output layer. The number of neurons, also called nodes, in these layers depends on what the network is meant to represent. Every neuron in a layer is connected to at least one, but often to multiple (or even all) neurons of the next layer. See Figure 6.1 for the structure of a neural network with one hidden layer.

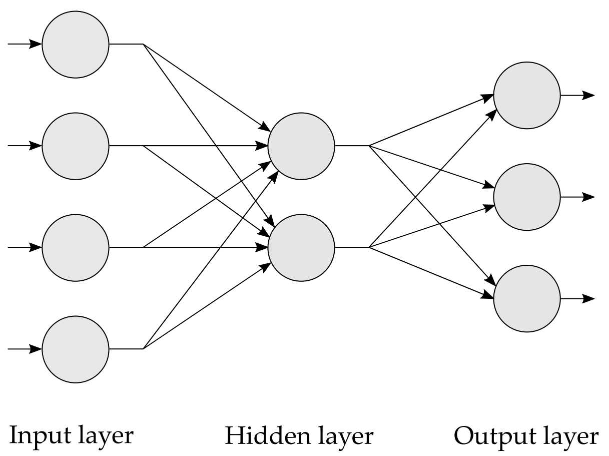


FIGURE 6.1: An example of the structure of a neural network with one hidden layer. In this network, the input layer consists of four, the hidden layer of two, and the output layer of three nodes or neurons. The neurons of all adjacent layers are connected with each other.

The connections between neurons are weighted links that can pass signals from one neuron to another. As can be seen in Figure 6.1, each neuron can receive multiple input signals through its incoming connections. Moreover, a neuron always produces a single output, that can be transmitted to multiple neurons. The output depends on three things, namely the value of the input signals, the weights of the incoming connections and a so called activation function. It is determined in the following two steps:

1. The neuron computes a weighted sum, based on the connection weights, of the input signals. That is, when the neuron receives n input signals $x_1, x_2, ..., x_n$ with corresponding weights $w_1, w_2, ..., w_n$, the weighted sum is equal to

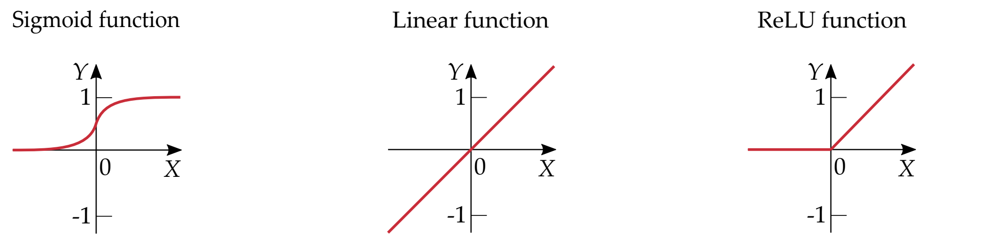
$$X = \sum_{i=1}^{n} w_i x_i.$$

2. To transform the input X, which can be any finite number, into a more convenient value, an activation function is used. One example of such a function is the Sigmoid function, which computes the output Y given the input X as follows:

$$Y^{\text{Sigmoid}}(X) = \frac{1}{1 + e^{-X}}.$$

   Two other common activation functions are the linear function and the ReLU (rectified linear unit) function. See Figure 6.2 for a graphical representation of these three activation functions.

Since the activation function is fixed throughout the training process, the only variables within the network are the connection weights. The network is able to learn by updating these weights after processing training examples. This updating is usually done by backpropagation. This is an algorithm used to adjust the weights in the network such that they minimise the current error made by the network. That error is the difference between the desired (or target) output vector and the predicted



$Y^{\text{Sigmoid}}(X) = \frac{1}{1+e^{-X}}$ $\quad$ $Y^{\text{linear}}(X) = X$ $\quad$ $Y^{\text{ReLU}}(X) = \max(0, X)$

FIGURE 6.2: A graphical representation of three commonly used activation functions that determine the output Y of a neuron given the input X.

output vector, and is computed by a so called loss function. One of the most commonly used loss functions is mean squared error (MSE). Suppose the target vector is denoted by $y = (y_1, y_2, ..., y_m)$ and the prediction vector by $\widehat{y} = (\widehat{y}_1, \widehat{y}_2, ..., \widehat{y}_m)$. Then the MSE loss function computes the loss as follows:

$$\text{loss} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \widehat{y}_i)^2.$$

In this thesis we will not explain the details of the backpropagation algorithm. In our implementation we use the commonly used Adam backpropagation algorithm. For more information about this method we refer the reader to Kingma and Ba (2014).

6.2 Implementation of the neural network for neural fitted Q iteration

In this section, we explain how we implement the neural network for our NFQ algorithm applied to argument-based inquiry optimisation. As mentioned before, the neural network needs to be able to accept a state as input, and output the corresponding Q-values of each possible action for that state.

First of all, recall that a state is a combination of observables, which can either be absent, or present in two ways (positive or negative). Therefore, we represent an observable by a node that can take the values 1 (if the observable is present in its positive form), -1 (if its negative form is present) and 0 (if the observable is absent). Therefore, the input layer consists of as many nodes as the number of observables, so n. An example of this is shown in Figure 6.3.

The output layer should return the Q-values for each state-action pair corresponding to the input state. Therefore, the output layer consists of one node for each possible action. In our case, an action always corresponds to asking for a specific observable. That means that the number of nodes in the output layer equals n.

In between the input and output layer, one can use a large variety of hidden layers. In this thesis, we use a single fully connected hidden layer of $3n$ nodes. For the activation function of the hidden layer, we use both a Sigmoid and a ReLU function. Further research is necessary to optimise the results.

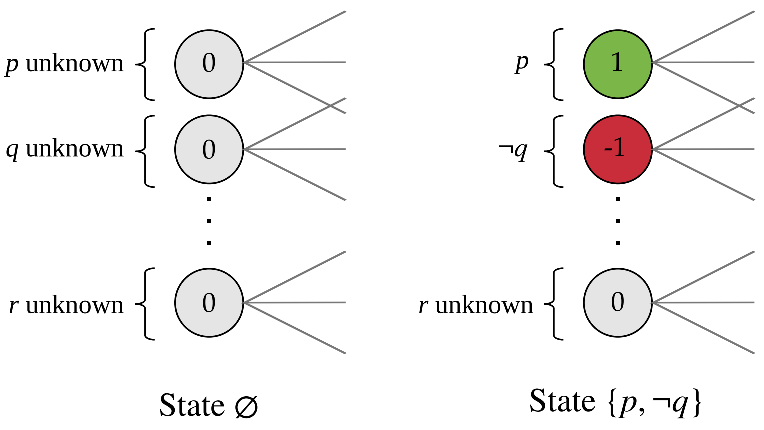


FIGURE 6.3: The input layer of the neural network for state $\emptyset$ (left) and for state $\{p, \neg q\}$ (right). Every node represents an observable, where a grey node (with value 0) denotes that the corresponding observable is unknown. Green nodes (with value 1) are used when the corresponding observable is present in its positive form. Red nodes (with value -1) denote that the negative form of the corresponding observable is present in the state.

6.3 Neural fitted Q iteration applied to argument-based inquiry

In the NFQ algorithm, the way the agent learns is exactly the same as in traditional Q-learning. That is, the initial state is the empty state ($\emptyset$) and the agent repeatedly executes the following steps: it selects and performs an action a, observes its reward r and next state s' and then updates the Q-value of the relevant state-action pair (s, a). The selection of the action is again done with the epsilon-greedy algorithm. With a probability ϵ, the agent chooses a random action, instead of the action with highest Q-value. The only difference between NFQ and traditional Q-learning resides in the way the Q-values are retrieved and updated.

Retrieving Q-values
As mentioned before, when using NFQ, the Q-values are calculated using the neural network, instead of directly being saved in a Q-table. When a state is entered in the input layer, the neural network returns an estimated Q-value for every action in the output layer. However, in many states, not every action is available anymore. Only the Q-values of actions that are relevant, as determined by the argumentation module, are used. The others are simply ignored.

Updating Q-values
Before discussing how the updating of Q-values works in NFQ, we first introduce some notation. The output of the neural network with state s as input will be denoted by $\widetilde{Q}(s)$. This is a vector of length n which contains for every action an estimated Q-value, computed by the neural network. The element of that output vector corresponding to the action a, so the estimated Q-value of state-action pair (s, a), is called $\widetilde{Q}_a(s)$.

Now recall that in NFQ, Q-values are not explicitly saved. Instead, they are implicitly stored in the neural network. Therefore we should actually speak of updating the neural network, rather than updating the Q-values. The updating of the network is done in the following way. After performing action a in state s and observing the next state s' and reward r, the target value of the relevant state-action pair (s, a), which is denoted by $T(s, a)$, is computed. This target value is an estimation of the value of taking action a in state s. Like in Formula 3.1 in Section 3.5, the estimation is based on the immediate reward that the agent receives and on the discounted future rewards that it will receive if it chooses optimal actions in the successor states. The target value is therefore equal to:

$$T(s, a) := r + \gamma \max_{a' \in A(s')} \widetilde{Q}_{a'}(s').$$

Note that the future Q-values ($\widetilde{Q}_{a'}(s')$) are now also estimated using the neural network. Just like in the traditional Q-learning algorithm, we set $\gamma = 1$. This value for gamma can be justified by the fact that, again, there is no need for discounting to bound the total return, since the agent always ends up in a final state. Therefore we can redefine the target value as:

$$T(s, a) := r + \max_{a' \in A(s')} \widetilde{Q}_{a'}(s'). \tag{6.1}$$

After the target value $T(s, a)$ is computed, a target vector is created. This vector is the same as the current prediction $\widetilde{Q}(s)$ at every position, except for the position corresponding to action a. At this position, the current prediction contains the element $\widetilde{Q}_a(s)$, which was the estimated value of (s, a) before receiving the latest information. The estimation of that value has now changed to $T(s, a)$. Therefore, in the target vector the element $\widetilde{Q}_a(s)$ is replaced with $T(s, a)$. An example of a prediction and target vector is shown in Figure 6.4.

$$\begin{array}{cc} \begin{pmatrix} -1.32 \\ -0.23 \\ 3.52 \\ \vdots \\ 2.33 \end{pmatrix} & \begin{pmatrix} -1.32 \\ 1.08 \\ 3.52 \\ \vdots \\ 2.33 \end{pmatrix} \\ \textbf{Prediction} & \textbf{Target} \end{array}$$

FIGURE 6.4: An example of a prediction vector and a target vector. In this case, the second action is considered and the computed target value is equal to 1.08. To create the target vector, only the second element of the prediction vector is replaced by the target value. Every other element remains unchanged.

The target vector is then compared to the current prediction $\widetilde{Q}(s)$ by using the MSE loss function. Since the prediction and the target vector are identical on all positions except for the position corresponding to action a, the total loss is equal to:

$$\begin{aligned} \text{loss} &= \Big(T(s, a) - \widetilde{Q}_a(s)\Big)^2 \\ &= \Big(\big(r + \max_{a' \in A(s')} \widetilde{Q}_{a'}(s')\big) - \widetilde{Q}_a(s)\Big)^2. \end{aligned}$$

This loss is used in the backpropagation algorithm to update the weights in the neural network. All in all, the NFQ algorithm is as shown in Listing 6.1.

```
Initialise all weights in neural network arbitrarily
Repeat (for each episode):
    Initialise $s = \emptyset$
    Repeat (for each step of the episode):
        Let neural network compute $\widetilde{Q}(s)$
        Select $a \in A(s)$
        Perform action $a$, observe $r$, $s'$
        Let neural network compute $\widetilde{Q}(s')$
        $T(s,a) \leftarrow r + \max_{a' \in A(s')} \widetilde{Q}_{a'}(s')$
        Create target vector by using $T(s,a)$
        Compute loss with MSE loss function
        Perform backpropagation to update weights
        $s \leftarrow s'$
    until $s$ is a final state
```

LISTING 6.1: Pseudocode of the neural fitten Q iteration algorithm

To illustrate the working of the NFQ algorithm, we now solve the MDP of the fraud detection robot from Example 3.3.1 with the Q-learning variant that uses a neural network to represent the Q-values.

Example 6.3.1. *The fraud detection robot revisited*
Recall that in this MDP, which is shown again in Figure 6.5, the goal is to find a policy for each state that minimises the number of questions required to reach a final state. We will now apply NFQ to the problem to find that policy.

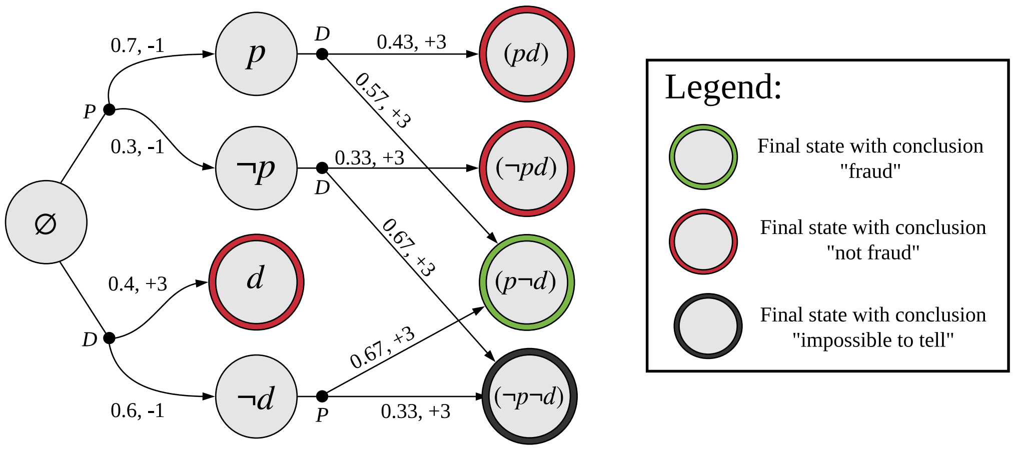

∅
P
D
0.7, -1
0.3, -1
0.4, +3
0.6, -1
p
¬p
d
¬d
D
0.43, +3
0.57, +3
0.33, +3
0.67, +3
0.67, +3
0.33, +3
P
(pd)
(¬pd)
(p¬d)
(¬p¬d)
Legend:
Final state with conclusion "fraud"
Final state with conclusion "not fraud"
Final state with conclusion "impossible to tell"

FIGURE 6.5: The transition graph for the finite MDP of the fraud detection robot. The large grey circles each represent a state and the small black dots stand for state-action pairs. Each transition, represented by an arrow, is labelled with its probability and reward.

As explained in Section 6.2, we will use a neural network with an input layer of $2n$ nodes, one fully connected hidden layer with $3n$ nodes and an output layer of n nodes. Recall that in this example there are two observables, namely p (paid) and d (delivered). Therefore the input, hidden and output layer consists of respectively 4, 6 and 2 nodes. For the hidden layer we use the Sigmoid activation function and for the output layer the linear function. See Figure 6.2 for a graphical representation of these activation functions. Moreover, the network is fully connected, which means that each node has a connection with every node in adjacent layers. All in all, the structure of the neural network that we will use is shown in Figure 6.6.

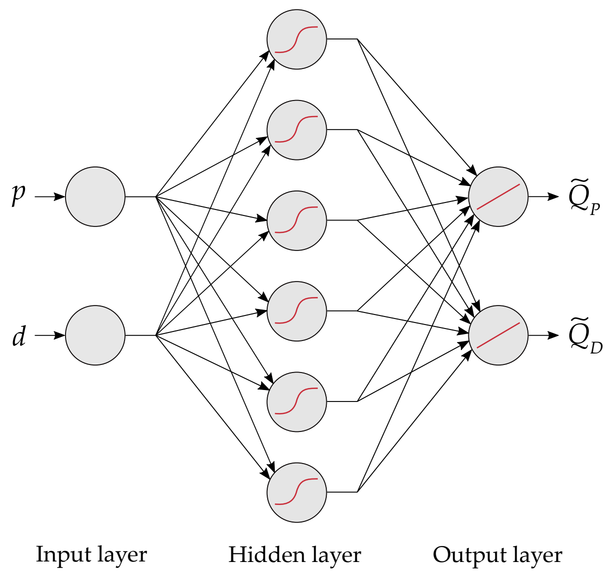


FIGURE 6.6: The structure of the neural network that is used to solve the MPD of the fraud detection robot. The input, which is a state, is represented by two nodes: one for every observable. Each input node can take the value 0 (if the observable is absent), 1 (if the observable is present in its positive form) or -1 (if the observable is present in its negative form). The hidden layer is fully connected and contains 6 nodes, all with a Sigmoid activation function. The output layer consists of two nodes with a linear activation function. The first node represents the estimated Q-value of action P for the input state, and the second node the Q-value of D.

Following the algorithm from Listing 6.1, we first initialise all the weights in the neural network arbitrarily. Then the Q-learning process, of which we will demonstrate the first episode, starts.

Episode 1

According to Listing 6.1, the episode starts by initialising $s = \emptyset$. Subsequently, we let the neural network compute the estimated Q-values for each action in state $\emptyset$ and learn that $\widetilde{Q}(\emptyset) = (0.28, 0.74)$. We use these values to select an action $a \in A(\emptyset) = \{P, D\}$. For the sake of simplicity, in this example we will use the greedy algorithm for action selection (that is, ϵ-greedy with $\epsilon = 0$). Since in the current prediction $\widetilde{Q}_D(\emptyset) > \widetilde{Q}_P(\emptyset)$ (namely $0.74 > 0.28$), the selected action is D. Now, the next state and corresponding reward is observed. In this case, the agent moves to state $\{d\}$ and receives a reward of 2. We use this information to update the neural network. Therefore we first need to compute $\widetilde{Q}(\{d\})$ and $T(\emptyset, D)$. It follows that

$\widetilde{Q}(\{d\}) = (0.46, 0.79)$ and

$$\begin{aligned} T(\emptyset, D) &= r + \max_{a' \in A(\{d\})} \widetilde{Q}_{a'}(\{d\}) \\ &= 2 + 0.79 \\ &= 2.79. \end{aligned}$$

Now, we create the target vector by replacing $\widetilde{Q}_D(\emptyset)$ by $T(\emptyset, D)$ such that the target vector becomes $(0.28, 2.79)$. Moreover, we compute the loss of the current prediction with the MSE loss function:

$$\begin{aligned} \text{loss} &= \Big(T(\emptyset, D) - \widetilde{Q}_D(\emptyset)\Big)^2 \\ &= \Big(2.79 - 0.74\Big)^2 \\ &= \; 4.20. \end{aligned}$$

The target vector and the computed loss are now used in the backpropagation algorithm to update the weights in the neural network. Since state $\{d\}$ is a final state, the episode now ends.

All other episodes proceed in the same way. Let us now look at the computed policy after we let a computer perform another 1000 episodes. Recall from Example 3.5.1 that in this MDP the only interesting state to find a policy for is the empty state ($\emptyset$), since in all other states there is no or only one available action. Therefore we will only look at the Q-values that the neural network returns for state $\emptyset$ as input. After running 1000 episodes of Q-learning, the output vector $\widetilde{Q}(\emptyset)$ is equal to $(2.13, 2.60)$, which means that $\widetilde{Q}_P(\emptyset) = 2.13$ and $\widetilde{Q}_D(\emptyset) = 2.60$. Since $\widetilde{Q}_D(\emptyset)$ is the highest of these two estimated Q-values, the policy in state $\emptyset$ will be to ask D. This is the same result as we obtained with traditional Q-learning and dynamic programming. $\diamondsuit$

6.4 Generalisation of states as potential benefit of neural fitted Q iteration

As said before, the major advantage of NFQ over traditional Q-learning is that it uses much less memory, since instead of all Q-values only the weights of the neural network must be saved. Besides this advantage, there might be a second one, namely generalisation of states.

We already explained in Section 4.5 that in the traditional Q-learning algorithm the agent sometimes arrives in a state that it has rarely seen before. This can happen when the state space is very large, but also in smaller state spaces when the probability to reach a certain state is very low. In such situations, the agent has no good policy for that particular state, causing random behaviour. This is an important drawback of traditional Q-learning.

In the NFQ algorithm, the way the agent moves through the state space is the same as in traditional Q-learning. Therefore in NFQ, the above described situation where the agent reaches a rarely seen state, can also occur. However, in the NFQ algorithm, Q-values are computed by a neural network that takes a state as input. States that are very similar to each other in the sense that they have almost the same set of observables, are represented by almost the same input vector. Therefore, similar states are likely to generate similar output vectors and thus to have similar Q-values. In other words, the way of computing Q-values in the NFQ algorithm causes generalisation of states. A consequence of this generalisation is that when the agent reaches a state that it has not seen before, instead of randomly selecting an action, it bases its choice on states that are similar to its current state. It is plausible that in a state without a policy, it is better to follow the policy of a similar state than to behave randomly. Therefore this generalisation might very well be a benefit of NFQ, especially for states that have a low probability of being reached.

6.5 An alternative implementation of the neural network

In Section 6.2, we described an implementation of the neural network in which the input is a state, and the output a vector with a Q-value for each action. During the backpropagation, the weights of the neural network are adapted such that they minimise the difference between the output and the target vector. This target vector is created as shown in Figure 6.4: only the Q-value of the action that has been executed is changed into its target value, all other Q-values are kept the same. This means that the neural network receives the feedback that the Q-values of all non-taken actions were predicted correctly. Therefore, the weights are adapted in a way that reinforces the predicted Q-values of all those non-taken actions. However, the correctness of these predictions is never tested. Since they might as well have been very ill, reinforcing them during backpropagation could be a drawback of the used implementation.

In this section, we present an alternative implementation of the neural network used for NFQ, which differs from the original implementation only in its input and output layer:

- Instead of just a state, which was the input in the original implementation, in the alternative implementation the input consists of a state-action pair. That means that the input layer contains nodes to represent the state, and nodes to represent the chosen action. The representation of the state is the same as explained in Figure 6.3: every observable is represented by a single node. The state-part of the input layer therefore consists of n nodes. To represent the action, one node for each possible action is used, where a 1 determines that the corresponding action is chosen, while a 0 means that the action is not chosen. Since there exist a question for each observable, the action-part of the input layer consists of n nodes as well. As an example, the input layers of state-action pairs $(\emptyset, P)$ and $(\{p, \neg q\}, R)$ are shown in Figure 6.7.
- The output of the neural network in the alternative implementation consists of a single node that represents the Q-value of the given state-action pair.

All in all, the structure of the neural network from Figure 6.6 with its new implementation is shown in Figure 6.8.

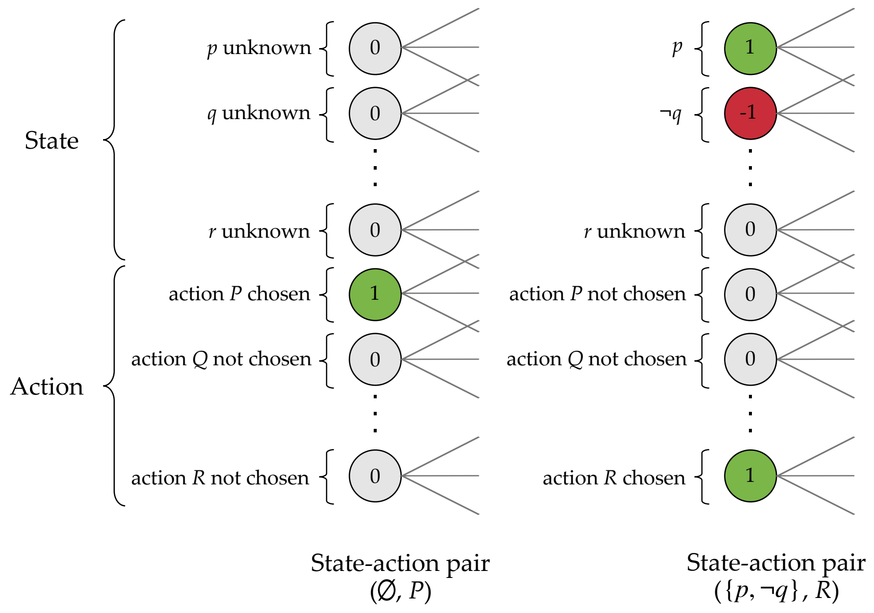


FIGURE 6.7: The input layer of the neural network in the alternative implementation for state-action pair $(\emptyset, P)$ (left) and $(\{p, \neg q\}, R)$ (right). The upper n nodes are used to represent the state and the remaining n nodes denote which action is chosen.

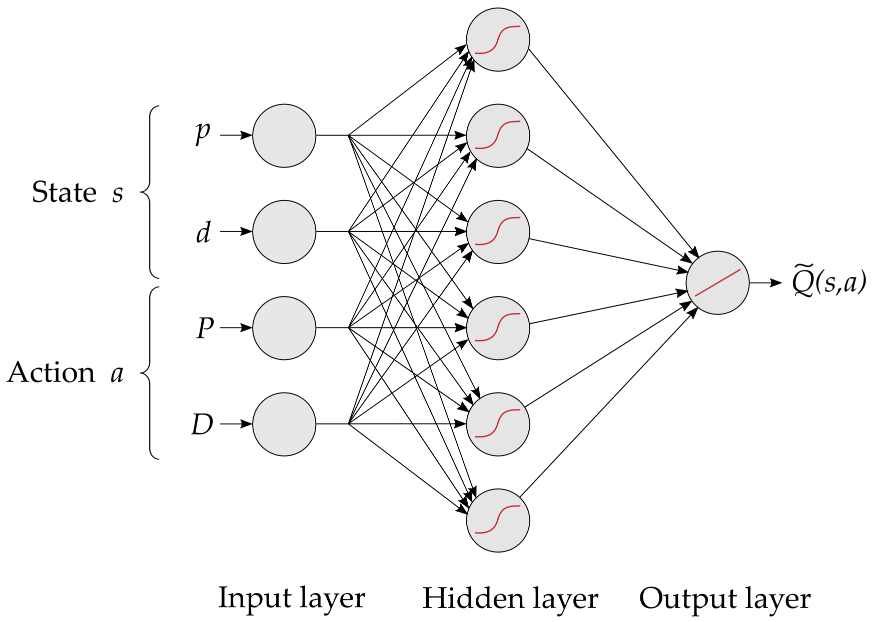


FIGURE 6.8: The structure of the neural network in the alternative implementation. The input, which is a state-action pair (s, a), is represented by four nodes: one for each observable and one for each action. The hidden layer is fully connected and contains 6 nodes, all with a Sigmoid activation function. The output layer consists of one node with a linear activation function. This node represents $\widetilde{Q}(s, a)$, the estimated Q-value of state-action pair (s, a).

The NFQ algorithm, which was shown in Listing 6.1, can stay unchanged when using this new implementation. However, since the input of the neural network is now a state-action pair and the output is a single Q-value, we must change some notation. The output of the neural network is now instead of $\widetilde{Q}(s)$ notated as $\widetilde{Q}(s, a)$, denoting the estimated Q-value of state-action pair (s, a). Formula 6.1 is then changed to:

$$T(s,a) := r + \max_{a' \in A(s')} \widetilde{Q}(s', a'). \tag{6.2}$$

Moreover, since the output is now a single node instead of a vector, we do not have to create a target vector anymore.

Chapter 7

Validity and efficiency

In the previous chapters, we gave a theoretical description of three different methods to optimise argument-based inquiry, namely Q-learning, dynamic programming and neural fitted Q iteration. To illustrate the working of these methods, we applied them to a very small and simplified example in the context of fraud detection. In this chapter, we will test the validity and efficiency of Q-learning and NFQ by applying them to a more extensive and more realistic example, and by comparing their results with the results of DP.

Since we want to measure the performance of the algorithms, we must use a setting with sufficiently many observables, such that solving the corresponding MDP is not a trivial task. On the other hand, we should be able to run the algorithms in a reasonable time to test them with different parameters. Therefore, we will use an example of intermediate size.

7.1 Methods

Since we do not have a meaningful problem of intermediate size in the context of fraud detection at our disposal, in this chapter we use an example from another department of the Netherlands National Police that has nine observables. The setup, that is, the observables, rules and topics, of this example is further described in Appendix A. Since in this example there are nine different observables, the state space exists of $3^9 = 19683$ possible states. Note that for simplicity, we use the convention that every question can only yield a single observable. Moreover, we use constant and uniform transition probabilities. That is, the answer to each question is 'yes' with probability 0.5 and 'no' also with probability 0.5.

In Chapter 5 we mentioned that DP finds a global optimal policy for each state. Because the example we use is of intermediate size, its state space is still manageable for the DP algorithm and therefore the optimal policy can be found. As explained in Section 5.1, this policy is computed by choosing in each state the action that minimises the expected number of questions required to reach a final (that is, stable) state.

The result of DP, which is the optimal policy for the MDP, is then compared to the Q-learning method from Chapter 4, and the NFQ method from Chapter 6. More specifically, after every few iterations, the policies of Q-learning and NFQ of that moment are compared to the optimal policy of DP. The percentage of correctly predicted actions is then saved. This way, we can monitor the performance of each learning algorithm during the learning process. Since both algorithms contain parameters that can influence their performance, multiple values for these parameters are considered.

In some states, there are multiple actions that minimise the required number of questions, since they have the same (lowest) expected value. In such situations, a prediction of Q-learning or NFQ is counted as correct if it coincides with one of these optimal actions. Moreover, in calculating the percentage of correctness, only states with at least two available actions are taken into consideration, since those are the only states in which the agent has something to choose, and therefore learn. In the example used in this chapter, there are 10773 such states, while there are 19683 states in total. If we would consider all states in calculating the percentage, we would count the $19683 - 10773 = 8910$ states in which the agent has no choice as correctly predicted states, causing a too optimistic output.

Since the argument-based inquiry is a stochastic process, the results of Q-learning and NFQ are different every time the experiment is performed. Therefore, each scenario is run multiple times and we then consider the average result.

7.2 Results

In this section, we discuss the results of Q learning and the NFQ algorithm applied to the example of Appendix A. As mentioned in Section 7.1, we will show the percentage of the correctly predicted actions when compared to the policy of DP.

First of all, in Figure 7.1 we plotted the evolution of the percentage of correctly learned actions for the traditional Q-learning algorithm. As explained in Section 3.5, there are two important parameters when applying Q-learning to argument-based inquiry: ϵ from the ϵ-greedy action selection and the learning rate α. To test the influence of these parameters, we choose for every parameter three different values: a low, a middle and a high value. Specifically, for ϵ we choose the values 0.1, 0.5 and 0.9 and for α we choose 0.01, 0.3 and 0.9. As can be seen in Figure 7.1, the performance of the algorithm is increasing with the number of iterations, indicating that the algorithm is learning. However, the speed of learning differs among the different values of parameters. The best result is found when using $\epsilon = 0.9$ and $\alpha = 0.01$. For this case, we obtain a correctness percentage of 90% after 300,000 iterations.

Second, in Figure 7.2, the results for the NFQ algorithm are shown. For this plot, we used the implementation of the neural network that takes only a single state as input, as described in Section 6.2. We use two different activation functions for the hidden layer, namely the Sigmoid and the ReLU function which where both shown in Figure 6.2. The parameters that we consider in this algorithm are ϵ and a decay factor. After every iteration, ϵ is multiplied by this decay factor, which causes ϵ to become smaller over time. This can be advantageous, since in the beginning of the learning process it is useful to explore, while after a while it is better to exploit. In the plot showed in Figure 7.2, we use again the values 0.1, 0.5 and 0.9 for ϵ, but we add a run in which we start with the high value ($\epsilon = 0.9$) and use a decay factor of 0.995.

Interestingly, using the ReLU activation function consistently yields better results than using the Sigmoid function. Moreover, the results with ReLU are more stable. According to Liew, Khalil-Hani, and Bakhteri (2016), this improved stability might

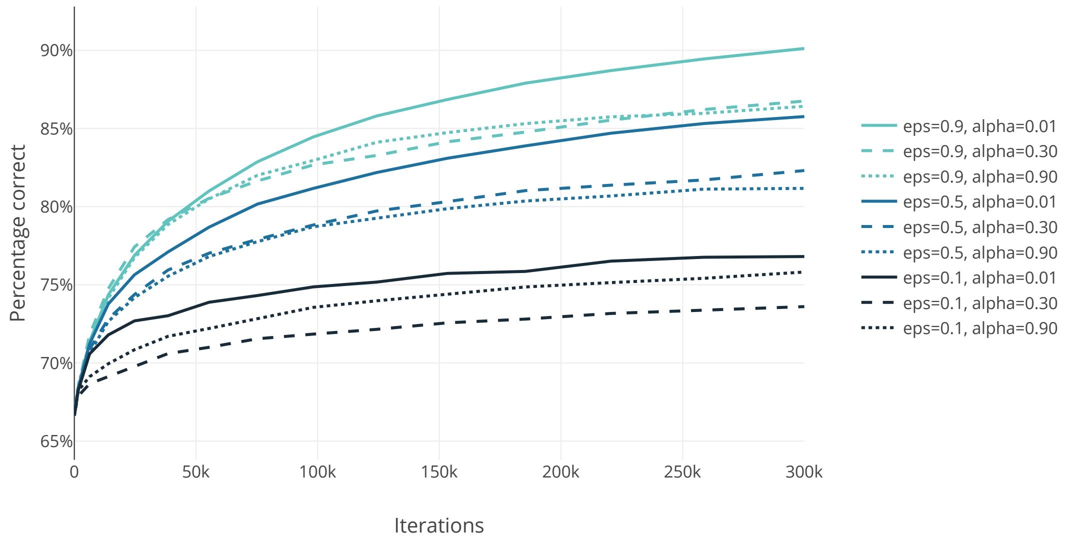


FIGURE 7.1: The results of the Q-learning algorithm for different values of ϵ and α. The x-axis denotes the number of iterations and the y-axis the percentage of correctly predicted actions. We used the average of 5 runs.

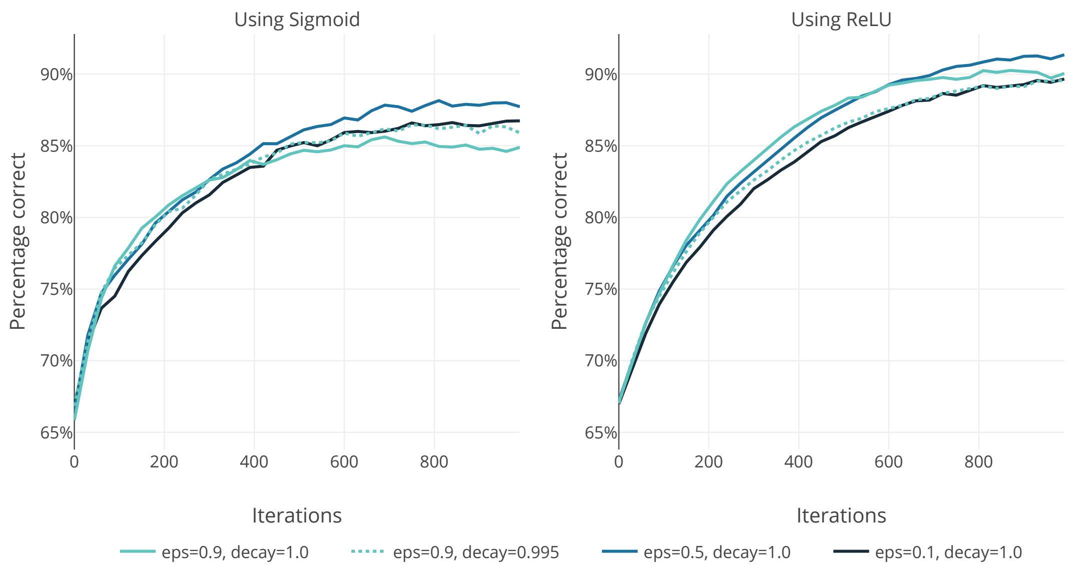


FIGURE 7.2: The results of the NFQ algorithm with a neural network that takes only a single state as input. Different values of ϵ and the decay factor are shown, as well as two different activation functions for the hidden layer, namely Sigmoid (left) and ReLU (right). The x-axis denotes the number of iterations and the y-axis the percentage of correctly predicted actions. For Sigmoid, we used the average of 200 and for ReLU the average of 100 runs.

be explained by the difference in the gradients of both activation functions, which are used during the backpropagation. The gradient of Sigmoid becomes very small when de input is large (positive or negative). Small numbers usually cause numerical instability in computations. ReLU does not suffer from this problem, since its gradient is either 0 or 1. Besides the difference in activation function, we notice that $\epsilon = 0.5$ yields the best results. The use of the decay factor does not cause real improvements. When using ReLU, the decay factor even deteriorates the result. We would like to point out that our NFQ implementation yields a high percentage of 91% after only 1000 iterations.

Third, in Figure 7.3, we show the results obtained by using NFQ with a neural network that accepts a state-action pair as input. This implementation of the neural network is described in Section 6.5. Surprisingly, when comparing Figure 7.3 to Figure 7.2, we see that this method yields significantly worse results than obtained with a neural network in which the input is only a state. It requires further research to investigate where this drop in accuracy comes from. One possible explanation for the poor performance lies in the fact that, when using a neural network to represent the Q-values, a weight change induced by an update in a certain part of the state space might influence the values in arbitrary other regions. It could be that the implementation that takes a state-action pair as input suffers more from that problem than the implementation that uses a single state as input. For the state-action pair implementation, it could therefore be beneficial to use another form of updating, namely off-line updating, on which we will elaborate in Section 8.2. Further research is needed to reveal whether the differences in performance for the two different implementations are still significant when an off-line updating rule is used.

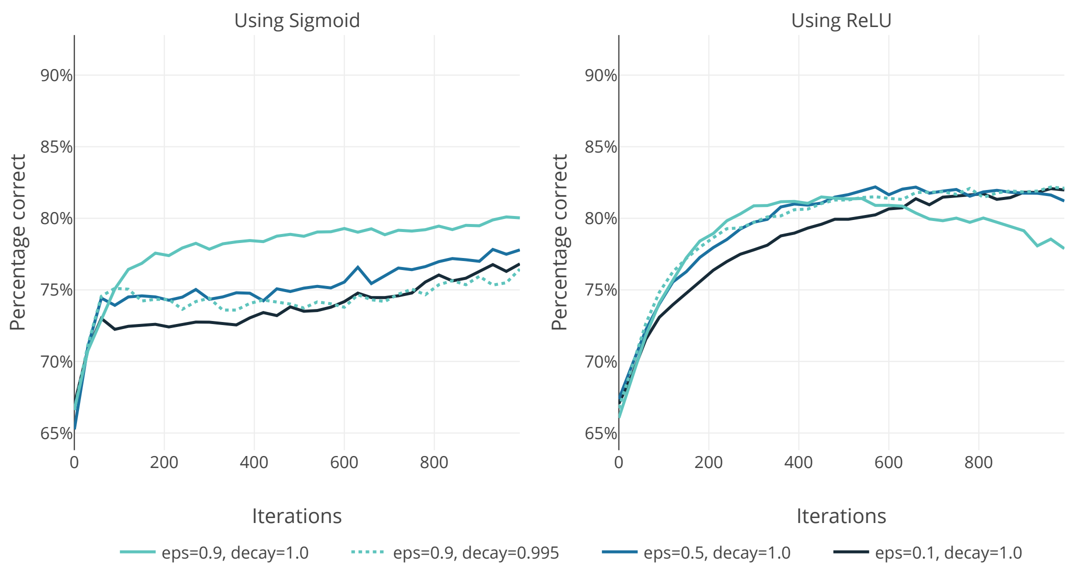


FIGURE 7.3: The results of the NFQ algorithm with a neural network that takes a state-action pair as input. Different values of ϵ and the decay factor are shown, as well as two different activation functions for the hidden layer, namely Sigmoid (left) and ReLU (right). The x-axis denotes the number of iterations and the y-axis the percentage of correctly predicted actions. For Sigmoid, we used the average of 200 and for ReLU the average of 100 runs.

Lastly, we compare the best result of traditional Q-learning with the best result of NFQ in Figure 7.4. Since NFQ requires significantly less iterations that Q-learning to obtain a high correctness percentage, we use a logarithmic scale on the x-axis. The plot shows that after only 700 iterations of the NFQ algorithm, the same accuracy (namely 90%) is reached as after 300,000 Q-learning iterations.

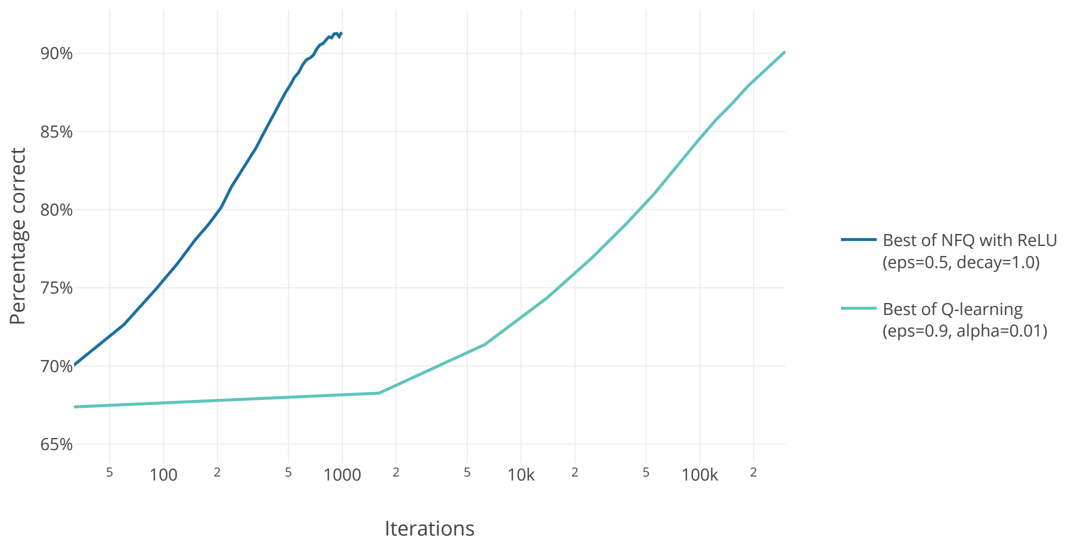


FIGURE 7.4: The best results of traditional Q-learning and the NFQ algorithm. The x-axis denotes the number of iterations and the y-axis the percentage of correctly predicted actions. Since NFQ requires significantly less iterations than Q-learning to obtain a high correctness percentage, a logarithmic scale is used on the x-axis.

Moreover, after 1000 iterations, Q-learning obtains a correctness percentage of only 68%, while NFQ has then already reached 91%. This illustrates that the generalisation of states, which we explained in Section 6.4, is indeed a big advantage of NFQ over traditional Q-learning: it is obvious from Figure 7.4 that after 1000 iterations, Q-learning has not gained enough information about the Q-values to find a good policy. This is because most states have not been visited enough times to obtain representative Q-values. However, while the NFQ algorithm has visited roughly the same number of states after 1000 iterations, its policy already reaches a 91% correctness. This may be explained by the generalisation of states that occurs in NFQ: every time the weights of the neural network are updated, the algorithm gains information not only about the current state, but also about states that are similar to the current state.

It should be noted that the performance of the NFQ algorithm is stagnating at 91%, even when more iterations are run. As explained in Section 3.5, one mathematical property of Q-learning is that it converges to an optimal policy. Therefore we know that Q-learning will eventually reach a correctness percentage of 100%. However, Figure 7.1 already showed that convergence would require a very high number of iterations.

7.3 Conclusion

In the previous section, a number of interesting results where shown. First of all, when using the NFQ algorithm to optimise argument-based inquiry, the ReLU activation function performs better than the Sigmoid function. However, a more important conclusion is that the NFQ algorithm, as proposed in Chapter 6, could be a solution to the problem of large state spaces when applying reinforcement learning to argument-based inquiry. This conclusion is based on the following findings:

- The results of NFQ are, with the right setting of parameters and a reasonable number of iterations, comparable to those of Q-learning. This can be seen by comparing Figures 7.1 and 7.2 in Figure 7.4.
- The large amount of memory required when using Q-learning is significantly reduced when using NFQ. This is because only the weights of the neural network must be saved, instead of the Q-value of every state-action pair.
- Besides the memory usage, the required processing time is significantly reduced by applying NFQ instead of Q-learning as well. This especially becomes clear in Figure 7.4, where the best results of Q-learning and NFQ are showed in the same plot. It can be seen that NFQ requires a lot less iterations than Q-learning to reach a high accuracy (700 against 300,000). Although an iteration of NFQ takes longer than one of Q-learning, NFQ is still significantly faster.
- Due to the generalisation of states that occurs in NFQ, after the learning process has ended, there will be less states without a (reasonable) policy when using NFQ than when using Q-learning. That is, less states in which a wrong action is chosen.

Chapter 8

Related work

In this thesis, we focussed on the combination of reinforcement learning, in particular Q-learning, and argument-based inquiry. We saw that when argument-based inquiry is modelled as an MDP, the state space soon becomes too large to store the Q-values of every state-action pair explicitly. As a solution to this problem, in Chapter 6 we explored the possibilities to represent the Q-function by using a neural network. In this chapter, we discuss how our research relates to other work on Q-learning with a large state space, not necessarily applied to argument-based inquiry. We start by discussing several methods to represent Q-values by an approximation function in Section 8.1, followed by some possible improvements of our current implementation based on recent literature in Section 8.2

8.1 Representing Q-values by an approximation function

In order to be able to apply Q-learning to large state spaces, the Q-values must be estimated by using an approximation function. In general, there are two types of such functions: parametric and non-parametric approximation functions. Among the first category are all functions where parameters are learned to predict the Q-values. Neural networks are almost always used for this. The weights of the neural network are then the parameters that must be learned. The second category contains mainly kernel-based methods, where the Q-function is built from multiple basis functions, or kernels. In this thesis we focussed on a parametric approximation function, namely a neural network. One possible drawback of parametric approximation functions is that they often do not converge to a unique solution (Boyan and Moore, 1995). Moreover, they are generally unstable. Therefore it is interesting to now look into non-parametric approaches that can be found in the literature.

In 2002, Ormoneit and Sen (2002) developed kernel-based reinforcement learning (KBRL). This is an algorithm that assigns value function estimates to the states of an MDP in a sample trajectory and updates these estimates iteratively, where each update is based on kernel-based averaging. A major advantage of this method is that additional training always improves the estimated policy and that it therefore converges asymptotically to the optimal policy. The idea of this algorithm is as follows. Instead of updating the Q-value of a single state-action pair, in KBRL, every iteration an extra kernel is added to the approximation function. Such a kernel is a function that decreases when the distance in the state-action space to the chosen state-action pair becomes larger. Often smooth radial basis functions are used as kernel. The value of these functions only depends on the distance to a certain state-action pair. A commonly used radial basis function is the Gaussian kernel.

As described in Barreto, Precup, and Pineau (2011), a major drawback of kernel-based approaches such as kernel-based reinforcement learning is its high computational complexity. This is caused by the fact that the size of the approximator grows with the number of sample transitions and makes the approach impractical for large problems. As mentioned by the authors, the computational complexity is the reason that kernel-based approaches are not widely applied, but that much more often parametric approaches such as neural networks are used.

8.2 Improvements of our implementation of neural fitted Q iteration

In Chapter 6, we presented neural fitted Q iteration, an algorithm that combines Q-learning with neural networks. Although we obtained some quite satisfactory results, there are of course still aspects that could be improved.

Currently, we use an on-line form of updating the weights of the neural network. That is, the Q-values are updated after each training sample. According to Riedmiller (2005), the problem with on-line updating is that it often takes a long time before an optimal, or near optimal, policy is found. One reason for this is that when the weights are updated for a certain state-action pair, the estimates for other state-action pairs can also change. Although we saw in Section 6.4 that this is also an advantage of neural fitted Q iteration over traditional Q-learning, it can cause unreliable and slow learning. Riedmiller (2005) proposes an off-line neural fitted Q iteration algorithm, where the updating is only performed after a certain number of iterations, considering an entire set of gained transition experiences. This set of experiences is used to adapt the weights of the neural network. It would be interesting to investigate whether off-line updating yields better results.

Another possible improvement could be to adapt an algorithm proposed by Van Hasselt, Guez, and Silver (2016), from the Google DeepMind team. They developed the double deep Q-network algorithm (double DQN). In NFQ, which we introduced in Chapter 6, a single neural network is created that is used for two different things: for choosing an action and for calculating the target value of the chosen state-action pair. This double use can be clearly seen when we write Formula 6.2 in an equivalent form, namely:

$$\begin{aligned} T(s,a) :&= r + \max_{a' \in A(s')} \widetilde{Q}(s',a') \\ &= r + \widetilde{Q}\Big(s', \arg\max_{a' \in A(s)} \widetilde{Q}(s',a')\Big). \end{aligned}$$

In the equivalent formulation, it can clearly be seen that the same Q-network, namely $\widetilde{Q}(\cdot)$, is used twice every iteration. The main idea of Van Hasselt, Guez, and Silver (2016) in their double DQN algorithm is to use two separate neural networks, denoted by $\widetilde{Q}(\cdot)$ and $\widetilde{Q}'(\cdot)$. The formula for the target value then becomes:

$$T(s,a) = r + \widetilde{Q}'\Big(s', \arg\max_{a' \in A(s)} \widetilde{Q}(s',a')\Big),$$

where the weights of $\widetilde{Q}'$ are updated after every iteration, while the weights of $\widetilde{Q}$ are only updated after every N iterations, considering the entire set of transitions that are executed. After the learning process has ended, the final policy is determined by using the network $\widetilde{Q}(\cdot)$.

Bibliography

Barreto, Andre S, Doina Precup, and Joelle Pineau (2011). "Reinforcement learning using kernel-based stochastic factorization". In: *Advances in Neural Information Processing Systems*, pp. 720–728.

Bellman, Richard et al. (1954). "The theory of dynamic programming". In: *Bulletin of the American Mathematical Society* 60.6, pp. 503–515.

Bex, Floris, Joeri Peters, and Bas Testerink (2016). "AI for online criminal complaints: From natural dialogues to structured scenarios". In: *Artificial Intelligence for Justice Workshop (ECAI 2016)*, p. 22.

Boyan, Justin A and Andrew W Moore (1995). "Generalization in reinforcement learning: Safely approximating the value function". In: *Advances in neural information processing systems*, pp. 369–376.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Liew, Shan Sung, Mohamed Khalil-Hani, and Rabia Bakhteri (2016). "Bounded activation functions for enhanced training stability of deep neural networks on visual pattern recognition problems". In: *Neurocomputing* 216, pp. 718–734.

Ormoneit, Dirk and Śaunak Sen (2002). "Kernel-based reinforcement learning". In: *Machine learning* 49.2-3, pp. 161–178.

Prakken, Henry (2010). "An abstract framework for argumentation with structured arguments". In: *Argument and Computation* 1.2, pp. 93–124.

Riedmiller, Martin (2005). "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method". In: *European Conference on Machine Learning*. Springer, pp. 317–328.

Sutton, Richard S, Andrew G Barto, et al. (1998). *Reinforcement learning: An introduction*. MIT press.

Testerink, Bas and Floris Bex (2019). "A Method for Efficient Argument-based Inquiry". In: *Internal technical report, submitted for publication*.

Van Hasselt, Hado, Arthur Guez, and David Silver (2016). "Deep reinforcement learning with double q-learning". In: *Thirtieth AAAI Conference on Artificial Intelligence*.

Watkins, Christopher J. C. H. and Peter Dayan (1992). "Q-learning". In: *Machine Learning* 8.3, pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

Watkins, Christopher John Cornish Hellaby (1989). "Learning from delayed rewards". PhD thesis. King's College, Cambridge.

Appendix A

Example used for testing Q-learning and neural fitted Q iteration

In Chapter 7, we use an example of intermediate size to test Q-learning and neural fitted Q iteration applied to argument-based inquiry. The example contains 9 observables, which are shown in Table A.1. The rules that are used in the argumentation dialogue to determine which observables are relevant in each state are displayed in Table A.2. The topics of the agent are $t\_feedback$ and t_relay_intel. That is, the goal of the agent is to stabilise its argumentation with respect to $t\_feedback$ and t_relay_intel.

$o\_positive\_only\_reply$	$o\_travel\_question$
o_travel_hit	$o\_crime\_suspect$
o_check_id	o_gba_hit
$o\_transfer$	$o\_discussed\_travelling\_with\_peer$
o_intel_hit	

TABLE A.1

$o\_positive\_only\_reply, \neg positive \Rightarrow \neg t\_feedback$
$\neg request \Rightarrow \neg t\_feedback$
$request, request\_handled \Rightarrow t\_feedback$
$o\_travel\_question \Rightarrow request$
$o\_travel\_question, \neg o\_travel\_hit \Rightarrow request\_handled$
$o\_travel\_question, \neg o\_travel\_hit \Rightarrow \neg positive$
$o\_travel\_question, o\_travel\_hit \Rightarrow positive$
$o\_travel\_question, o\_travel\_hit, \neg o\_transfer \Rightarrow request\_handled$
$o\_travel\_question, o\_travel\_hit, o\_transfer, o\_discussed\_travelling\_with\_peer \Rightarrow request\_handled$
$o\_crime\_suspect, o\_intel\_hit \Rightarrow t\_relay\_intel$
$o\_check\_id \Rightarrow request$
$o\_check\_id, \neg o\_gba\_hit \Rightarrow \neg positive$
$o\_check\_id, o\_gba\_hit \Rightarrow positive"$
$o\_check\_id, \neg o\_gba\_hit \Rightarrow request\_handled"$
$o\_check\_id, o\_gba\_hit \Rightarrow request\_handled$

TABLE A.2