INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
RENNES

**Victor KLÖTZER**

**4th year student
at the department
of Applied Mathematics**

Year group 2022

Internship report

# Creation of a Reinforcement Learning Service

**Definition of a generic AI agent for decision support within a SOA platform and principle development of an interaction method with existing simulation modules**


| | |
|---|---|
| *Host company*: | IABG, Industrieanlagen-Betriebsgesellschaft mbH |
| *Address*: | Einsteinstraße 20, 85521 Ottobrunn, Germany |
| *Company supervisor*: | Hartmut SEIFERT |
| *INSA supervisor*: | Aziz BELMILOUDI |

*October 11, 2021*

# Abstract

**English version:**

As a student in applied mathematics at INSA Rennes, France, I present in this document the work I did during my first engineering internship at the end of the fourth year of my studies. Despite travel and social contact restrictions caused by the coronavirus crisis, I had the opportunity to work on-site for three months in a project group at Industrieanlagen-Betriebsgesellschaft mbH (IABG), Germany. My task was to extend an existing IT solution of IABG with a Reinforcement Learning Service.

My internship covered lots of phases of an engineering project, from documentation, popularization, conceptualization, implementation to the presentation of the work. So this internship allowed me to discover and learn about a theory of machine learning and to gain my first professional experience as a mathematical engineer.

**French version:**

Élève-ingénieur en mathématiques appliquées à l'INSA de Rennes, en France, je présente dans ce document le travail que j'ai réalisé dans mon premier stage ingénieur à la fin de ma quatrième année d'étude. Malgré les restrictions en matière de voyages et de contacts sociaux dues à crise du coronavirus, j'ai eu l'occasion de travailler en présentiel pendant trois mois dans un groupe de projet chez Industrieanlagen-Betriebsgesellschaft mbH (IABG) en Allemagne. Ma tâche consistait à ajouter un Service d'Apprentissage par Renforcement à une solution informatique déjà existante de l'IABG.

Mon stage a couvert de nombreuses phases d'un projet d'ingénieur, de la documentation, à la vulgarisation, la conceptualisation, la mise en œuvre jusqu'à la présentation du travail. Ainsi, ce stage m'a permis de découvrir une théorie d'apprentissage automatique et de réaliser ma première expérience professionnelle en tant qu'ingénieur mathématicien.

**German version:**

Als Student der Angewandten Mathematik an der INSA Rennes, Frankreich, stelle ich in diesem Dokument die Arbeit vor, die ich während meines ersten Ingenieurspraktikums am Ende des vierten Studienjahres durchgeführt habe. Trotz der Reise- und Kontaktbeschränkungen verursacht durch die Coronavirus-Krise hatte ich die Möglichkeit, drei Monate lang vor Ort in einer Projektgruppe der Industrieanlagen-Betriebsgesellschaft mbH (IABG) in Deutschland zu arbeiten. Meine Aufgabe war es, eine bestehende IT-Lösung der IABG um einen Reinforcement Learning Service zu erweitern.

Mein Praktikum umfasste mehrere Phasen eines Ingenieursprojektes, von der Dokumentation, über die Popularisierung, Konzeptionierung, Implementierung bis hin zur Präsentation der Arbeit. Dieses Praktikum ermöglichte es mir also eine für mich neue Theorie des maschinellen Lernens kennen zu lernen und erste berufliche Erfahrungen als mathematischer Ingenieur zu sammeln.

# Acknowledgements

I would like to take this opportunity to sincerely thank the people who made it possible for me to complete this internship under very good conditions.

First of all, I would like to thank *Jacques Turbert*, Vice-President of INSA Alumni Rennes, who put me in contact with the company IABG where I did my internship.

Then I would like to thank the three leaders of the RuDi project group at IABG who welcomed me and supported me during the three months of my internship. First there is *Hartmut Seifert*, my internship supervisor, who especially gave me the opportunity to present my work during the internship at the company. Then there is *Ralf Umlauf*, who helped me a lot with the technical and implementation part. Finally, there is *Markus Franke*, who specified the topic of my internship and with whom I exchanged ideas on the development of the concept to be created. I would also like to thank these three gentlemen for their very nice camaraderie.

I would also like to thank three other members of the RuDi team, starting with *Anne Diefenbach*, who was my discussion partner for my internship topic and will certainly be developing the topic further in the context of the project. Next, I would like to thank *Saif Alam*, an office colleague, and *Elsa Demars*, an intern like me but on IT projects, both of whom have been very good work colleagues.

Finally I would like to thank *Ute Indelen*, the secretary of the corporate department I was in, who was always very caring and helpful and also a good conversationalist and advisor for walks and Bavarian excursions.

# Contents

# List of Figures

# 1. Introduction

I am a student engineer in applied mathematics at INSA Rennes, France. During their studies, aspiring INSA engineers are invited to do internships in companies in their field of study in order to discover and learn about their future professional environment. I have just finished my $4^{th}$ year of studies and in this summer 2021, despite the coronavirus health crisis, I had the chance to do my first engineering internship in Bavaria, Germany. For three months, I worked on a new service developed by a project team of the company *IABG*. This document is the report of this engineering internship.

IABG, for IndustrieAnlagen-BetriebsGesellschaft, is a German company that does analysis and test engineering. The company, which was founded by the German government in 1961, offers its customers both consulting and direct implementation of integrated and innovative solutions in the following sectors: Automotive, IT, Mobility & Energy, Environment & Geodata Services, Aerospace and Defence & Security.

At IABG, I joined a team in the Defence & Security department called *RuDi*, for Referenzumgebung Dienste, in English reference environment for services. This team of six people has been developing for more than ten years an IT architecture, called *RuDi platform*, to ensure trustworthy relationships between different military users (between different countries, for example). The services that can be exchanged through this architecture are numerous, and in my case, I had to show how a reinforcement learning service could be integrated into their solution.

Reinforcement learning is a group of machine learning methods in which decision making is the central element, so that what is learned by the reinforcement learning algorithms are optimal strategies for solving some given task. It is currently used in video games, for example, to create artificial players that win against almost any human players because these artificial players have learned to find the optimal strategy to win the game.

For the RuDi project, especially for use in a military context, reinforcement learning may have an important role to play in decision support, for instance in finding the best strategy to use in military operations. Therefore, the objective of my internship was to define a generic Reinforcement Learning Service in the RuDi platform. This Reinforcement Learning Service should be created in such a way that users can create a decision making scenario and then implement their own reinforcement learning algorithms to finally provide decision support to human operators.

To define this Reinforcement Learning Service, I first had to familiarise myself with reinforcement learning in order to provide the RuDi team with an understandable explanation of reinforcement learning for their future use. Then, using WSDL files and Java programming, I had to define and describe the concept of this service by creating a first version and finally prove with an example that the concept of a generic Reinforcement Learning Service is viable. As the RuDi team works with international partners, including participation in NATO meetings and conferences, my work was mainly conducted in English.

Therefore, this internship report first presents the basic theory of reinforcement learning. It then describes the RuDi Reinforcement Learning Service, and finally proves the concept of this service with a simple use case.

Note that in order to make this document easy to understand, some computational issues have been deliberately omitted in this internship report.

# 2. The basics of reinforcement learning

*"I never lose, I either win or learn."*

This sentence by Nelson Mandela basically describes what *Reinforcement Learning* (RL) is about. Reinforcement learning uses an agent which has a mission in an environment. Trial after trial, this agent learns about its environment by walking intelligently through it, sometimes achieving its goal and sometimes not. Depending on the agent's ability to move in the environment, it gets feedback: the more intelligent the movement to fulfill the mission, the more positive the feedback. This way, the agent reinforces its knowledge about the environment as well as the strategy to use in order to successfully complete its mission, so that finally, it almost always wins. The agent learns with the process of "trial and error", and based on the experience, it learns to perform the task better.

## 2.1. Basic idea and elements

The *agent* and its *environment* are the two main entities of reinforcement learning. The agent is a kind of virtual robot that is learning to fulfill the task it was given within an environment: it can perceive and explore this environment and act upon it. It could for instance be a mouse that has to escape a maze in a finite number of moves. These moves the agent can make within the environment are called the actions (e.g. up, down, left, and right for our mouse).

After an action was taken by the agent, the environment returns the new situation the agent is in. These possible situations of the environment are called *states* (e.g. the squares in a maze if it is on a grid, or a screenshot of video game).

In addition to the new state, the environment also returns feedback to the agent for its choice of action from the previous state. This feedback is known as *reward*: for each good action, the agent gets a positive reward, and for each bad action, the agent gets a negative reward. In the case of our mouse, it will for instance get a positive reward of $+10$ if, from its current state, the mouse chooses the correct action to exit the maze. On the contrary, if the mouse chooses a bad action so that it remains in the maze, the environment will give it a negative reward of e.g. $-1$.

The agent's experience comes from the rewards it receives from the environment. This is the only behavioural quality information that the agent obtains, so the values given to the rewards are very important for the correct definition of the agent's mission.

Reinforcement learning therefore serves to solve a specific type of problem where the decision making is sequential and the goal is long-term, so that the agent can reinforce its strategy little by little.The strategy of the agent in choosing which action to take from a current state is called the *policy*. The goal of the agent and thus of the reinforcement learning algorithm is therefore to find the best policy for the given task, i.e. to choose the best actions in the travelled states to maximize the cumulative rewards. Notice that this will often differ from trying to generate immediate maximum rewards.

To be most successful in its mission, the mouse will try to find the best policy. Here for instance, two policies are displayed: the blue policy is better than the green one because the mouse escapes the maze faster, getting a higher cumulative reward by taking the blue trajectory. Notice that trying to maximize immediate reward by collecting the one piece of cheese was not the optimal solution here. Notice also that the green path is worse than the blue one because of the way the mission was defined with the rewards; for example, by giving a bigger reward for a piece of cheese the green path would have become more interesting (as the objective of the mission would have been defined a bit differently).

In the example in Figure 2.1, the mouse (agent) is to escape the maze (environment). A state of the maze contains information about the current position of the mouse and about the presence or absence of pieces of cheese (because the mouse can eat a piece of cheese only once).

After taking an action (up, down, left, or right), the mouse gets a reward: $-1$ if it is still in the maze, $+1$ per piece of cheese and $+10$ if it escapes the maze (the cumulative rewards are noted in colour). With these rewards defining the agent's mission, escape is the most important thing for the mouse to achieve, eating a piece of cheese is an interesting option, and the mouse will try to stay as little as possible inside the maze.

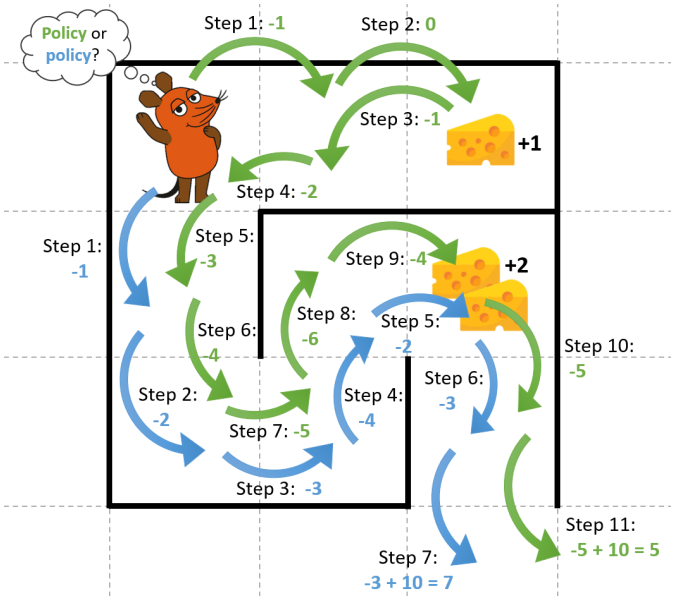



Fig. 2.1.: Example of a mouse escaping a maze

## 2.2. Mathematical formulation

For (video) tutorials and documentation about reinforcement learning, please refer to [jav18], [dee18] and [SB18], which served as a basis for the explanations given in this document.

### 2.2.1. Markov Decision Processes

To more formally model the sequential decision making that reinforcement learning is about, *Markov Decision Processes* (MDP) are used. In an MDP, there is a decision maker called the agent that lives in an environment which is described by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, and a set of rewards $\mathcal{R}$. Without loss of generality for later applications, we can assume that each of these sets has a finite number of elements.

At each time-step $t = 0, 1, \cdots$ the agent receives the state $S_t \in \mathcal{S}$ as representation of the environment. Based on this state, the agent then selects an action $A_t \in \mathcal{A}$. This gives the state-action pair $(S_t, A_t)$.

The time is then incremented to the next time-step $t + 1$, and the environment is transitioned to a new state $S_{t+1} \in \mathcal{S}$. At this moment, the agent receives a reward $R_{t+1} \in \mathcal{R}$ for the action $A_t$ taken from the state $S_t$.

Figure 2.2 gives an illustration of the entire idea of MDP with the running explained on the right.

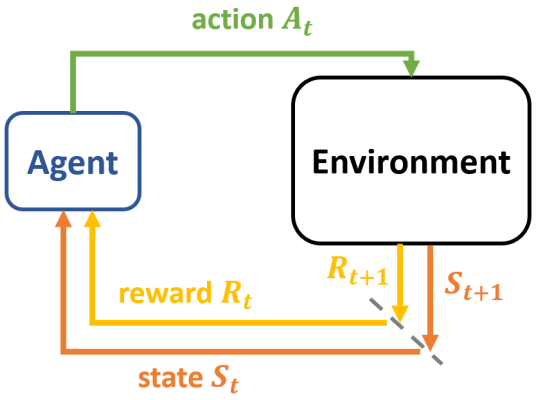



Fig. 2.2.: Markov Decision Process diagram

For each move until the game is over:

1. At time $t$, the environment is in state $S_t$.
2. The agent observes the current state $S_t$ and selects action $A_t$.
3. The environment transitions to state $S_{t+1}$ and grants the agent reward $R_{t+1}$.
4. This process then starts over for the next time-step $t + 1$.
   The dotted line on the bottom right represents the beginning of a new loop: $t + 1$ becomes the present and current time-step $t$, and $S_{t+1}$, $R_{t+1}$ become $S_t$ and $R_t$.

The last concept to be clarified is that of policies. The policy describes how probable it is for an agent to select any action from a given state. A policy is thus a function that maps a given state to the probabilities of selecting each possible action from that state. The symbol $\pi$ denotes a policy and formally we say that an agent "follows a policy".

For example, if an agent follows policy $\pi$ at time $t$, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Say the other way around, at time $t$, under policy $\pi$, the probability of taking action $a$ in state $s$ is $\pi(a|s)$.

### 2.2.2. Discounted return

To get a way to measure how good it is for an agent to be in a given state or to select a given action, we will now introduce the discounted return so that we can then create a value function.

Recall that the agent's goal is to maximize the cumulative rewards. This notion is carried by the *discounted return* that is simply a weighted sum of future rewards. Mathematically, we define the discounted return $G$ at time $t$ as:

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

where $\gamma \in (0, 1)$ is called the *discount rate* or *discount factor*.

This discount rate allows the agent to care more about the immediate reward over future rewards since future rewards will be more heavily discounted. So, while the agent does consider the rewards it expects to receive in the future, the more immediate rewards will have more influence when it comes to the agent deciding about taking a particular action. Furthermore, as the rewards are finite and thanks to $|\gamma| < 1$, the discounted return will always be finite, even if the return at time $t$ would be a sum of an infinite number of terms.

So, it is agent's goal to maximize the expected discounted return of rewards.

Nevertheless, notice that in the agent's decision-making, the discount rate is an important hyperparameter that controls a tradeoff between short-term and long-term interest for future rewards. Small values for $\gamma$ will make the agent more likely to prioritize immediate rewards, while higher values for $\gamma$ will let the agent prioritize long-term rewards in its decision-making. One could for instance suppose that the larger the environment is and the farther the objective is from the agent's initial position, the more it might be necessary to have a long-term interest.

### 2.2.3. Value function (Q-function)

In order to evaluate the long-term result and to estimate how interesting it is for the agent to be in a given state, or to estimate how interesting it is for the agent to perform a given action from a given state, reinforcement learning uses so-called *value functions*.

This notion of how good a state or state-action pair is is given in terms of expected return. The rewards an agent expects to receive are dependent on what actions the agent takes in given states, so the value functions are defined with respect to specific ways of acting. Since the way an agent acts is described by the policy it is following, the value functions are thus defined with respect to policies.

There are at least two types of value function, the state value function and the action value function. We will only talk about the second one here because it is the one used in the reinforcement learning algorithms presented later.

The action value function for policy $\pi$, denoted as $q_\pi$, tells how good it is for the agent to take a given action from a given state while following policy $\pi$. In other words, it gives the value of an action under $\pi$. Formally, the value of action $a$ in state $s$ under policy $\pi$ is the expected return from starting from state $s$ at time $t$, taking action

$a$, and following policy $\pi$ thereafter. Mathematically, we define $q_\pi(s, a)$ as:

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

This action value function is most of the time referred to as the *Q-function*, and the output from this function for any given state-action pair is called a *Q-value*. The letter "Q" stands for the quality of taking a given action from a given state. The Q-function is the value function that is most commonly used in reinforcement learning algorithms because it directly depends on the current state as well as on the chosen action.

### 2.2.4. Optimality

The goal of reinforcement learning algorithms is to find a policy that will yield the highest cumulative reward for the agent if the agent indeed follows that policy. Specifically, reinforcement learning algorithms seek to find a policy that will provide more return to the agent than all other policies: they seek to find an *optimal policy*.

A policy $\pi$ is optimal if all the expected returns under policy $\pi$ are higher than or equal to those under any other policy. Mathematically, $\pi$ is an optimal policy if: or

$$\text{for all policy } \pi^{'}, q_\pi(s, a) \geq q_{\pi'}(s, a) \text{ for all } a \in \mathcal{A} \text{ and for all } s \in \mathcal{S}$$

The optimal policy has an associated *optimal Q-function*, which is denoted $q_*$ and defined as:

$$q_*(s, a) := \max_\pi (q_\pi(s, a))$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. In other words, $q_*$ gives the largest expected return achievable by any policy $\pi$ for each possible state-action pair.

### 2.2.5. Bellman optimality equation

One fundamental property of the optimal Q-function $q_*$ is that it must satisfy the Bellman optimality equation:

$$q_*(s, a) = \mathbb{E}\left[ R_{t+1} + \gamma \max_{a'} \left( q_*(s^{'}, a^{'}) \right) \right]$$

This equation states that, for any state-action pair $(s, a)$ at time $t$, the expected return from starting in state $s$, selecting action $a$ and following the optimal policy $\pi_*$ thereafter (i.e. the optimal Q-value of this pair) is going to be the expected reward $R_{t+1}$ gotten from taking action $a$ in state $s$, plus the maximum expected discounted return that can be achieved from any possible next state-action pair $(s^{'}, a^{'})$.

With this fundamental equation of reinforcement learning, we get an idea of how the agent will recursively find the trajectory that maximizes the future rewards. Since the agent is following an optimal policy, the following state $s^{'}$ will be the state from which the best possible next action $a^{'}$ can be taken at time $t + 1$, and so on.

Therefore, the main objective of reinforcement learning algorithms is to find the optimal Q-function $q_*$ because, for any state $s$, they will be able to find the action $a$ that maximizes $q_*(s, a)$ and thus determine the optimal policy. And to this end, the Bellman equation is the key to success.

## 2.3. Reinforcement learning algorithms

In this part, some concepts of reinforcement learning algorithms will be presented. As with almost all machine learning methods, a training procedure in which the agent tries again and again to accomplish its mission is needed, so that the agent improves its performance iteration after iteration. These repetitions of the same type of mission in the same kind of environment are referred as *episodes*.

### 2.3.1. Exploration/exploitation tradeoff: epsilon greedy strategy

Before going deep into some reinforcement learning algorithms that find the optimal policy in an MDP, we will first focus on an issue that appears when we want the agent to explore its environment. Recall that we want to find the optimal Q-function, and this is basically what is going to be searched or constructed by the algorithms. In order to build this Q-function, two different strategies oppose each other: on the one hand, the idea of exploiting the Q-values found so far because they are supposed to be optimal and it's therefore more likely that an optimal trajectory will follow these higher Q-values. But on the other hand, the agent also needs to try new actions so that it can explore other parts of the environment and maybe discover an even better trajectory. This dilemma is known as the exploration/exploitation tradeoff: for the choice of the next action, the right balance between using exploration or exploitation needs to be found.

The *epsilon greedy strategy* aims to achieve this balance. With this strategy, an *exploration rate* $\varepsilon \in (0, 1]$ is defined, which corresponds to the probability that the agent will explore the environment rather than exploit it. With $\varepsilon = 1$, it is 100% certain that the agent will explore the environment, while if $\varepsilon$ is equal to 0, it is 100% certain that the agent will exploit the found Q-values. As at the beginning of the training, the Q-function is totally unknown, $\varepsilon$ is thus initially set to 1, i.e. the agent will start by exploring.

While the training progresses, the agent learns more about the environment. Therefore, at the start of each new episode, $\varepsilon$ will decay by some *exploration decay rate* $\delta_\varepsilon$ that is set so that the likelihood of exploration becomes less and less probable as the agent learns more and more about the environment. The agent will become "greedy" in terms of exploiting the environment once it has had the opportunity to explore and learn about it. A commonly used formula for this is:

$$\varepsilon_k = \varepsilon_{\text{min}} + (1 - \varepsilon_{\text{min}})e^{-k\delta_\varepsilon}$$

where $\varepsilon_k$ is the value of the exploration rate $\varepsilon$ at episode $k$ and $\varepsilon_{\text{min}}$ is the minimum value that $\varepsilon$ can take (one can for instance choose $\varepsilon_{\text{min}} = 0.01$).

Concretely, to decide whether to explore or to exploit the environment in the next action, a random number between 0 and 1 is drawn for each time-step within an episode. If this number is greater than the exploration rate, then the agent will exploit the environment and choose the action that has the highest Q-value for the current state. If, on the other hand, the random number is less than or equal to the exploration rate, then the agent will explore the environment and sample a random action.

### 2.3.2. Q-Learning

The first reinforcement learning algorithm presented here is called *Q-Learning*. This technique aims to find the optimal policy in a Markov Decision Process, in the sense that the expected value of the total reward over all successive steps is the maximum achievable. So, in other words, the goal of Q-Learning is to find the optimal policy by learning the optimal Q-values for each state-action pair.

To do so, the Q-Learning algorithm iteratively updates the Q-values for each state-action pair until the Q-function converges to the optimal Q-function $q_*$. This is therefore called a value iteration approach. To update the Q-value for the action $a$ taken from the previous state $s$, the Bellman equation seen previously will be used: $q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'}\left(q_*(s', a')\right)\right]$.

The idea is to make the Q-value for the given state-action pair $(s, a)$ as close as possible to the right-hand side of the Bellman equation so that this in-construction Q-value will converge to the optimal Q-value $q_*$. This will happen over time by iteratively comparing the loss between the Q-value and the optimal Q-value for the given state-action pair. Each time this same state-action pair is encountered, the Q-value is updated in order to reduce the loss:

$$loss = q_*(s, a) - q(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'}\left(q_*(s', a')\right)\right] - \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right]$$

To update the Q-values, we finally need to introduce the *learning rate* $\beta \in (0, 1)$. This number can be thought of as how quickly the agent abandons the previous Q-value for a given state-action pair for the new Q-value. For instance, suppose we have the Q-value $q(s, a)$ for some arbitrary state-action pair $(s, a)$ that the agent has experienced in a previous time-step. Then, if the agent experiences that same pair $(s, a)$ again at a later time-step once it has learned more about the environment, the Q-value $q(s, a)$ will need to be updated to reflect the change in expectations the agent now has for future returns. And rather than just overwriting the old Q-value, the learning rate determines how much information will be kept about the previously computed Q-value versus the new Q-value calculated for pair $(s, a)$. This learning rate $\beta$ is another hyperparameter that will have to be adjusted; the larger $\beta$ is, the more quickly the agent will adopt the new Q-value and vice versa.

After receiving a reward $r$ for having taken action $a$ from state $s$, arriving thus in a new state $s'$, the formula for calculating the new Q-value for state-action pair $(s, a)$ is therefore:

$$q^{\text{new}}(s, a) = (1 - \beta) \underbrace{q(s, a)}_{\text{old value}} + \beta \Big( \underbrace{r + \gamma \max_{a'} \Big( q(s', a') \Big)}_{\text{learned value}} \Big)$$

$$= \underbrace{q(s, a)}_{\text{old value}} + \beta \underbrace{\Big( r + \gamma \max_{a'} \Big( q(s', a') \Big) - q(s, a) \Big)}_{\text{loss between learned value and old value}}$$

corresponding to $1 - \beta$ percent of the old value plus $\beta$ of the learned value. Or seen in another way, the new Q-value is the old Q-value plus $\beta$ percent of the difference (loss) between the learned and the old Q-values.

As the number of states and actions is finite, the Q-values can be stored in the so-called *Q-table*, with dimensions the number of states by the number of actions. The Q-table thus gives the updated Q-values for each state-action pair.

At the beginning of an episode, the agent knows nothing about the environment or the expected rewards for any state-action pair, so all the Q-values in the table are first initialized to zero or randomly (except for the final states which have to be at zero because no action will be taken from there). Over time, the Q-table becomes updated, so that in later moves and later episodes, the agent can look in this Q-table and base its next action on the highest Q-value for the current state by exploiting it.

Here is a summary of what the Q-Learning algorithm is doing:

**Algorithm 1:** Q-Learning algorithm

Initialize all Q-values in the Q-table to 0 (or randomly except for the final states)
**foreach** *episode* **do**
  Choose an initial state $s$
  **foreach** *time-step in each episode* **do**
    Choose an action $a$ with the $\epsilon$-greedy strategy (considering this way the exploration/exploitation trade-off)
    Take action $a$ and observe the reward $r$ and the next state $s'$
    Update the Q-value function, i.e. the Q-table, using

$$q^{\text{new}}(s, a) = (1 - \beta) q(s, a) + \beta \Big( r + \gamma \max_{a'} \Big( q(s', a') \Big) \Big)$$

    This formula will, overtime, make the Q-value function converge to the right-hand side of the Bellman equation.
    Set $s$ to the value $s'$

Nevertheless, the use of Q-learning algorithm has its limits. Q-learning is particularly suitable as a learning process when the number of possible states and actions is manageable. Otherwise, the problem will be difficult to solve in a reasonable time. For example, if the number of states is too large, it will be computationally difficult to

update all the values of a very large Q-table until all the values converge to optimality. Basically, one can remember this: the more complex an environment is (for example, a maze with cheese is much more complex than a maze without cheese), the larger the number of states to describe it, the larger the Q-table and the longer the training.

For this reason, in high-dimensional state and action spaces, rather than using a value-based approach for the approximation of the Q-values, model-based approaches are often used, such as the Deep Q-Learning that will be presented later. A value-based approach is discrete and finite in the sens that states must be enumerable, whereas a model-based approach is continuous and does not require states to be enumerable.

Another difficulty in the application of Q-Learning becomes apparent when the rewards are very far from the agent's state and scope of action. If there are no rewards in nearby states, the agent can only propagate rewards far in the future after a long exploration phase.

*During the state of the art research I was interested in some other details of reinforcement learning. In particular I compared Q-Learning with another relatively basic reinforcement learning algorithm and I tried to understand and explain some differences in the types of algorithms, see Appendix A.1 for more details.*

## 2.4. Going Deeper with Deep Q-Learning

As the name suggests, *Deep Q-Learning* uses deep learning and especially deep neural networks called *Deep Q-Networks* (DQN). One of the ideas behind this technique is that for large state spaces, it becomes a very challenging and complex task to define and update a Q-table as it is used for Q-Learning. As the state space increases in size, the time it will take to traverse all those states and iteratively update the Q-values will increase rapidly. With DQN, instead of searching the optimal Q-function by storing the Q-values in a table, a neural network will be used to approximate the Q-function. Therefore, this is a model-based method, where the aim is to reproduce the optimal Q-function with a neural network model. This approach solves the issue of directly dealing with states and all their Q-values, thus allowing a much larger range of states. For instance, we will see that a state could even be a succession of a few images to take advantage of movement information.

### 2.4.1. Deep Q-Networks

Deep Q-Networks have no outstanding features, they have a quite common neural network structure. For a first approach, there is no need to know much about neural networks, apart from the fact that a neural network is just a function $f : x \mapsto y = f(x)$, where $f$ is defined by a set of weights. And in order to build this function $f$, i.e. to find the right weights so that $f$ outputs the right $y$, this function has to be trained.

In our case the $x$ will be the states, the outcomes $y$ of $f$ will be the optimal Q-values, and the so-called Deep Q-Network, which is just a function $f$, has to become an approximation of the optimal Q-function $q_*$, i.e. in the end we want to reach: $f \simeq q_*$.

#### The inputs

A DQN accepts states from a given environment as input. As deep neural networks are now used, it is no longer necessary to represent states using for example a simple coordinate system. Instead, more complex environments can be handled and for instance images or even a small number of sequential images can be used as inputs, i.e. a sequence of images is concatenated to form a single state.

For images, some preprocessing operations will usually be done to try to emphasise the most important details of the images. For instance, RGB data can be converted into grayscale data since the color in the image is usually not going to affect the state of the environment. Additionally, some cropping and scaling to both cut out unimportant information from the frame and shrink the size of the image might be employed.

For example, in the case of a video game, we could grab four consecutive frames as inputs, then grayscale, crop and rescale them the same way and finally stack them on top of each other in the order in which they occurred in the game. This would be done because a single frame usually isn't giving enough information to the network, or even to a human brain. For instance, getting a single image of a moving ball is not sufficient to fully understand the state of the environment. In which direction is the ball moving? How fast is it moving? Is it even moving? If we use four consecutive frames, though, then we would have a much better idea about the current state of the environment.

#### The outputs

For each given input state $s$, the trained DQN outputs the estimations of the optimal Q-values $q_*(s, a)$ for every action $a$ that can be taken from that state $s$.

For example, suppose a game where the actions consist of moving left $a_1$, moving right $a_2$, jumping $a_3$, and ducking $a_4$. Then, for an input state $s$, the output layer of the network would consist of four nodes, each of them representing one of those four actions, i.e. $q_*(s, a_1)$, $q_*(s, a_2)$, $q_*(s, a_3)$ and $q_*(s, a_4)$, see Figure 2.3.

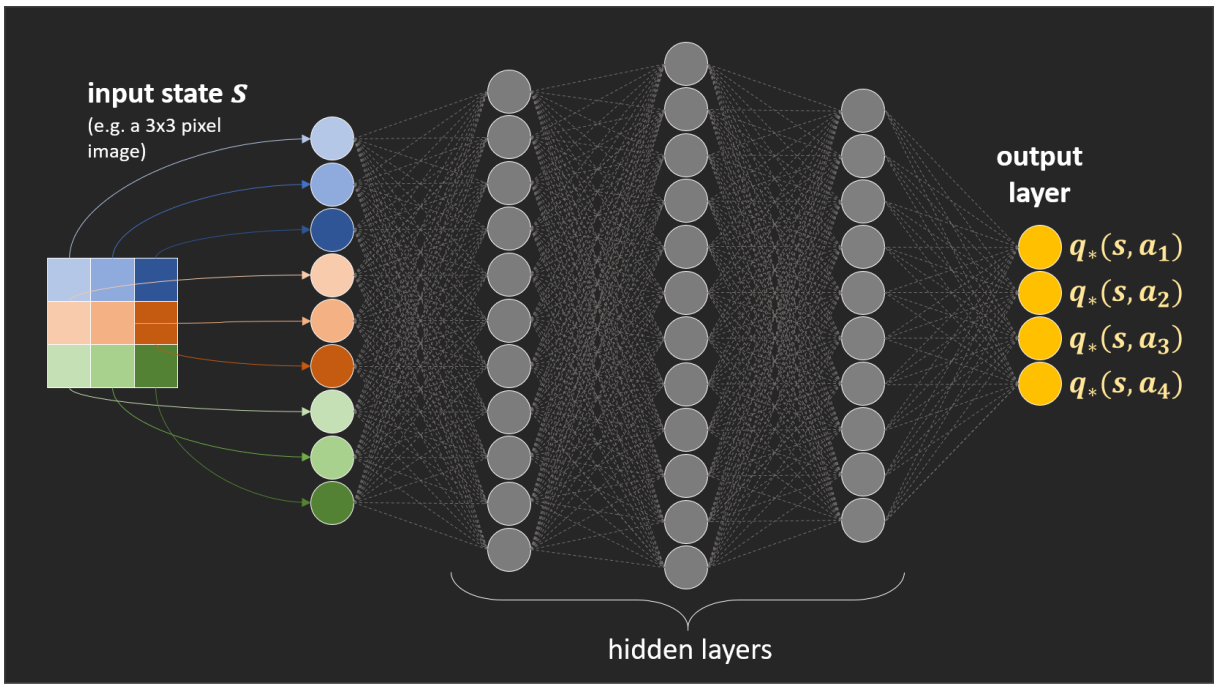



Fig. 2.3.: Example of a trained Deep Q-Network

### 2.4.2. First approach to Deep Q-Learning

As in Q-Learning, the Deep Q-Learning algorithm learns iteratively so that it progressively converges towards the optimal Q-function. Iteration after iteration, while the Deep Q-Network is being trained, a loss is calculated between the current Q-value and the targeted optimal Q-value. This loss is used to update the DQN in order to improve the outputted Q-values and so improve the approximation of the optimal Q-function. Just as in Q-Learning, for the targeted optimal Q-function, the fact that it must satisfy the Bellman equation is used: $q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'}\left(q_*(s', a')\right)\right]$.

Thus, the loss between the in-construction Q-value q(s,a) and the optimal Q-value $q_*(s, a)$ is:

$$loss = q_*(s, a) - q(s, a) = \left(r + \gamma \max_{a'}\left(q_*(s', a')\right)\right) - q(s, a)$$

where $r$ is the reward observed after taking action $a$ from state $s$, and $s'$ is the next state.

To approximate the max-term $\max_{a'}\left(q_*(s', a')\right)$ coming from the Bellman equation, one can simply make a second pass through the DQN using this time the next state $s'$ as input and taking the maximum Q-value outputted.

This process is done over and over again until the loss is sufficiently minimized so that the DQN closely approximates the optimal Q-function.

Here is simplified pseudo-code of the Deep Q-Learning algorithm:

**Algorithm 2:** Simplified Deep Q-Learning algorithm

```
Initialize the Deep Q-Network (DQN) with random weights
foreach episode do
    Choose an initial state s
    foreach time-step in each episode do
        Select an action a from state s (via ε-greedy strategy with the DQN)
        Take action a and observe the reward r and the next state s'
        Pass preprocessed state s throw the DQN and get the Q-value q(s, a)
        Calculate loss between output Q-value q(s, a) and target Q-value q_*(s, a)
          The target Q-value q_*(s, a) is obtained by a pass through the DQN of the next state s' and using
          the Bellman equation.
        Update the weights in the DQN
        Set s to the value s'
```

*For more details and explanations about an effective Deep Q-Learning algorithm, one can have a look at Appendix A.2.*

## 2.5. Outlook

Reinforcement learning combined with deep learning can result in very powerful and efficient algorithms, as the Deep Q-Learning presented above. Nevertheless, it is important to know that there is a multitude of reinforcement learning algorithms and that, depending on the characteristics of the considered problem, some algorithms will perform better than others (here is a non-exhaustive list of different reinforcement learning algorithms: https://en.wikipedia.org/wiki/Reinforcement_learning#Comparison_of_reinforcement_learning_algorithms).

As seen in the few algorithms presented above, there are a number of parameters that need to be adjusted for an agent to best learn to perform its task. For this purpose, it will be necessary to train an agent with different combinations of parameters in order to find the best result. Also, a clever state-preprocessing might hugely improve the performance.

The case of multi-agent reinforcement learning has not been covered here. Like ants working together, the idea behind these multi-agents is to combine the intelligence of several agents to gain even more performance and complete more complicated missions. However, it should be noted that the mathematical and computational challenges are becoming greater when one wants to use more and more agents. This is why the Reinforcement Learning Service defined in the next chapter is currently designed for reinforcement learning algorithms with just one agent.

# 3. A generic reinforcement learning service

In this chapter, I explain how I defined a service to solve reinforcement learning problems in the RuDi platform. The main objective of my internship was to create a first version of this service and then to prove with an example that the concept of a generic Reinforcement Learning Service is viable. This example is presented in chapter 4.

## 3.1. Reinforcement learning in a Service-Oriented Architecture

A Service-Oriented Architecture (SOA) can be seen as a design style of a computer solution in which a communication protocol defined between different services of a program intends to simplify the implementation and the maintenance of software. The RuDi platform developed by the RuDi team of IABG is a SOA platform.

First, here is a sketch of the Reinforcement Learning Service. Figure 3.1 contains the main reinforcement learning elements presented in the previous pages and gives a first overview of how information is to be exchanged between the different entities of the Reinforcement Learning Service of RuDi: a user entity on the left communicates with a reinforcement learning procedure consisting of an agent and an environment. In addition, the definition of the environment is stored in a Storage Service. These interfaces and the Reinforcement Learning Service itself will be presented in more detail in the following sections.

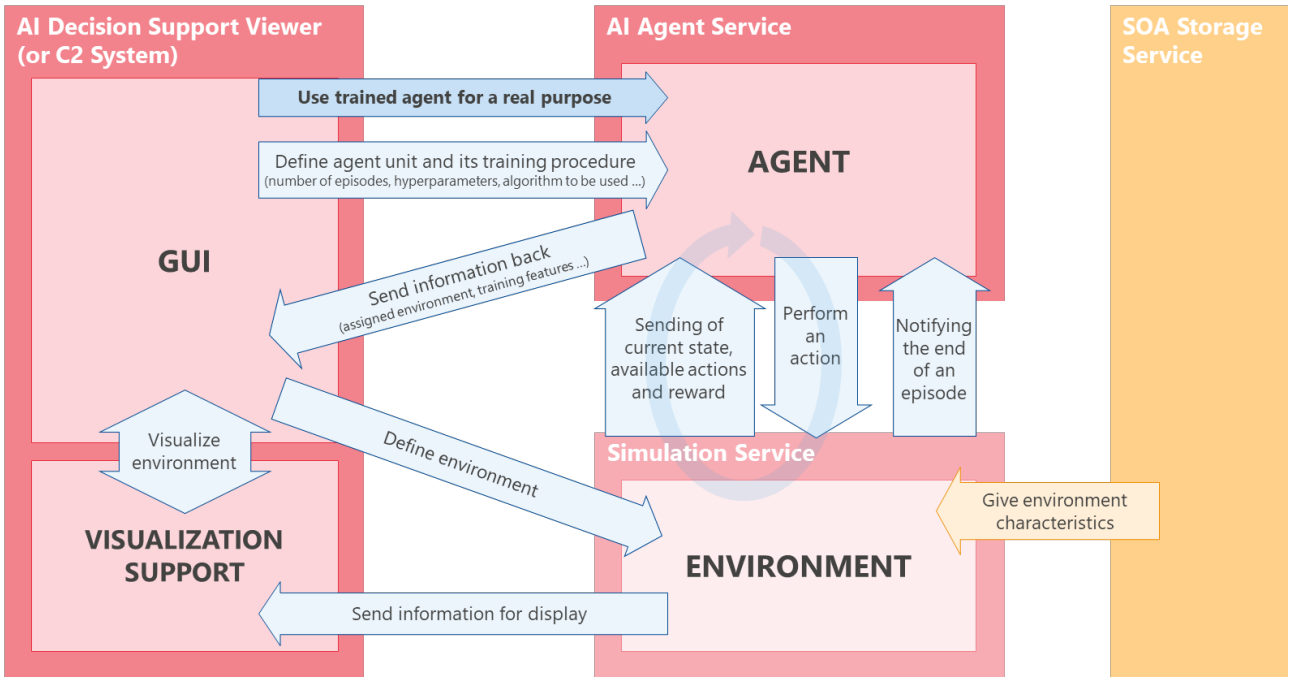



Fig. 3.1.: Reinforcement Learning Service sketch

## 3.2. Architecture of a Reinforcement Learning Service

The SOA Reinforcement Learning Service of the RuDi platform has to be a generic interface for modelling any kind of reinforcement learning problem. In other words, with the reinforcement learning interface provided by RuDi, a *developer* will be able to implement their own reinforcement learning services with various algorithms, for several agents in different environments. The *end-user* will then be free to use the created service to train agents in a user-defined environment and finally employ the trained agents for a real purpose.

The Reinforcement Learning Service is composed of the following elements:

- a SOAP/XML-based service interface (SOAP is a communication protocol used in the SOA and XML is the computer language used in it, see Glossary for more details),
- the business code of the reinforcement learning agents and environments,
- and a data storage for the Reinforcement Learning Service (this point is not covered in this report).

The definition of the interface is the very first step for such a service as it will give it the overall structure. Since the service is to be generic, the business code part is not hard-coded into the Reinforcement Learning Service but has to be provided or selected by the developer according to the reinforcement learning problem they have, and the data they have at their disposal. Thus, the following section describes the definition of the interface, and for the business code part an example is presented in chapter 4.

## 3.3. Reinforcement Learning Service interface definition

### 3.3.1. Structure of the service

The first thing to describe for the Reinforcement Learning Service interface are the main entities that will communicate with each other. For the reinforcement learning part, an agent entity and an environment entity are of course necessary. In addition to that, a user entity is also needed for a human operator to define and communicate with agents and environments, see Figure 3.2.

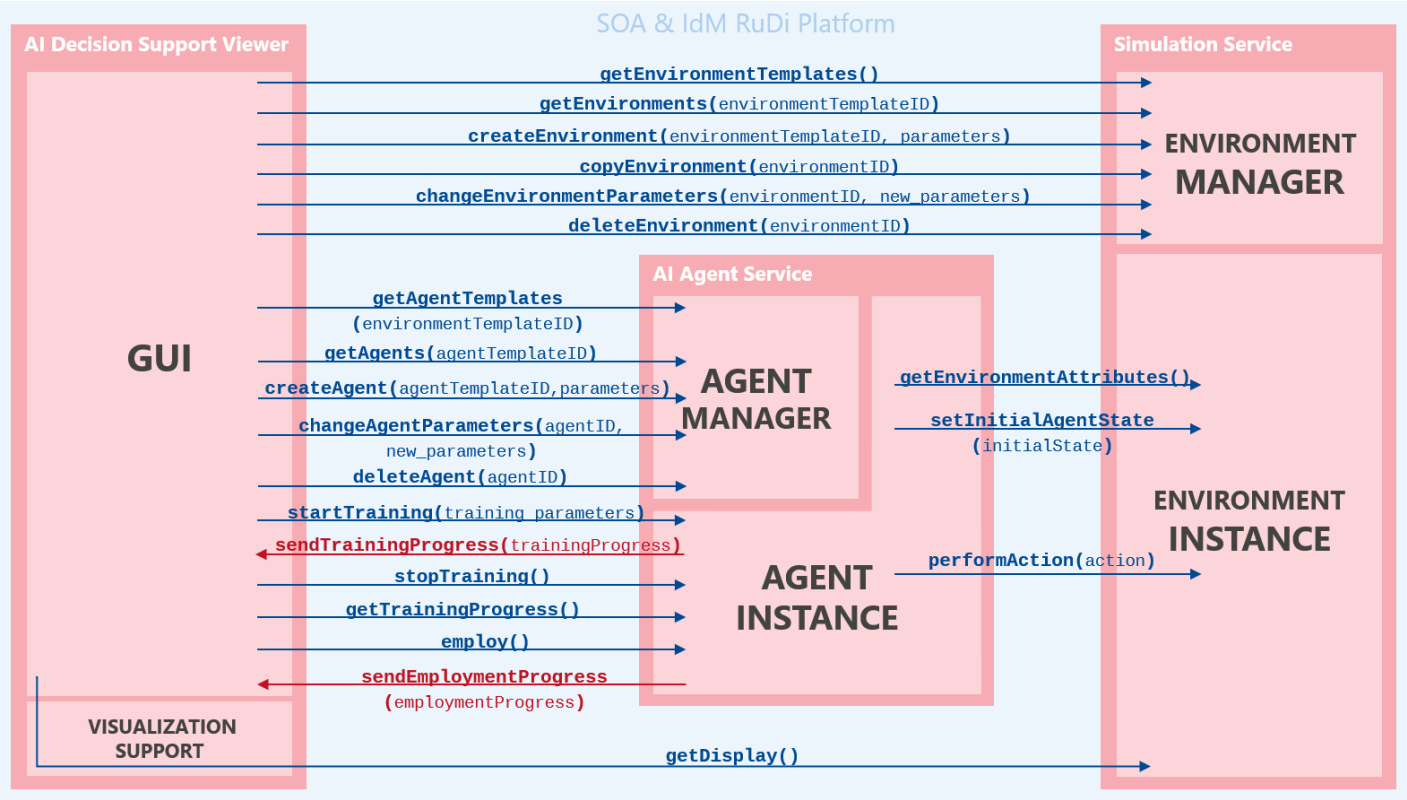



Fig. 3.2.: Reinforcement Learning Service interface overview

Thus, three sub-services compose the Reinforcement Learning Service: an *AI Decision Support Viewer* from which a human operator has control and access to reinforcement learning services, an *AI Agent Service* for managing and parameterizing agents, and a *Simulation Service* for managing environments and providing them to agents.

The Simulation Service is itself divided into an *Environment Manager* and *Environment Instances*. The Envi-

ronment Manager contains environment templates, i.e. an implementation of the environment without definite parameterization or dimensioning. These environment templates implemented by a developer can then be used by an end-user to create some operating environments instances, in which an agent can operate and evolve.

The AI Agent Service is also divided into two sub-entities, which are the *Agent Manager* and *Agent Instances*. The Agent Manager contains agent templates for a given environment template, i.e., agents that are implemented to be used on a given type of environment but without the values for the agent's learning parameters for example. The Agent Instance entity is an agent created from an agent template that can be trained and finally employed on an environment (instance).

### 3.3.2. A SOAP service defined with WSDL files

Here is first a general presentation of the interactions and methods of the service as shown in Figure 3.2 above.

From the *GUI* (Graphical User Interface), two main operations can be taken. The first one is creating a training procedure, i.e. defining an environment and an agent with its training procedure, and then launching the training so that this agent can learn to fulfill its mission in its environment. The second main operation is to use a trained agent for a real purpose.

The other very import interaction shown in this figure is between an agent instance and its environment instance. Recall that to learn to fulfill its mission, the agent needs to discover its environment step by step and always get feedback about the actions it took. This is basically the reinforcement learning training procedure.

A last communication presented here is the one between the environment and the operator. This interaction is necessary to give the end-user a display of what is happening in the environment, to see how the agent is currently behaving.

The SOA Reinforcement Learning Service is a SOAP service, thus communication between the entities is defined with so-called WSDL files. As each of the five entities provide methods to at least one other entity, five interfaces and thus five WSDL files are needed to define the messages to be exchanged between the entities. Figure 3.3 gives a design display of those five interfaces.

Now that the methods have been defined, it is worth looking at the types of objects that will be exchanged in the messages received and returned by the methods. A detailed description of the methods of these interfaces, which are defined in the WSDL files, can be found in Appendix A.3.

### 3.3.3. Exchanged types defined in XSD files

In addition to these WSDL files, to define the data elements that will be exchanged in the messages, XML Schema Definition files, in short XSD files, define the different types of data exchanges within the Reinforcement Learning Service. All the types are defined in these separated XSD files, which are then imported into each WSDL file so that the types can be used for the messages.

More precisely, two XSD files will be used. The main one named *types.xsd* (see Figure 3.3) defines all the fundamental types exchanged within the messages, for example the types *agentInfos*, *agentParameters*, *environmentInfos*, *environmentAttributes*, *action*, *reward*, *state*, *trainingParameters*... These types are predefined and cannot be directly changed.

Nevertheless, recall that the definition of the Reinforcement Learning Service must be generic, so some data elements cannot be very precisely defined as they will depend on the problem to be solved. To allow this flexibility, some of the fundamental types contain so-called *<any> elements*. With these <any> elements that can be filled in with any kind of type, the developer will be able to define the inner details of their Reinforcement Learning Service. Therefore, a second XSD file named *_ANY_elements.xsd* (see Figure 3.3) that is to be written by the developer is used to complete the <any> elements left in the fundamental types.

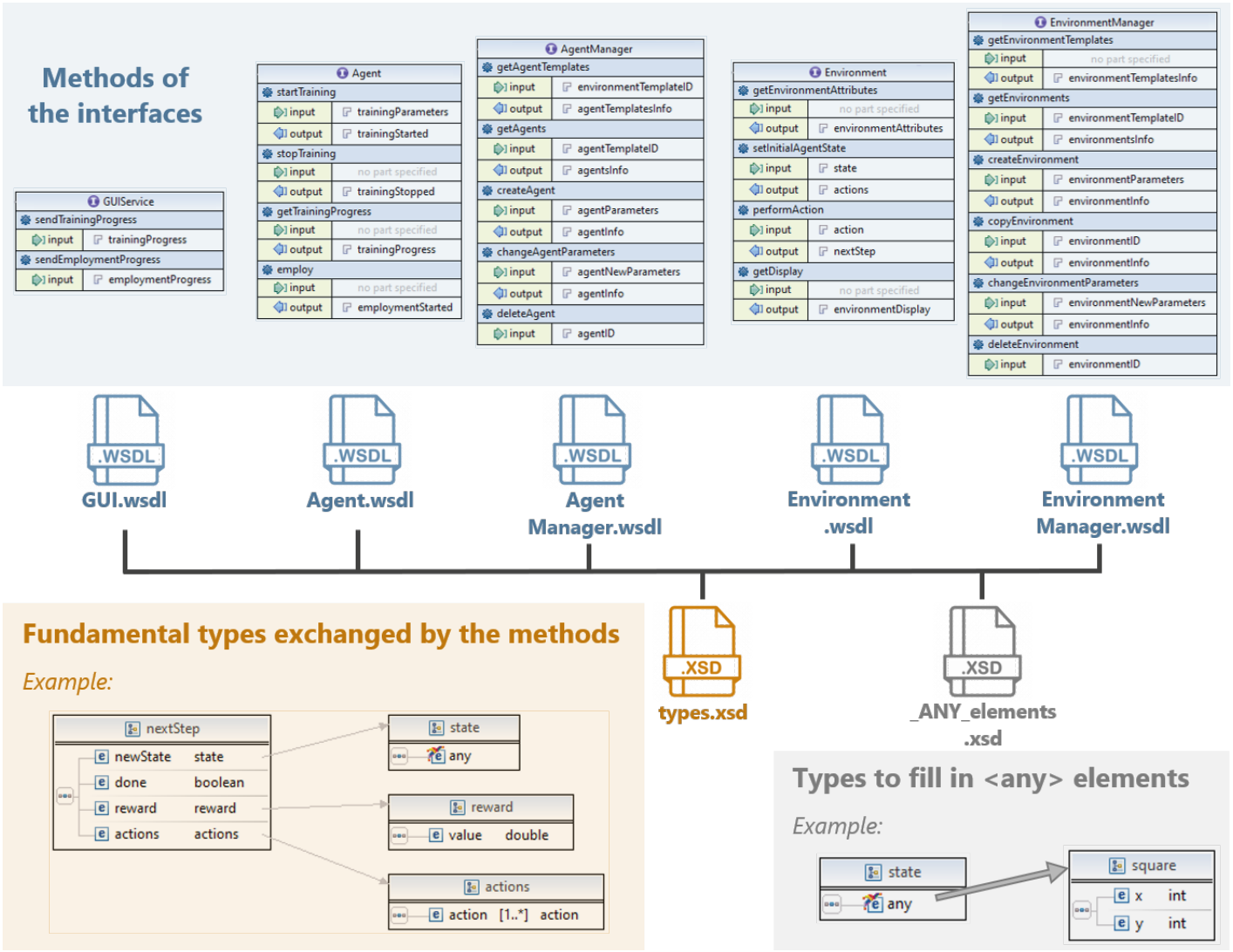



Fig. 3.3.: WSDL and XSD files used for the definition of the Reinforcement Learning Service interface

For example, the states of an environment can have various forms, e.g. being simply coordinates for a simple maze game (without cheese), or even being images when using Deep Q-Learning for a more complex video game. Therefore, the fundamental type state cannot be precisely defined, and it is set with an <any> element, see Figure 3.4. So, in the *_ANY_elements.xsd* file, the developer will have to define their own type describing a state and this type will be used to fill in the <any> element of a state type.

In the case of a mouse escaping a maze problem without cheese, coordinates can be used. A type named *tuple* can then be defined by the developer in the *_ANY_elements.xsd* file in order to complete the state type. Thus, in that case, the <any> element of a state is a *tuple* (`x,y`) with `x` and `y` of type *int*, see Figure 3.4.

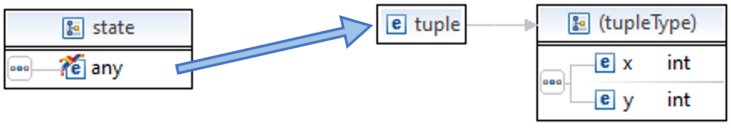



Fig. 3.4.: XSD <any> type example with the type *state*

Finally, from the WSDL and XSD files, Java code of the service interfaces can be generated so that the Java classes and packages of the Reinforcement Learning Service can be stored into a JAR file (Java ARchive file). In the created Java classes of the GUI, the Agent, the AgentManager, the Environment and the EnvironmentManager, the business code for reinforcement learning problems can then be implemented. A use case is presented in chapter 4.

# 4. Example implementation

In order to prove the concept of the Reinforcement Learning Service interface and to illustrate how it works, I have implemented a simple example. The latest version of the example is presented in this chapter.

Recall that from the WSDL files, Java classes are created for each entity of the Reinforcement Learning Service (for the GUI, the Agent, the AgentManager, the Environment and the EnvironmentManager). These Java classes, which contain the service methods presented previously, must be completed to meet the needs of the problem to be solved.

The ultimate goal would be to be able to create a Reinforcement Learning Service from the Reinforcement Learning Service interface provided by RuDi, by having a developer implement only Java classes of environment and agent templates, as well as define their own <any> elements in the *_ANY_elements.xsd* file. The template files and *_ANY_elements.xsd* file will then only need to be imported into the Reinforcement Learning Service, so that an end-user can create, train, and employ agents in user-defined environments.

During my internship, the Managers were not tested, and the notion of templates was not truly used. However, the example below operates a Q-Learning procedure that illustrates how communication between the GUI, an Agent and an Environment work.

## 4.1. Presentation of the use case

Consider once again the example of the mouse escaping a maze. The situation is given in Figure 4.1. A mouse moves in a maze containing ten squares, nine inner squares, and an additional square $(3, 0)$ representing the outside. The mouse starts its mission from one of the inner squares of the maze, for instance from the square $(1, 3)$. Without going through the walls, the mouse can then move down, up, right or left. Also, there are some pieces of cheese on another inner square, for example 3 pieces of cheese on the square $(2, 3)$ in Figure 4.1.

The mission of our agent, the mouse, is to escape the maze. This mission is defined with the following rewards: $-1$ if the mouse is still in the maze, $+1$ per piece of cheese and $+10$ if it escapes the maze (a piece of cheese can be eaten only once).

The environment is a maze containing a mouse on one square and pieces of cheese on another square. This environment is small and not very complex, so the states can be enumerated and therefore Q-Learning will be used.

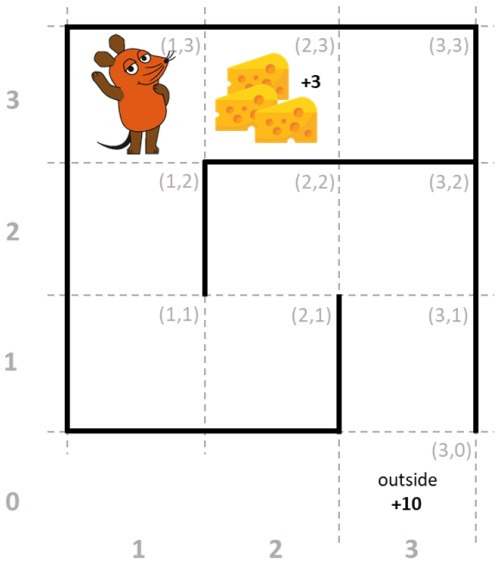



Fig. 4.1.: Use case of the mouse escaping a maze

Recall that a state of an environment has to describe it at a given time $t$. Therefore, in the case of our maze, a state has to contain information about the agent's current position as well as information about the presence or absence of the pieces of cheese, i.e. whether they have already be eaten by the mouse or not.

Notice that it is very important to distinguish the situation where the mouse is on a square and the cheese has not been eaten from the situation where the mouse is on the same square but the cheese has already been eaten. Because at the end of the Q-Learning algorithm, the agent will have learned to choose a single action from each state. Thus, when the mouse is on square $(1, 3)$, in order to choose once to go to the right from this square when

there is still cheese in the maze, and to choose another time to go down because there is no more reason to go to the right since there is no more cheese, two different states must be used.

Finally, the mouse is only allowed to make 12 moves during a mission, i.e. whether it reaches the outside of the maze or not, after the $12^{th}$ action the mission is over. This restriction of the number of movements is important to control the duration of the episodes and to avoid having an infinite episode in which an agent would only go back and forth. If the mouse has reached the outside within 12 moves, then the mission has ended successfully, otherwise the mouse has not managed to escape. Recall that this information about the success or not of the mission will not be directly known by the agent. The rewards, which define the mission, are the only information about the quality of its behaviour that the agent obtains. And this is enough for it to learn about the environment, as well as being finally able to learn to almost always fulfill its mission successfully.

## 4.2. Completing the types with <any> elements

First of all, in order to create a RuDi Reinforcement Learning Service, the RuDi development environment is required, as for the creation of the interfaces of the Reinforcement Learning Service, I used RuDi version 3.2.0.1. The XSD and Java programming work is then to be done.

The first thing to do in order to implement this mouse escaping a maze example, is to create the XSD types that will be filled in the <any> elements of some incomplete types. Here, the *State*, *EnvironmentDisplay* and *EnvironmentAttributes* new types need to be completed, and written in the *_ANY_elements.xsd* file, see Figure 3.3.

A *State* type only contains one <any> element. As explained in the previous section, for this use case, a state can be described by the coordinates of the current mouse position and by a boolean indicating whether or not the cheese is still present in the maze. So, the developer creates in the *_ANY_elements.xsd* file a type named *SquareAndCheesePresence* that will fill in the *State* type. A *SquareAndCheesePresence* type contains a Square element and a boolean called *CheesePresent*. The *Square* type is itself composed of the abscissa and the ordinate of a square of the maze.

For the *EnvironmentDisplay* type, a simple String type is created to be able to send a string of characters representing the environment.

The *EnvironmentAttributes* type imperatively contains the initial states, as well as the final success states of the environment. It also contains an <any> element that can be used if necessary. The *EnvironmentAttributes* type is returned to an agent when it requests the attributes of its environment using the *getEnvironmentAttributes()* method. As Q-Learning will be used here, the agent needs some information to be able to create a Q-table, namely the list of all existing states of the environment, and the list of actions available from each state. Therefore, the types *AvailableActionsFromState* and *AvailableActionsFromStateList* are created by the developer in the *_ANY_elements.xsd* file. Figure 4.2 shows design displays of those new types.

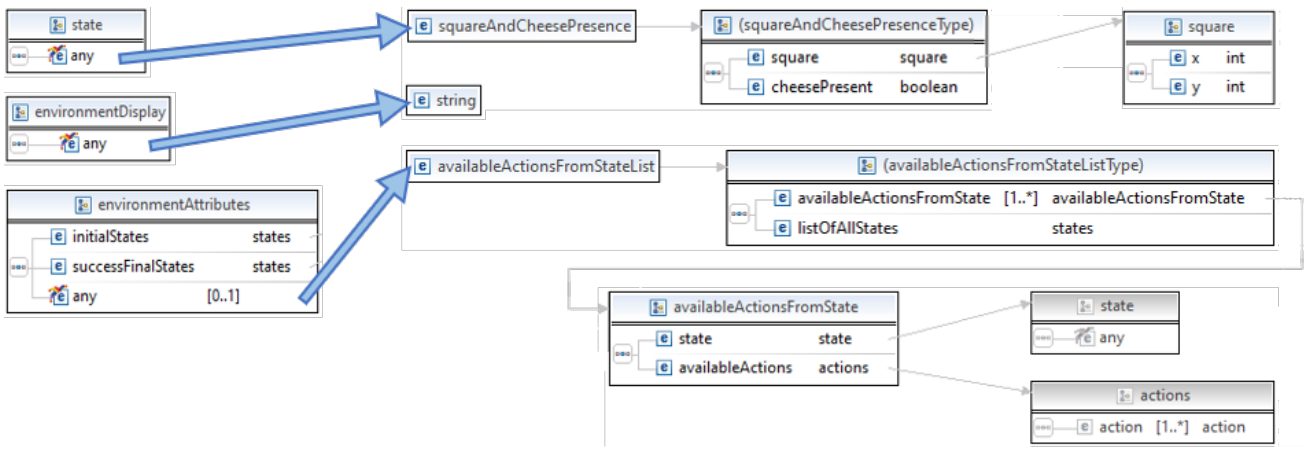



Fig. 4.2.: Types created to fill in <any> elements

Once the *_ANY_ elements.xsd* file was completed, it is used to generate the Java classes for the newly created types.

## 4.3. Business code implementation

During my internship I did not implement any code for the AgentManger and the EnvironmentManger, because this was not part of my work plan and because it was not necessary to prove the concept of the service. So the business code part of the Reinforcement Learning Service is supposed to evolve. Ultimately, in order to create a Reinforcement Learning Service, after defining the <any> elements, a developer will only have to write Java classes for agent and environment templates.

For now, as the AgentManager and the EnvironmentManager were not used in this example, the agent's and the environment's parametrization are directly stored in the agent and the environment implementations. Three repositories contain the Java files used for the business code:

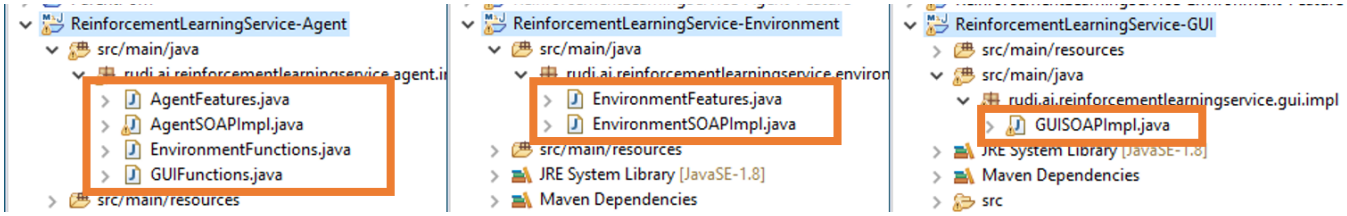



Fig. 4.3.: Java files used for the implementation

The methods of the Reinforcement Learning Service interfaces are implemented in the *GUISOAPImpl.java*, *AgentSOAPImpl.java* and *EnvironmentSOAPImpl.java* files.

For both agent and environment, sort of template Java classes were created in this example and named *AgentFeatures.java* and *EnvironmentFeatures.java*. These two features Java classes contain methods to get access to information about the agent or the environment, as well as some methods to be able to modify that information (e.g. in the *EnvironmentFeatures.java*, there is a variable containing the current state that will be modified every time an action is performed in the environment).

Also, in order to try out different environment configurations and to adapt the agent parameters to the environment configuration, two very basic user interfaces for the Agent and the Environment have been created in this use case (see Figure 4.4). Using these two basic user interfaces can be seen as managing an agent template and an environment template. Remember, however, that ultimately agent and environment templates should be managed by the AgentManager and the EnvironmentManager and controlled directly from the GUI service.

The rest of the reinforcement learning training and the employment process is controlled from the GUI service, as it is ultimately supposed to be. A very basic user interface was implemented to perform requests from the GUI service to the Agent and the Environment services, see Figure 4.4.

In Figure 4.4, once an environment configuration and the parameters of the agent were chosen, the buttons from the GUI service can be used to deal with a reinforcement learning training procedure.

With the "Display" button, the *getDisplay()* method is called on the environment to get a view of the environment. Recall that a String type was created in the *_ANY_ elements.xsd* file to fill in the *EnvironmentDisplay* type, and that finally such a string is returned here when calling the *getDisplay()* method.

With the "Start training" button, the *startTraining()* method is called on the agent with as parameters the number of episodes that is entered in the "Number of episodes" text field. Here, this method thus starts a training procedure for the agent with 100 episodes.

With the "Training result" button, the *getTrainingProgress()* method is called on the agent. In particular, the Q-table is returned by this method.

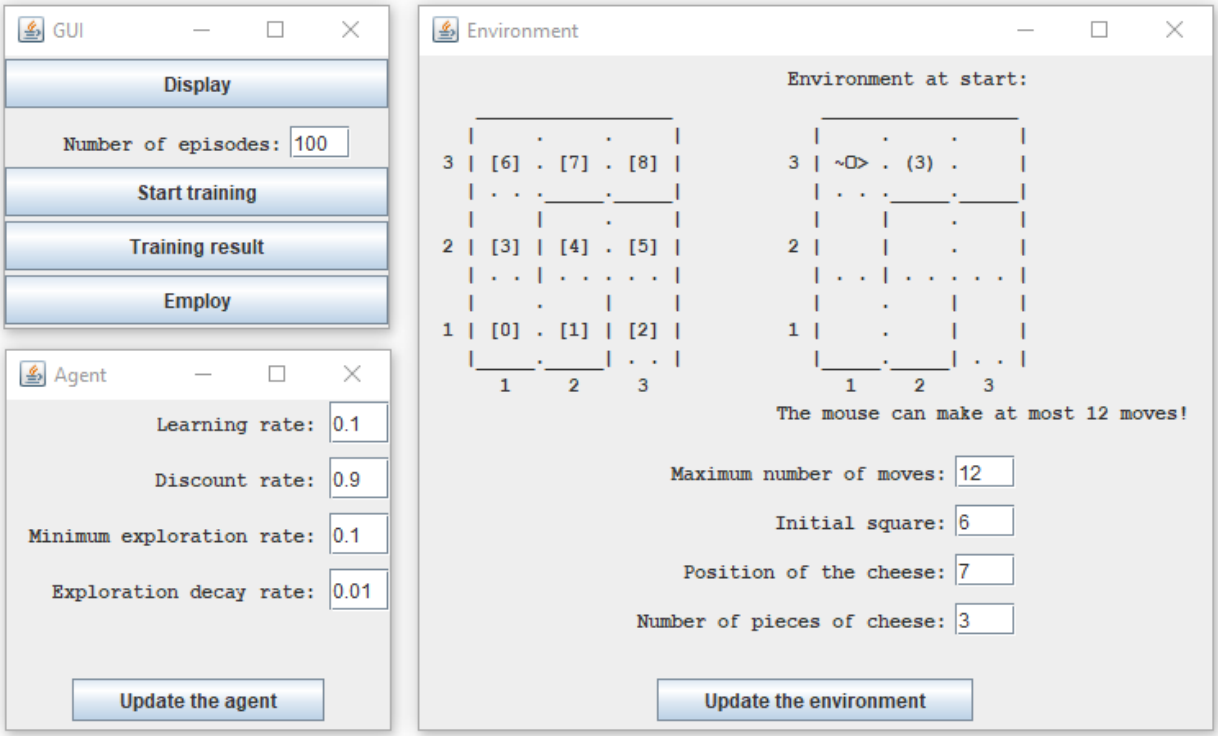



Fig. 4.4.: Simple user interfaces to perform requests from the GUI

Finally, With the "Employ" button, the *employ()* method is called on the trained agent to make use of the strategy learned by the agent, i.e. the mouse, in order to escape the maze.

Once the business code has been written, it needs to be installed and deployed in so-called Karaf containers in order to create the Agent, Environment and GUI services. Figure 4.5 presents the launching of the three services. The arrows illustrate the communications that take place between these services to handle a reinforcement learning problem.

## 4.4. Rendering the created Reinforcement Learning Service

Once the three containers were started (the Agent, the Environment, and the GUI), the three simple interfaces presented in Figure 4.4 should appear. The following are the steps to run a resolution to this reinforcement learning problem:

1. One can chose the environment configuration as well as the agent parameters by changing the values in the different text fields and then clicking on "Update the environment" and/or "Update the agent" for the updates to take effect.
2. Then, using the "Display" button of the GUI service one can get a display of the configured environment in the shell of GUI service.
3. The training can be launched with the "Start training" button and the training progress will be shown in the shell of the GUI service.
4. Once the training ended, the Q-table learned by the agent can be displayed by using the "Training result" button.
5. Finally, one can click on the "Employ" button to obtain the list of actions to perform using the strategy learned by the agent.

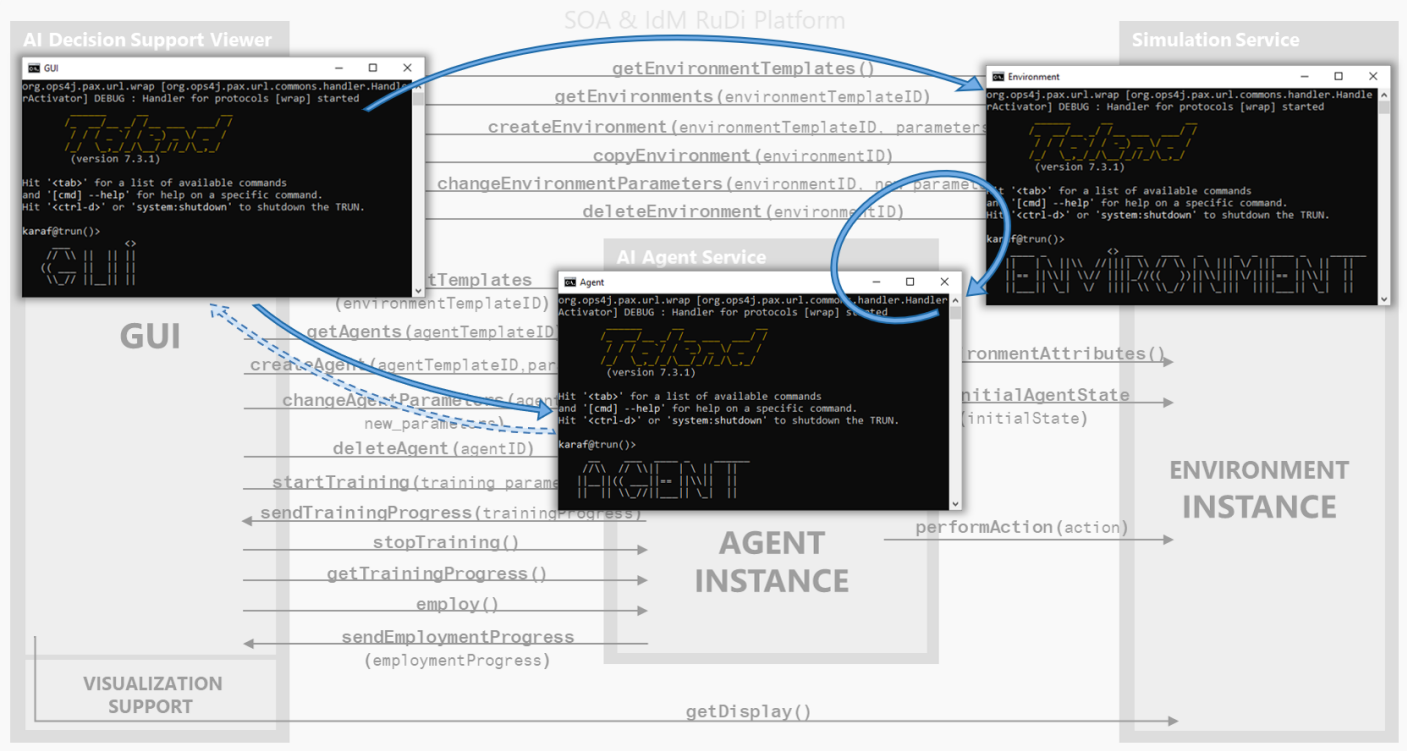



Fig. 4.5.: The three Karaf containers of the three services (which were defined in the WSDL files)

By using appropriate agent and training parameters, this use case should work nicely for every maze configuration. As presented in Figure 4.4 and in Figure 4.6, one can for instance use the following settings:

- learning rate: 0.1
- discount rate: 0.9
- minimum exploration rate: 0.1
- exploration decay rate: 0.01
- number of episodes: 100
- maximum number of moves: 12
- initial square: 6
- position of the cheese: 7
- number of pieces of cheese: 3

Here are some comments on the settings proposed above.

This maze problem is not very complicated, one can easily determine the optimal strategy by hand. Here for example, if the mouse is in position 6 (square $(1, 3)$), and if three pieces of cheese are in position 7 (square $(2, 3)$) then the mouse should follow the following path to maximize the rewards: right, left, down, down, right, up, right, down, down. This path should be found by the agent because even if getting the cheese lengthens the way out, it increases the cumulative rewards (being still in the maze gives a $-1$ reward but getting 3 pieces of cheese gives a $+3$ reward, so even with the small round trip, taking the pieces of cheese is advantageous). And after eating the cheese, the mouse leaves the maze as directly as possible to limit the number of $-1$ rewards received and to finally get the $+10$ reward for escaping the maze.

Thus, in the end, the mouse will need only 9 moves to complete the mission. A maximum number of moves of 12 is used here to make sure that the mouse will be able to explore up to the outside state while limiting the number of round trips it can make during an episode.

As already mentioned, the number of cheeses was chosen so that the mouse would have an interest in taking these three pieces of cheese even if it wasted some time.

The learning rate was set to 0.1 so that the update of the Q-values will not be too fast (this seems appropriate for such a small environment in which the agent will be able to visit each state many times).

The discount rate allows for more or less consideration of long-term rewards. Recall that the closer this number is to 1, the more importance is given to the rewards that the agent expects to obtain in a far future. Here the mouse will very often receive a reward of $-1$, and will receive the reward of $+10$ only if it goes to the end of the maze, so a discount rate of 0.9 seems consistent.

For the exploration rate, the minimum was set to 0.1 so that the agent will still explore a bit when it reaches this minimum exploration value. And the exploration decay rate was set to 0.01 so that the exploration part remains important during a good first half of the training.

Finally, a number of 100 episodes (which is not very large for a reinforcement learning training but sufficient for this small environment) was set to ensure that the agent will have enough opportunities to explore each state of the environment several times.

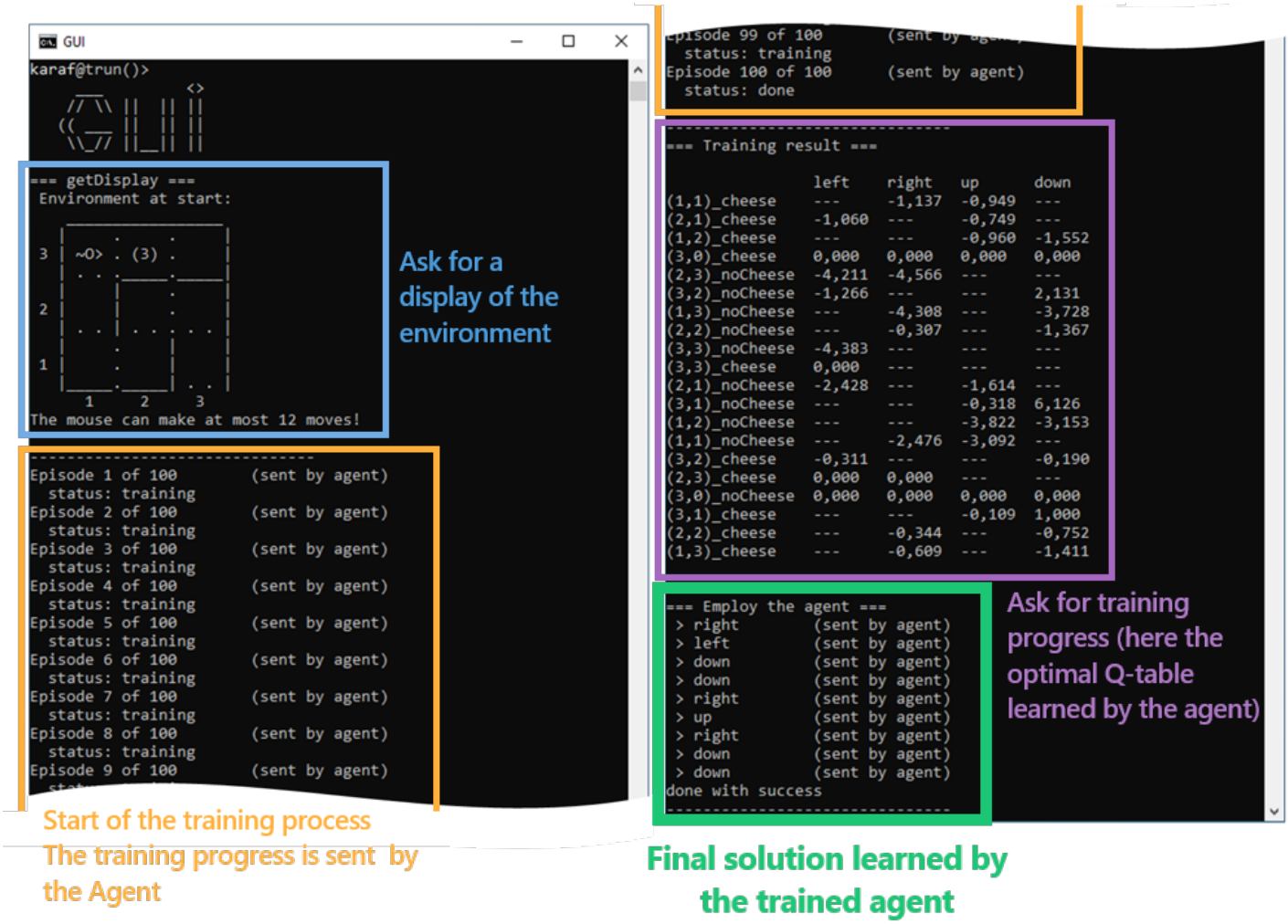



Fig. 4.6.: Reinforcement Learning Service in action: GUI service

In Figure 4.6, a display of the environment is first requested. Then, a training procedure with 100 episodes is run. And finally, the learned Q-table is requested as well as the actions to be taken to follow the strategy learned during the training. By following the list of actions returned by the trained agent, the mouse actually succeeds in getting intelligently out of the maze!

To see some information from the inside of the methods called from the GUI service see Appendix A.4.

# Conclusion

The objective of this internship was to demonstrate the viability of a Reinforcement Learning Service while documenting both the concept and the interface of the service created. Finally, an example was used to demonstrate how the service works.

I have actually successfully defined and created a first version of this service, thus demonstrating the concept of a generic Reinforcement Learning Service. Future extensions or enhancements are planned. Among others, the AgentManager and the EnvironmentManager have not been tested yet, consequently, the concept of templates and their management needs to be tested in the future. Recall again that for the Reinforcement Learning Service, the ultimate goal is that the developer only has to code agent and environment templates, as well as to define his own types to fill in the <any> elements. There are also some improvements to foresee in the implementation of the service, in particular to make the training process run separately from the rest of the code execution. Furthermore, in the quite simple use case presented above, a Q-Learning algorithm was successfully implemented. In the future, one should also try to implement a Deep Q-Learning algorithm on a more complex environment (e.g. the Little Brick Out game).

Personally, it was very exciting to discover the theory of reinforcement learning, which is one of the three parts of machine learning. It fits perfectly into my applied mathematics engineering studies because it was a subject I had not studied before, but for which I had all the necessary theoretically background and knowledge of scientific tools to immerse and to understand. It was a very rewarding exercise, too, that I had to present this theory by popularizing it several times during the internship by adapting my vocabulary and in this way to ensure a common understanding with all stakeholders.

Thanks to the knowledge of the applied mathematics department of INSA de Rennes and the ability to learn to learn gained at INSA, the internship went well, and the results obtained were very satisfactory and useful for the RuDi team.

Moreover, this internship took place within the Erasmus+ program, thus allowing me to get professional experience in a foreign country. Both personally, being in another country with a new culture and new people to discover, and professionally, working in a foreign cultural and linguistic environment, this internship was a rich experience in terms of developing my engineering as well as my interpersonal skills.

Thank you for your attention and for reading this report.

# Glossary

**business code**: The lines of programming code containing the most important parts of the implementation, for example the Java implementation of the reinforcement learning algorithm.

**IABG**: for IndustrieAnlagen-BetriebsGesellschaft (mbH). The company where I completed my internship, see https://www.iabg.de/en/ for more information.

**interface**: A generic word representing the boundary between two software or services, and the communication procedure to be used from one service to access the functions of another. In particular, the methods and types to be exchanged are described in an interface.

**Java**: Object-oriented programming language used for the implementation in this project.

**JAR**: for Java ARchive. It is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution.

**MDP**: for Markov Decision Process. It is a probabilistic model that is widely used in artificial intelligence.

**Q**: In Q-Learning, Q-value, DQN, etc. Q stands for "quality (function)"

**RL**: for Reinforcement Learning

**RuDi**: for **R**eferenz**u**mgebung **Di**enste, in english reference environment services. Name of the project I was involved in during my internship.

**SOA**: for Service-Oriented Architecture. It is a set of software design principles and methodologies.

**SOAP**: for Simple Object Access Protocol. It is a messaging protocol specification for exchanging structured information in the implementation of web services, like the one created during this internship. It uses XML (see below) for its message format.

**WSDL**: for Web Services Description Language. It is an XML-based interface description language that is used for describing the functionality offered by a web service using SOAP, for instance.

**XML**: for eXtensible Markup Language. It is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

**XSD**: for XML Schema Definition. It specifies how to formally describe the elements in an XML file.

# Bibliography

## Reinforcement learning tutorial, series and book


[jav18] javatpoint. *Reinforcement Learning Tutorial*. 2018. URL: https://www.javatpoint.com/reinforcement-learning.

[dee18] deeplizard. *Reinforcement Learning - Goal Oriented Intelligence*. 2018. URL: https://deeplizard.com/learn/video/nyjbcRQ-uQ8.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction - second edition*. 2018.


## Discussions of the differences between on-policy and off-policy reinforcement learning algorithms


[Sag20] Ram Sagar. *On-Policy VS Off-Policy Reinforcement Learning*. 2020. URL: https://analyticsindiamag.com/reinforcement-learning-policy/.

[Sta20] StackExchange. *What is the difference between off-policy and on-policy learning?* 2020. URL: https://stats.stackexchange.com/questions/184657/what-is-the-difference-between-off-policy-and-on-policy-learning.

[Mao19] Lei Mao. *Reinforcement Learning: An Introduction - second edition*. 2019. URL: https://leimao.github.io/blog/RL-On-Policy-VS-Off-Policy/.

# A. Appendices

## A.1. More about reinforcement learning algorithms

### A.1.1. SARSA algorithm

*SARSA* stands for State Action Reward State Action because this algorithm directly uses the quintuple $(s, a, r, s', a')$, where $s$ is the original state, $a$ is the original action, $r$ is the reward observed while taking action $a$ from state $s$, and $s'$ and $a'$ are the new state and the next action. Just as with Q-Learning, the goal of SARSA is to calculate the optimal Q-function for all state-action pairs but while following the current selected policy $\pi$, i.e. using the current constructed Q-values.

The main difference between Q-Learning and SARSA algorithms is that unlike Q-Learning, the maximum reward for the next state is not required for updating the Q-value in the Q-table. In SARSA, the new action and reward are selected using the same policy which has determined the original action. For calculating the new Q-value for state-action pair $(s, a)$, SARSA directly uses the known Q-value of the next best state-action pair $(s', a')$ selected. With $r$ the reward received after taking action $a$ from state $s$, the formula for the update is therefore:

$$q^{\text{new}}(s,a) = (1-\beta)\underbrace{q(s,a)}_{\text{old value}} + \beta\underbrace{\left(r + \gamma q(s',a')\right)}_{\text{learned value}}$$

where unlike for Q-Learning, the action $a'$ was previously determined.

Here is pseudo-code for this algorithm:

**Algorithm 3:** SARSA algorithm

```
Initialize all Q-values in the Q-table to 0 (or randomly except for the final states)
foreach episode do
    Choose an initial state s
    Choose an action a from state s using a policy derived from the Q-table (with ε-greedy strategy)
    foreach time-step in each episode do
        Take action a and observe the reward r and the next state s'
        Choose an action a' from state s' with the ε-greedy strategy
        Update the Q-value function, i.e. the Q-table, using
                                q^new(s,a) = (1-β)q(s,a) + β(r + γq(s',a'))
        This formula will, overtime, make the Q-value function converge to the right-hand side of the
        Bellman equation.
        Set s to the value s' and a to the value of a'
```

### A.1.2. On-policy vs off-policy

The discussion below is based on [Sag20], [Sta20] and [Mao19].

As seen above, the ideas behind SARSA and Q-Learning are very similar, the main difference being the way the Q-function is updated. While SARSA uses actions determined from the current policy for learning the Q-function,

Q-Learning doesn't care about the current policy for the choice of the next action and thus uses a different policy for updating the Q-function. In SARSA the action $a'$ is taken according to the current policy, whereas in Q-Learning $a'$ are all the actions that are probed in state $s'$. SARSA is therefore called an on-policy algorithm while Q-Learning is said to be an off-policy method.

The difference between on- and off-policy algorithms can also be explained introducing the notion of behaviour policy and update policy. The update policy is the policy the agent learns and that is intended to become the optimal policy, while the behaviour policy is how the agent behaves during the learning process.

In Q-Learning, the agent learns optimal policy using a kind of absolute greedy policy, i.e. only with exploitation, and behaves using an $\varepsilon$-greedy policy. Because the update policy is different from the behaviour policy, Q-Learning is therefore off-policy.

In SARSA however, the agent learns optimal policy and behaves using the same $\varepsilon$-greedy policy. As update and behaviour policies are the same, SARSA is on-policy.

Notice thus that the distinction would disappear if the current policy was an absolute greedy policy. However, as discussed earlier, such an agent would not be efficient since it never explores.

For a general idea, on-policy reinforcement learning algorithms seem to be more stable during learning than off-policy methods, because the policy used for the behaviour and to get the optimal policy is the same. With off-policy, the behaviour policy may be very different from the update policy so that learning is more unstable, and exploration is less controlled. As a result, off-policy therefore allows a kind of continuous exploration that makes it more likely to learn an optimal policy since it will explore some paths that an on-policy algorithm may not as easily try. On-policy reinforcement learning being more stable is thus also more likely to learn a sub-optimal policy.

One therefore has to notice one thing when comparing those two policies: as off-policy shows more fluctuation than on-policy, the agent is less certain about what it knows from the environment and thus the overall reward might be lower for off-policy algorithms even on a more optimal path than the one found by an on-policy. By comparing on-policy with off-policy algorithms, using the return is therefore not necessarily relevant.

Nevertheless, after an infinite learning time, both on- and off-policy are supposed to converge to the same optimal policy and depending on the problem one can have a good surprise in terms of computation time with one or the other policy. It seems however that off-policy algorithms are more commonly used in applications, as is the case with Q-Learning and Deep Q-Learning.

## A.2. Deep Q-Learning

The following subsections give more details and explanations about effective Deep Q-Learning.

### A.2.1. Experience replay and replay memory

First, a so-called *experience* of an agent is simply a tuple $e = (s, a, r, s')$, containing the state of environment $s$, the action $a$ taken from state $s$, the reward $r$ given to the agent as the result of the previous state-action pair $(s, a)$, and the next state $s'$. Notice that the state $s$ can be referred as the starting state of the experience $e$.

Also notice that, unlike in the simplified Deep Q-Learning algorithm presented above, when updating the weights of a neural network, a sample of experiences is usually used rather than a single experience. One main reason for this, is that one specific unlikely experience could update the weights incorrectly, which is much less likely to happen if several experiences are used for updating.

The technique of *experience replay* is often used in DQN and allows Deep Q-Learning to effectively work because it solves some correlation issues. To use experience replay, during the training of the network, the agent's experiences are stored at each time-step in a data set called the *replay memory*. But for reasons of computing time and storage space, the replay memory is set to some finite size limit $M$, and will therefore only store the last $M$ experiences of the agent (depending on the problem, the last experiences can even be as many as $M = 100000$). This replay memory data set is what will be randomly sampled to train the network, and it turns out that using only the near past $M$ experiences is far enough for its training.

So, experience replay is the act of training the DQN on random samples from the replay memory, rather than just providing the network with the sequential experiences as they occured in the environment. In particular, replay memory breaks the correlation between consecutive experiences samples. If the network only learned from consecutive samples of experiences as they occurred sequentially in the environment, the samples would be highly correlated and would therefore lead to inefficient learning. Taking random samples from replay memory breaks this correlation.

### A.2.2. The policy network

In the simplified version of the Deep Q-Learning algorithm presented earlier in the report, only one deep neural network was used. It actually turns out that two neural networks can be used to improve the training procedure. The neural network that was previously referred as DQN is known as the *policy network* because it has to reflect the optimal policy by mimicking the optimal Q-function. In the Deep Q-Learning algorithm, what we finally want to learn are the weights of this policy network. The second network, called the *target network*, will be used as an approximation of the targeted optimal Q-function to calculate loss, so that the policy network's weights can be updated. The target network is presented in the next subsection.

Here is first a recap of how the policy network is updated iteration after iteration.

After storing an experience in replay memory, a random batch of experiences is drawn from that replay memory in order to train the policy network. The starting states of the experiences of the batch are first preprocessed (grayscale conversion, cropping, scaling, etc.) before being given one after another as input state into the policy network. They then forward propagate through the network and finally, as presented previously, the model outputs an estimated Q-value for each possible action for each given input state of the batch. So, for a given experience $e = (s, a_3, r, s')$, the state $s$ enters the policy network that then returns the Q-values $q(s, a_i)$ for every possible action $a_i \in \mathcal{A}$, including the action $a_3$ of the experience $e$.

Then, the loss is calculated. For each experience, this is done by comparing the Q-value outputted by the network for the action in the experience tuple, with the corresponding optimal Q-value, or target Q-value, which is estimated thanks to the right-hand side of the Bellman equation. So, taking again the example of the experience

$e = (s, a_3, r, s')$, the Q-value given by the network for $q(s, a_3)$ will be compared with the target Q-value $q_*(s, a_3)$ that will be estimated thanks to the right-hand side of the Bellman equation. As for Q-Learning, for a given state-action pair $(s, a)$, the loss is thus calculated by subtracting the outputted Q-value $q(s, a)$ from the optimal Q-value $q_*(s, a)$:

$$loss = q_*(s, a) - q(s, a) = \left(r + \gamma \max_{a'} \left(q_*(s', a')\right)\right) - q(s, a)$$

In order to estimate the max-term $\max_{a'} \left(q_*(s', a')\right)$, rather than using the policy network, the target network will be used, as explained in the next subsection.

Once we were able to calculate the loss for every starting state of the batch, the weights in the policy network are updated to minimize the loss as is typically done in neural networks. In this case, minimizing the loss means that we are correcting the policy network output Q-values for each state-action pair in order to approximate the target Q-values given by the Bellman equation.

Up to this point, the policy network was updated with a single batch for just a few time-steps. We then move to the next batch in the episode and repeat this process again and again until the end of the episode is reached. At that point, a new episode starts, and the same operations are done over and over until the loss was sufficiently minimized or a maximum number of episodes was reached.

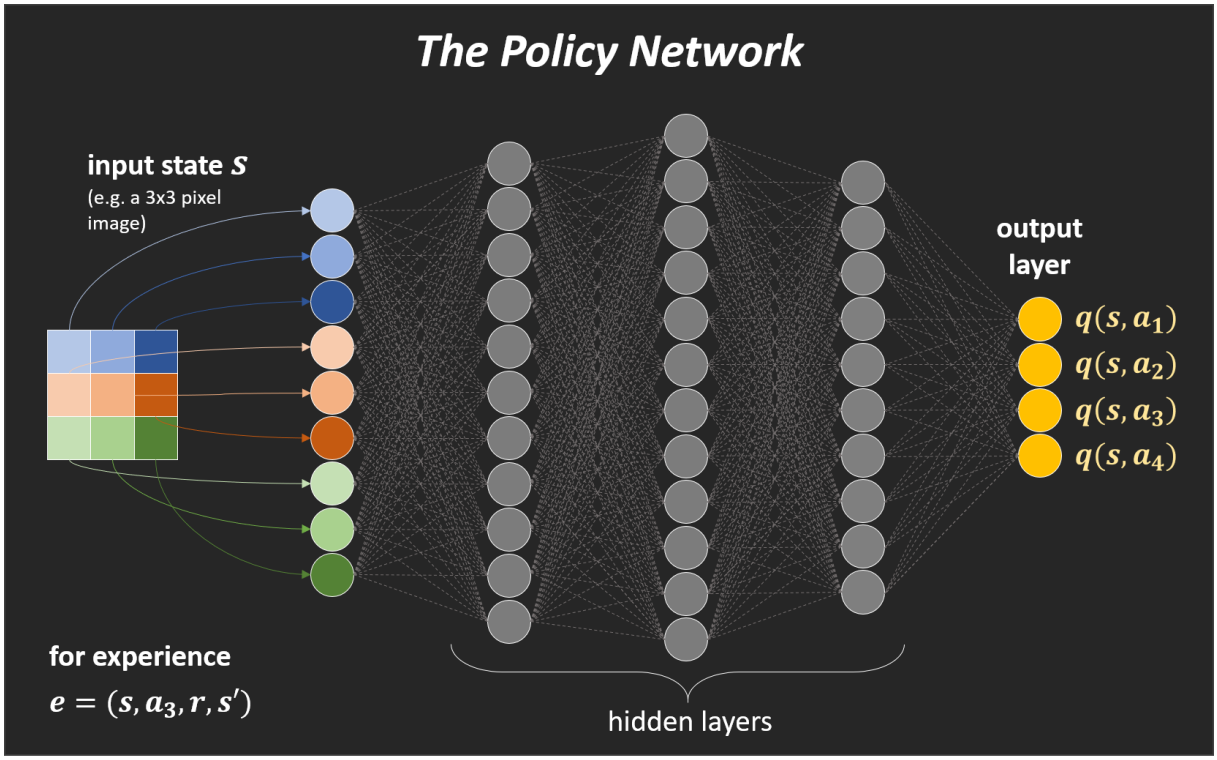



Fig. A.1.: Policy network

### A.2.3. The target network

For the loss calculation, to get the max term $\max_{a'} \left(q_*(s', a')\right)$ coming from the Bellman equation that is used to get the optimal Q-values, another deep neural network is used. The target network is in fact not a completely new network, it is just an earlier copy of the policy network. In particular, this target network won't need to be trained. Its only usage will be to provide the Q-values for all state-action pairs at an earlier fixed time $t$.

Once for an experience $e = (s, a, r, s')$, the policy network returned a value for $q(s, a)$, we need to compute the max term $\max_{a'} \left(q_*(s', a')\right)$ where $s'$ is the next state, so that the loss can be calculated and then the weights of the policy network can be updated. As presented in the simplified version of the Deep Q-Learning algorithm, to get $\max_{a'} \left(q_*(s', a')\right)$, the policy network could simply be used by giving it the state $s'$ as input and taking then the

maximum Q-value outputted by the network. In fact, this is exactly what is going to be done but taking advantage of the target network.

The problem of using a single network, the policy network, for both getting the updated Q-values and calculating the optimal Q-values using Bellman equation, is that the network is always updating its weights in order to converge to the optimal Q-function, so that this optimal Q-function calculated from the same network would therefore also change after each update. So, the Q-values would be updated with each iteration to move closer to the target Q-values, but the target Q-values would also be moving in the same direction. This may lead to instability and a kind of mouse-running-around-a-tree-chasing-its-own-tail problem.

The target network improves this problem considerably. As mentioned before, it is a clone of the policy network. Its weights are frozen with the original policy network's weights, and the weights in the target network are updated to the policy network's new weights every certain amount of time-steps $N$. This certain amount of time-steps $N$ is another hyperparameter that will have to be optimized. Generating the target Q-values using an older set of parameters like this adds a delay between the time an update to the Q-function is made and the time the update affects the target Q-values, thus making divergence or oscillations much more unlikely. Notice that with this, Deep Q-Learning is an off-policy method because the update policy (target network) and the behaviour policy (policy network) are not the same (see on-policy vs off-policy discussion in Appendix A.1).

So, for an experience $e = (s, a, r, s')$, the first pass with $s$ will occur with the policy network, while another pass with $s'$ through the target network will be used for the max-term computation. With this target network, the maximum Q-value $\max_{a'} \left(q_*(s', a')\right)$ for the next state $s'$ is obtained and by plugging this value into the Bellman equation, we finally can estimate the target Q-value $q_*(s, a)$ for the starting state $s$.

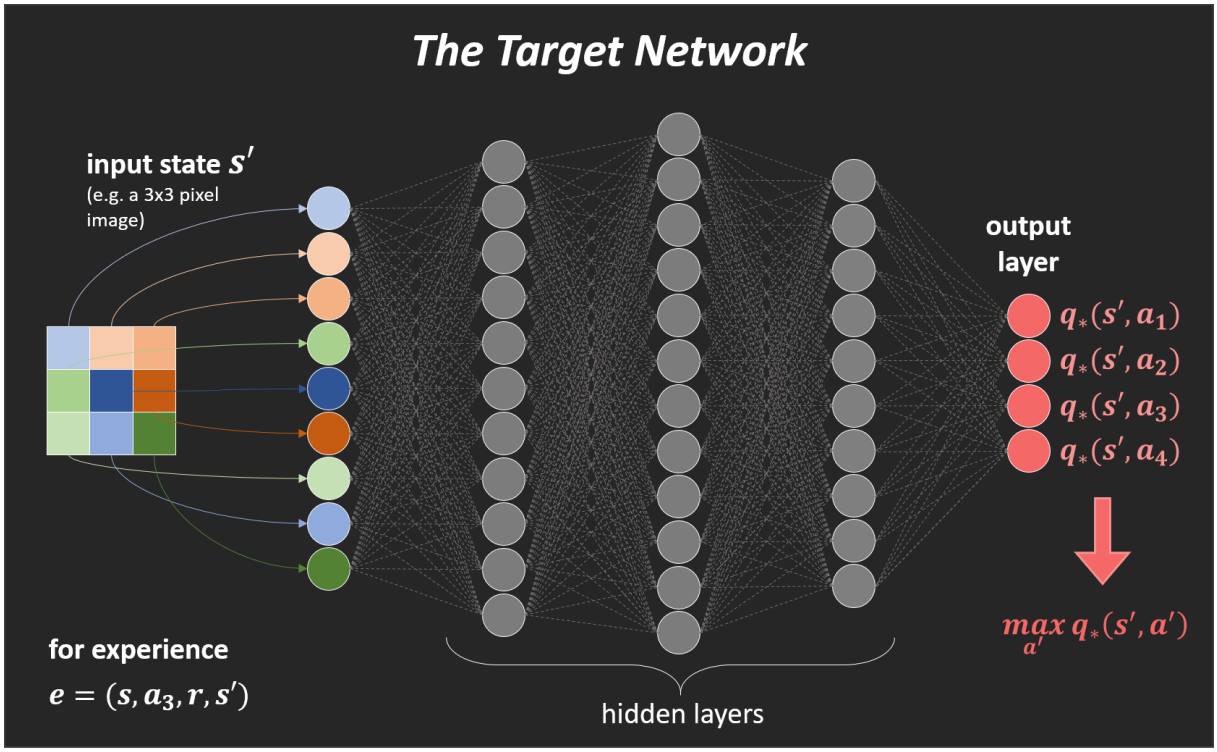



Fig. A.2.: Target network

### A.2.4. Deep Q-Learning algorithm

Here is a pseudo-code of Deep Q-Learning using replay memory and a target network:

**Algorithm 4:** Deep Q-Learning algorithm

Initialize replay memory capacity $M$
Initialize the policy network with random weights
Create the target network as a clone of the policy network
**foreach** *episode* **do**
    Choose an initial state $s$
    **foreach** *time-step in each episode* **do**
        Select an action $a$ from state $s$ (via $\epsilon$-greedy strategy with the policy network)
        Take action $a$ and observe the reward $r$ and the next state $s^{'}$
        Store the experience $(s, a, r, s^{'})$ in the replay memory.
        If there is no space left, this new experience replaces the oldest experience in the replay memory.
        Sample a random batch $(e_1, e_2, \cdots, e_m)$ from the replay memory, with $e_i = (s_i, a_i, r_i, s_i^{'})$ for each experience from 1 to $m \in [1, M)$
        Pass batch of preprocessed states $(s_1, s_2, \cdots, s_m)$ throw the policy network and get the Q-values $q(s_1, a_1), q(s_2, a_2), \cdots, q(s_m, a_m)$
        Calculate loss between output Q-values $q(s_1, a_1), q(s_2, a_2), \cdots, q(s_m, a_m)$ and target Q-values $q_*(s_1, a_1), q_*(s_2, a_2), \cdots, q_*(s_m, a_m)$
        The target Q-values are recovered by the passes throw the target network of the next states $(s_1^{'}, s_2^{'}, \cdots, s_m^{'})$ and using Bellman equation.
        Update the weights in the policy network
        After $N$ time-steps, weights in the target network are updated to the weights in the policy network.
        Set $s$ to the value $s^{'}$

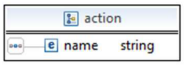
## A.3. Description of the types and methods of the Reinforcement Learning Service



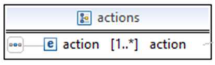
### Description of the Types

Below are the descriptions of all the types defined in the *types.xsd* file.

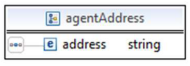
**Action**
An action only needs to be described by its name. The real action behind the name will be determined in the business code of an agent template.



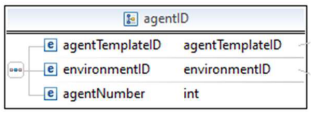

**Actions**
List of one or more actions.



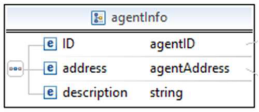

**AgentAddress**
Provider address or port where an agent instance implementation can be found.



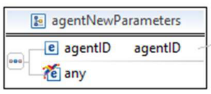

**AgentID**
The ID of an agent consists of the ID of the agent template it is based on, the ID of the environment it is built for, and an instance number.



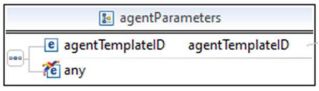

**AgentInfo**
The main information about an agent: its ID, at which adress it can be found, and a text description.




**AgentNewParameters**
Parameters to use to reconfigure the parameters of an existing agent. The ID of the agent to be updated is provided, as well as an <any> element containg all the other parameters of the agent (discount rate, exploration rate, neural network parameters, preprocessing parameters…).



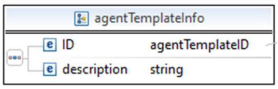

**AgentParameters**
Parameters to use to create an agent from an agent template. The ID of the agent template to base on is provided, as well as an <any> element because all the other parameters of the agent depend on the chosen Reinforcement Learning algorithm (discount rate, exploration rate, neural network parameters, preprocessing parameters…).




**AgentsInfo**
List of information of agents.


**AgentTemplateID**
The ID of an agent template consists of the ID of the environment template it was built for and its own name.


**AgentTemplateInfo**
Information about an agent template: its ID and a text description.

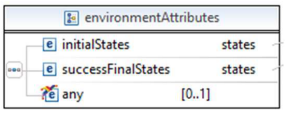



**AgentTemplatesInfo**
List of information of agent templates.

**EmploymentProgress**
Progress of an agent when being employed on its environment. It contains the next action that the agent will take according to the learned strategy, the status of the agent after taking that action ("working", "done with success" or "done but failed"), and it can contain a text message.

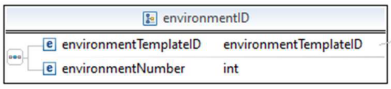

**EnvironmentAddress**
Provider address or port where an environment instance implementation can be found.


**EnvironmentAttributes**
Some attributes of an environment. This element contains the possible initial states for the agent, the final states in which the agent would have successfully fulfilled its mission, and it can contain other elements by using the <any> element. For instance, when using Q-Learning, it is useful to know the list of all the existing states and actions to create a Q-table.

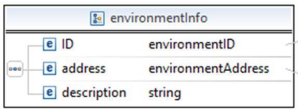



**EnvironmentDisplay**
Display of an environment. Depending on the problem, it might be simple text, an image, both, or something else.


**EnvironmentID**
The ID of an environment consists of the ID of the environment template it is based on and an instance number.

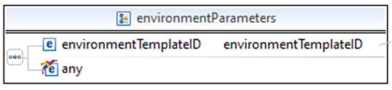



**EnvironmentInfo**
The information about an environment consists of its ID, at which adress it can be found, and a text description.



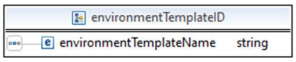

**EnvironmentNewParameters**
Parameters to use to update an environment. Therefore, the ID of the environment to update on is provided, as well as an <any> element containing all the other parameters of the environment (e.g. if it is a maze, there should be some elements to give the shape of the maze, the layout of the walls, the maximal number of moves allowed…).


**EnvironmentParameters**
Parameters to use to create an environment from an environment template. Therefore, the ID of the environment template to base on is provided, as well as an <any> element because all the other parameters depend on the environment itself (e.g. if it is a maze, there should be some elements to give the shape of the maze, the layout of the walls, the maximal number of moves allowed…). Notice that this type differs from the *EnvironmentNewParameters* type above, which contains the ID of an environment and not that of an environment template. *EnvironmentNewParameters* is used to update an existing environment, while *EnvironmentParameters* is used to create another environment instance.

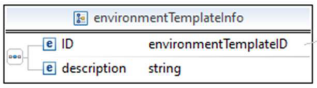


**EnvironmentsInfo**
List of information of environments.

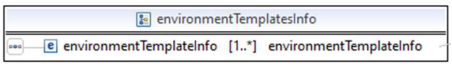

**EnvironmentTemplateID**
The ID of an environment template is simply the name of the environment template, that must of course be unique (e.g. “maze”).



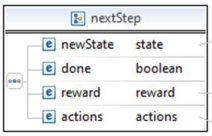

**EnvironmentTemplateInfo**
Information about an environment template: its ID and a text description.




**EnvironmentTemplatesInfo**
List of environment template information.




**NextStep**
Contains the elements returned by an environment to an agent after it has performed an action: the new state of the environment, a boolean that is true if the mission is done (either with or without success), the reward received for the previously taken action, and a list of the next actions that could be taken.



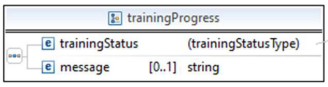

**Reward**
A reward simply contains its value (that can be positive or negative).


**State**
A state only contains an <any> element because depending on the environment, a state could for instance be coordinates, or an image, or anything else. For a given environment, the developer has to define a type in the *_ANY_elements.xsd* file and this type will be used inside the state.


**States**
List of states.


**TrainingParameters**
Parameters for the training procedure of an agent. It contains the number of episodes that will be performed, and can contain any other parameters depending on the Reinforcement Learning algorithm used.

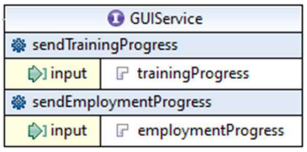

**TrainingProgress**
Progress of an agent when being trained on its environment. It contains the curent training status of the agent (“untrained”, “started”, “stopped” or “trained”), and it can contain a text message.




### Description of the Methods

Below are the descriptions of all the methods of the Reinforcement Learning Service interface.

## ➢ GUI




### sendTrainingProgress

*input:* **TrainingProgress** trainingProgress

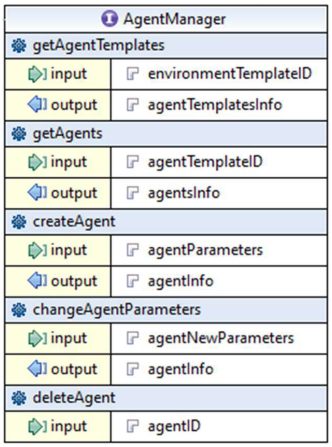
Used from an Agent. Sends the current training progress into the GUI (number of episodes done so far, observed cumulative reward, Q-table, whether the agent is in training or is already trained…).

### sendEmploymentProgress

*input:* **EmploymentProgress** employmentProgress

Used from an Agent. Sends the current employment progress into the GUI (whether the agent is still working or is done, the action that should be taken next by the agent…).

## ➢ AGENT MANAGER




### getAgentTemplates

*input:* **EnvironmentTemplateID** environmentTemplateID
*output:* **AgentTemplatesInfo** agentTemplatesInfo

Used from the GUI. For a given environment template, returns a list of information about existing agent templates.

### getAgents

*input:* **AgentTemplateID** agentTemplateID
*output:* **AgentsInfo** agentsInfo

Used from the GUI. For a given agent template, returns a list of information about created agents.

### createAgent

*input:* **AgentParameters** agentParameters
*output:* **AgentInfo** agentInfo

Used from the GUI. Instantiates an agent from the entered parameters, including the agent template to be used for the creation of this agent, as well as some fixed parameters of the Reinforcement Learning algorithm used (algorithm parameters, state preprocessing settings…). It returns information about the instantiated agent.

### changeAgentParameters

*input:* **AgentNewParameters** agentNewParameters
*output:* **AgentInfo** agentInfo

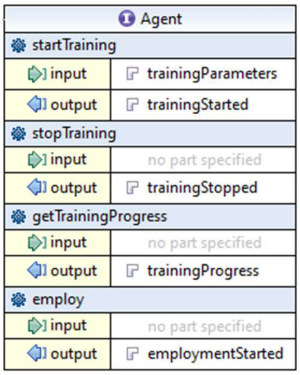
Used from the GUI. Updates the parameters of an existing agent with the entered new parameters, which also include the ID of the agent to be updated. It returns information about the modified agent.

### deleteAgent

*input:* **AgentID** agentID

Used from the GUI. Deletes an agent identified by its ID.

## ➢ AGENT INSTANCE




### startTraining

*input:* **TrainingParameters** trainingParameters
*output:* **boolean** trainingStarted

Used from the GUI. Starts the training for the agent with the entered training parameters (number of episodes, and optionally other parameters depending on the algorithm). It returns a boolean that is true if the training successfully started.

### stopTraining

*output:* **boolean** trainingStopped

Used from the GUI. Stops a training in progress and returns a boolean that is true if the training was successfully interrupted.

### getTrainingProgress

*output:* **TrainingProgress** trainingProgress

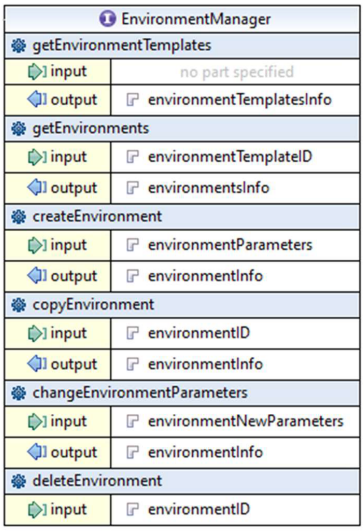
Used from the GUI. Returns information about the agent's current training (number of episodes done so far, observed cumulative reward, Q-table, whether the agent trained or not...).

### employ

*output:* **boolean** employmentStarted

Used from the GUI. Launchs the employment of the trained agent for a real purpose. It returns a boolean that is true if the employment successfully started.

## ➢ ENVIRONMENT MANAGER




### getEnvironmentTemplates

*output:* **EnvironmentTemplatesInfo** environmentTemplatesInfo

Used from the GUI. Returns a list of information about existing environment templates.

### getEnvironments

*input:* **EnvironmentTemplateID** environmentTemplateID
*output:* **EnvironmentsInfo** environmentsInfo

Used from the GUI. For a given environment template, returns a list of information about created environments.

### createEnvironment

*input:* **EnvironmentParameters** environmentParameters
*output:* **EnvironmentInfo** environmentInfo

Used from the GUI. Instantiates an environment from the entered parameters, including the environment template to be used for the creation of this environment, as well as some parameters that describe the environment (size of the environment, layout, choice of the initial

and final states, values of the rewards, maximum number of steps…). It returns information about the instantiated environment.

### copyEnvironment

*input:* **EnvironmentID** environmentID
*output:* **EnvironmentInfo** environmentInfo

Used from the GUI. Creates a copy of an existing environment and returns information about the newly instantiated environment.

### changeEnvironmentParameters

*input:* **EnvironmentNewParameters** environmentNewParameters
*output:* **EnvironmentInfo** environmentInfo

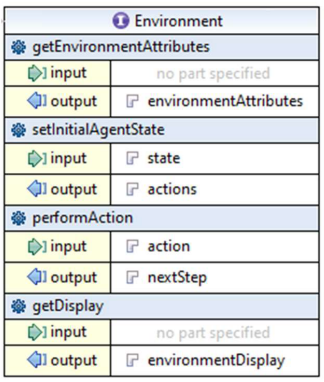
Used from the GUI. Updates the parameters of an existing environment with the entered new parameters, which also includes the ID of the environment to be updated. It returns information about the modified environment.

### deleteEnvironment

*input:* **EnvironmentID** environmentID

Used from the GUI. Deletes an environment identified by its ID.

## ➢ ENVIRONMENT INSTANCE




### getEnvironmentAttributes

*output:* **EnvironmentAttributes** environmentAttributes

Used from an Agent. Returns to an agent some useful attributes of its environment, including a list of existing initial states, a list of success final states, and optionally other information (a list of all the existing states and actions, the available actions from each state…).

### setInitialAgentState

*input:* **State** state
*output:* **Actions** actions

Used from an Agent. Sets the entered initial state for an episode and returns the available actions from that initial state.

### performAction

*input:* **Action** action
*output:* **NextStep** nextStep

Used from an Agent. Instructs the environment to perform the entered action. After having taken the action, it returns the next state, if the mission ended or not, the available actions from the new state, and the reward for the action taken.

### getDisplay

*output:* **EnvironmentDisplay** environmentDisplay

Used from the GUI. Returns a current display of the environment.

## A.4. Agent and Environment services


Fig. A.3.: Reinforcement Learning Service in action: Agent and Environment services

Internship report

# Creation of a Reinforcement Learning Service

by Victor Klötzer, 4th year student at the department of Applied Mathematics at INSA Rennes

*October 11, 2021*


## Abstract

As a student in applied mathematics at INSA Rennes, France, I present in this document the work I did during my first engineering internship at the end of the fourth year of my studies. Despite travel and social contact restrictions caused by the coronavirus crisis, I had the opportunity to work on-site for three months in a project group at Industrieanlagen-Betriebsgesellschaft mbH (IABG), Germany. My task was to extend an existing IT solution of IABG with a Reinforcement Learning Service.

My internship covered lots of phases of an engineering project, from documentation, popularization, conceptualization, implementation to the presentation of the work. So this internship allowed me to discover and learn about a theory of machine learning and to gain my first professional experience as a mathematical engineer.

## Résumé

Élève-ingénieur en mathématiques appliquées à l'INSA de Rennes, en France, je présente dans ce document le travail que j'ai réalisé dans mon premier stage ingénieur à la fin de ma quatrième année d'étude. Malgré les restrictions en matière de voyages et de contacts sociaux dues à crise du coronavirus, j'ai eu l'occasion de travailler en présentiel pendant trois mois dans un groupe de projet chez Industrieanlagen-Betriebsgesellschaft mbH (IABG) en Allemagne. Ma tâche consistait à ajouter un Service d'Apprentissage par Renforcement à une solution informatique déjà existante de l'IABG.

Mon stage a couvert de nombreuses phases d'un projet d'ingénieur, de la documentation, à la vulgarisation, la conceptualisation, la mise en œuvre jusqu'à la présentation du travail. Ainsi, ce stage m'a permis de découvrir une théorie d'apprentissage automatique et de réaliser ma première expérience professionnelle en tant qu'ingénieur mathématicien.

## Kurzfassung

Als Student der Angewandten Mathematik an der INSA Rennes, Frankreich, stelle ich in diesem Dokument die Arbeit vor, die ich während meines ersten Ingenieurspraktikums am Ende des vierten Studienjahres durchgeführt habe. Trotz der Reise- und Kontaktbeschränkungen verursacht durch die Coronavirus-Krise hatte ich die Möglichkeit, drei Monate lang vor Ort in einer Projektgruppe der Industrieanlagen-Betriebsgesellschaft mbH (IABG) in Deutschland zu arbeiten. Meine Aufgabe war es, eine bestehende IT-Lösung der IABG um einen Reinforcement Learning Service zu erweitern.

Mein Praktikum umfasste mehrere Phasen eines Ingenieursprojektes, von der Dokumentation, über die Popularisierung, Konzeptionierung, Implementierung bis hin zur Präsentation der Arbeit. Dieses Praktikum ermöglichte es mir also eine für mich neue Theorie des maschinellen Lernens kennen zu lernen und erste berufliche Erfahrungen als mathematischer Ingenieur zu sammeln.


INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839 - 35708 Rennes Cedex 7
Tél. +33 (0) 2 23 23 82 00 - Fax +33 (0) 2 23 23 83 96
www.insa-rennes.fr