ROYAL MILITARY ACADEMY

172nd Promotion POL

Promotion Captain Maxime Séverin

Academic year 2021 – 2022

2nd Master

Generalisation in Multi-Agent Reinforcement Learning

Optional Subtitle

Second Lieutenant Officer Cadet

Hadrien ENGLEBERT

Master Thesis of the department CISS
presented to obtain the academic degree
of Master in Engineering Science
under the supervision of Senior Captain Koen BOECKX, ir
Brussels, 2022

ROYAL MILITARY ACADEMY

Department CISS

Version 1.0

4/4/2022

Generalisation in Multi-Agent Reinforcement Learning

Optional Subtitle

Hadrien ENGLEBERT

Preface

Contents

List of Figures

List of Tables

1. Theory

1.1. Introduction

Artificial intelligence is a broad field of study that aims at making machines execute tasks that are normally assigned to human beings. Machine learning is a sub part of it which is oriented towards developing computer programs that will learn to execute tasks optimally without being explicitly programmed to do so, for example with sequences of `if ... else` statements.

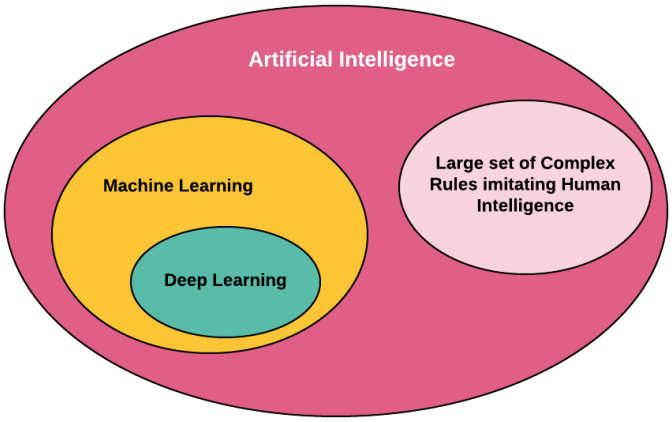

Figure 1.1. Artificial intelligence overview, retrieved from https://vitalflux.com/wp-content/uploads/2020/05/Artificial-Intelligence-vs-Machine-Learning.pngvitalflux.com

It has been a trending topic since the last century but the improvements in computer technologies that the two decades brought allowed to discover and exploit the true potential of various techniques that were already discovered dozens of years ago. Reinforcement learning (RL) and neural networks are perfect examples of this. In the next sections, we will cover the notions of reinforcement learning and more precisely the sub parts of reinforcement learning that we used in this work in order to provide the reader with a basic understanding of the principles that were used in our study. The equations and theoretical notions used in this part are taken from the reference book [5].

1.2. Reinforcement learning

The reinforcement learning framework can be described based on its main elements, namely the **environment**, the **agents** the **state**, the **reward**, the **policy** and the **value**.
The **agent** is the element that will be running a reinforcement learning algorithm in a given medium, often times some sort of game. This medium is what we call the **environment**. The environment determines what the field of action of an agent is, what the interactions between the agents will be like in case there are multiple agents and the most importantly what the **reward** of an agent upon a given action will be. The **reward** will be positive or negative according to the type of behaviour we want to emphasise in the agent. The goal of the agent is only to maximise the **rewards** it obtains by means of the reinforcement learning algorithm that has been implemented for it. In this way, quite similarly to how humans learn in real life, the agent will learn by trial and error to adopt a certain behaviour that we

characterised by shaping the rewards judiciously. As opposed to supervised learning, in this case the learning happens online, by letting the agent actually playing the game. To assess the **reward** it will get in a certain situation if they take a particular action from their action space, the agent applies its RL algorithm to map the **state** of the environment to the estimated reward it would get if it took a specific action, which is called the **value** of the *(state, action)* tuple. The **state** of the **environment** contains all the characteristic that describe the current state of the **environment** and the **agent**. Nonetheless, sometimes the agent cannot monitor the whole **state** of the environment but only has access to part of it. This is why we most of the time rather talk about **observations** as input of the mapping function of the agent. This mapping of the **observation** and each action to certain values, is called the **policy**. Learning the optimal policy is ultimately the goal in reinforcement learning. We will also see that different techniques exist to implement this policy, going from simple linear iterative functions to using neural networks in the case of deep reinforcement learning.

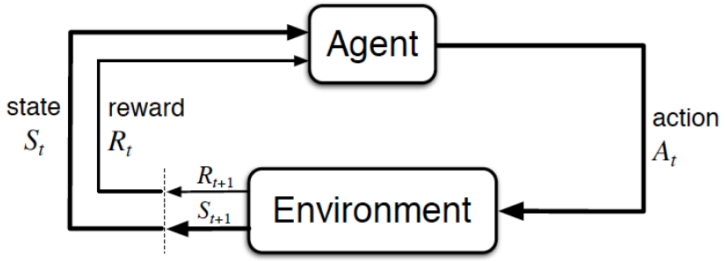

Figure 1.2. Reinforcement learning framework, retrieved from [5]

However, reinforcement learning has a certain amount of pitfalls, well described in [2], that one must take into consideration when working with these techniques. As we explained previously, the agent learns a policy by taking actions and receiving rewards accordingly. But if the agent is stuck in a loop where it only gets negative rewards and keeps taking the same actions it might lead it to believe there is no higher reward available and thus not learn the optimal policy. To avoid this, we need to find a trade of between **exploitation** of the learnt policy and **exploration** of the environment and the possible actions. Indeed, too much exploration, that is taking random actions that are not necessarily recommended by the policy, might lead the agent to forget the learned policy.
Moreover, another difficulty is that the reward an action causes can also be delayed and only be received by the agent several steps later, like when playing a complex strategy in chess.

A common example to illustrate these notions is the game Pac-Man. In this case, the agent would be Pac-Man and it would receive let's say a reward of `+10` when it eats a pellet, `+100` when it eats a ghost and `-500` when it is eaten by a ghost. Its action space would be the movements that are available to it at each moment and the observations could either be the whole **state** of the board with all the pellets' and ghosts' location or simply all that surrounds it until a given distance.

1.3. Markov Decision Process

Markov Decision Processes (MDP) are also an important notion in reinforcement learning as they allow us to mathematically represent ML problems.
An MDP allows to describe a situation in which the results of the actions of an agent in a given states depend not only on the action taken by the agent but also are stochastic variables. A Markov Decision Process is characterised by ([1]) :

- The set of states **T** Containing all the possible states in which the environment can be.
- The set of actions **A** It is composed of all the possible actions an agent can take.
- The transition function $\mathbf{T} : \mathbf{S} \times \mathbf{A} \times \mathbf{S}$ represents the probability that the agent taking an action *a* leads to state *s'*.

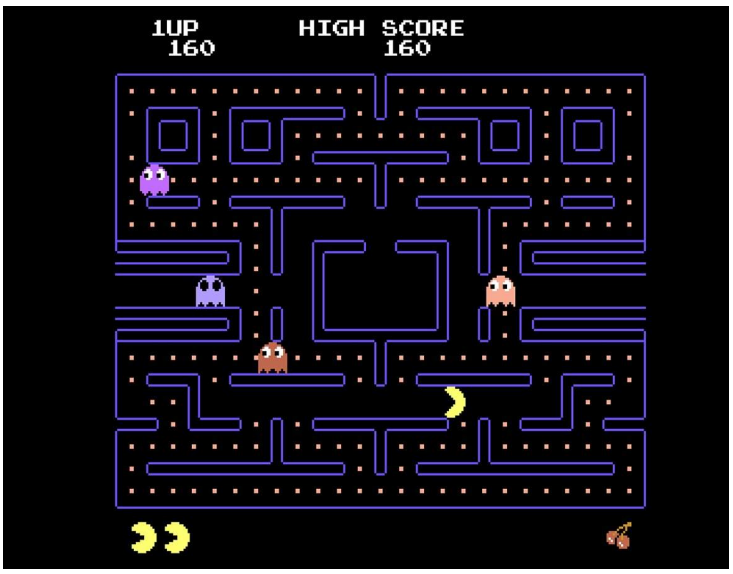

Figure 1.3. Pac-Man game, retrieved from https://www.ns-businesshub.com/wp-content/uploads/sites/11/2018/09/Pac-Man.jpgns-businesshub.com

- The reward **R(s)** which is a sort of utility function, informing the agent over how advantageous the current state is. However this is only true in the short term as we will see with the properties of MDPs.

A Markov Decision Process bears this name because it displays the Markov property. The Markov property assumes that the next state in which the agent is going to find itself does not depend on the history of previous states and actions and only depends on the current state and the action the agent is about to take. In other words, we consider a memoryless stochastic process. To find the optimal policy, the agent will have to maximise its cumulative reward. That is, its reward over the history of the episode and not the instantaneous one. Furthermore, we do not simply consider the cumulative reward as the sum of all the transition rewards at each step which will result in equation 1.1 but add a discount factor γ, which sets the weight of future rewards on the actual decision, resulting in equation 1.2. γ has of course to be between 0 and 1. The smaller this discount factor is, the more the agent will tend to maximise the immediate reward at $t+1$ over the long term cumulative reward.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{1.1}$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.2}$$

In figure 1.4, an example of a simple MDP is displayed where the goal is for the agent to let the car travel quickly without overheating the engine, which can lead to very negative rewards.

To solve an MDP, most RL algorithms will compute what we call **value functions** that are functions of the state and that give the agent an estimation of how advantageous the state in which it fares is or of the expected reward taking a certain action might bring. When following a π policy (as we defined them earlier), [5] define the value function $v_\pi(s)$ as following:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \tag{1.3}$$

Where E_π is the mathematical expectation of the stochastic variable that is the reward, under the assumption that the agent follows the π policy and given that the agent is in state $S_t = s$. This thus reduces to the expectation of the cumulative reward as defined earlier in case the agent is in a particular state.

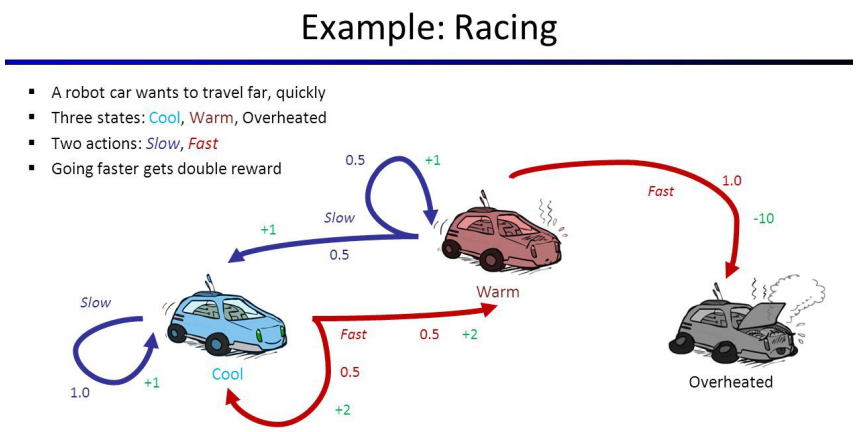

Figure 1.4. MDP example, retrieve from Berkeley's C188 AI Course, Spring 2014

In the same fashion, [5] defines the action value $q_\pi(s,a)$ for an agent following policy π and being in state $S_t = s$ as the expected value of the future return in case the agent takes action $A_t = a$:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{1.4}$$

Those value and action functions can be learned in different ways and this is the core of reinforcement learning algorithms. A simple way to do this by experience would be to simply let the agent play over many iterations and average the value of the functions for all state over all the cycles. This relies on Monte Carlo methods and can become complex because it requires to store lots of data for each combination of state and action which can amount to great numbers depending on the state and action spaces. Therefore many different methods exist to obtain estimates of these functions.

Finally, the last step is to derive the optimal policy from these functions, the policy that will lead the agent to take only actions that maximise the value function. The optimal policies are noted as π_* and all share the same optimal state-value and action-value functions :

$$v_*(s) = \max_\pi v_\pi(s) \tag{1.5}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{1.6}$$

As $q_*(s,a)$ is the expected return if the agent takes action a in state s while following the optimal policy π, we can write the optimal action-value function $q_*(s,a)$ in function of v_*:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*\left(S_{t+1}\right) \mid S_t = s, A_t = a \right] \tag{1.7}$$

Next we will go over different methods to solve Markov Decision Processes and find the optimal policy in different use cases, using other machine learning techniques, with a single agent or several cooperating agents.

1.4. Q-learning

The first method we will see is Q-learning. Q-learning was first proposed by C. Watkins in 1989 and is an off-policy, model free RL algorithm. To explain what these characteristic imply, let's take a look at its working. Q-learning directly approximates the optimal $q*$ action-value function by trial and error with the following equation :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{1.8}$$

This equation will thus approximate the optimal action-value function whatever the policy that the agent follows during the learning, which is why we call such an algorithm off-policy. Besides, Q-learning is also a model-free algorithm, this means that is does not use the transition probability function or the reward function which form the model of the MDP environment. These properties make Q-learning very simple and flexible and explains why it is so much used in reinforcement learning.

However, the policy still somehow influences the learning of the algorithm as it will determine the state-action pairs (s, a) that will be updated. Q-learning has been proven to converge as long as all pairs of the space are updated ([5]).

In its basic form, Q-learning will store the value of all the state-action pairs in what is called a q-table and at each step update the value corresponding to the transition until a terminal state is reached and the episode is finished. This process can then be repeated until the algorithm converges. Here under is an example of a Q-table.

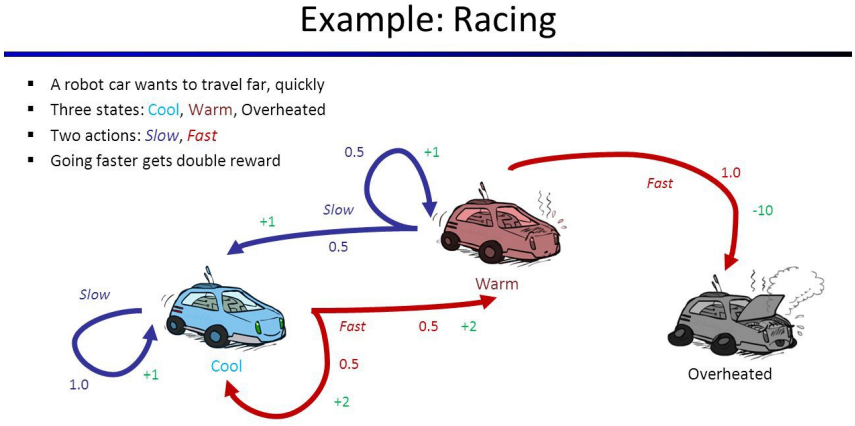

Figure 1.5. Q-table example, retrieved from https://blog.j-labs.pl/2019/09/Reinforcement-Learning-with-Q-Learningblog.j-labs.pl

As previously mentioned, the followed policy will still influence this off-policy algorithm as it will determine the updated pairs. This leads us to two important notion in reinforcement learning, namely **exploration** and **exploitation**. Exploration stands for the testing of all the possible state-action combinations by the agent in order to find the optimal policy and not converge on the first possibility that the agent randomly tests first. But then when the optimal policy is learned, the agent has to apply it and converge towards it to work optimally. In order to find a suitable trade-off between these two principles, a few techniques such as **epsilon greedy** can be of great help. // Epsilon greedy consists in choosing an $0 \geq \epsilon \leq 1$ value and generating at each step a random value between 0 and 1. If this value is greater than ϵ, the agent will take the action with the highest state-action value, otherwise, the agent will take a random action and this way enable exploration of the state-action space of the environment.

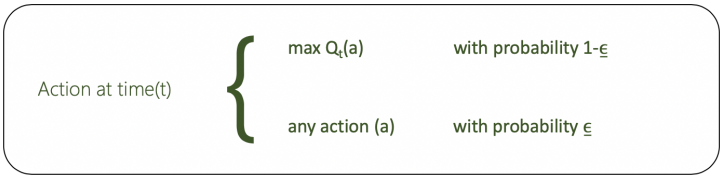

Figure 1.6. Epsilon-greedy, retrieved from https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/geeksforgeeks.org

One can further fine-tune this technique by letting the value of epsilon decay linearly or exponentially over the iterations. //

Now we covered all the aspects of a basic RL problem and of an elementary algorithm that allows to solve it. However, we have encountered various challenges that can make the implementation of a basic Q-learning algorithm much less efficient. For example, in case the environment has significantly big state and action spaces, the Q-table can become so large that the algorithm becomes inefficient. This problem can be circumvented by modifying the way the state§-action value function is learned as we will see in the next sections.

1.5. Deep learning

Deep learning is another subclass of machine learning that makes use of artificial neural network, or also simply called neural networks. Artificial neural networks can be seen models made out of many layers of neurons that are made to solve AI problems, approximate certain functions or recognise patterns in observation inputs. Such a layered model is very flexible and is supposed to be able to model almost any type of function.

At every layer, each neuron will take as input a linear combination of all the outputs of the neurons of the previous layer and have as output this linear combination added to a certain bias and finally passed to an activation function in order to bound the output of each neuron. The coefficients of the linear combination are called the weights of the neuron. See figure 1.7

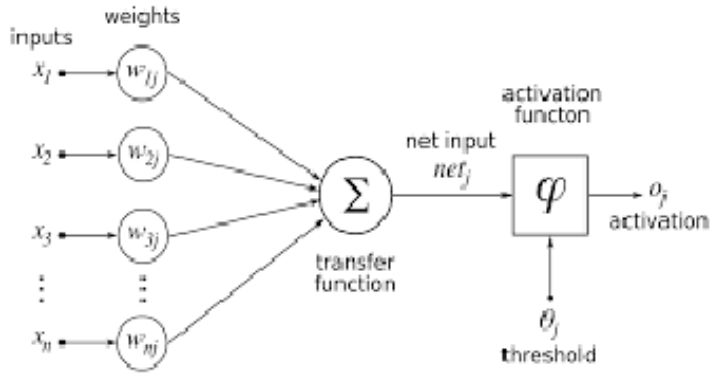

Figure 1.7. Neuron diagram, retrieved from https://cdn.educba.com/academy/wp-content/uploads/2019/03/Working-with-Neural-Network-1.pngcdn.educba.com

The output will represent the possible solution values of the problem the neural network is approximating. For example in an image recognition example where our network would try to distinguish dogs from cats, the input would be the pixels of the image in RGB values and the output would be a value for each of the last two neurons. If the activation threshold is reached for one of the two options, the network has recognised one of the two patterns. Each layer of the neural network will process a certain aspect of the function they are to approximate.

To compute the weights and the biases of the network, the most common method is to use **backpropagation**. We will not explain this method in details here as it is not the focus of our work. To evaluate the accuracy of the model of an artificial network estimating a function, we determine a loss function and the goal of the network is of course to minimize this loss function. The main idea behind backpropagation is to calculate the gradient of the loss function in function of each one of the weights, iteratively layer by layer starting from the last layer and using the chain rule. We can then use this gradient in various ways to minimize the loss function, one of the most used ones is **stochastic gradient descent**. Basic gradient descent will iteratively use the gradient to optimize the parameters and minimise the loss function. On the other hand, stochastic gradient descent uses random samples of the dataset to calculate the gradient in order to reduce greatly the mathematical complexity of the problem and thus improve the computational speed of the algorithm.

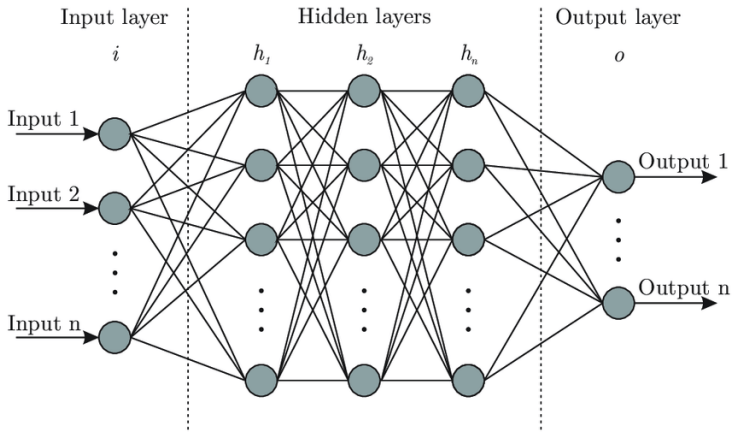

Figure 1.8. Neural network diagram, retrieved from $https://miro.medium.com/max/1000/1 * ub - ifcgdi9xgryqvo0_GRA.png$miro.medium.com

1.6. Deep Q-learning (DQN)

1.7. Q-Mix

1.8. Literature Review

2. Conclusion

A. Supplementary Information

Bibliography

[1] Contributeurs aux projets Wikimedia. modèle mathématique stochastique, Jul 2006.

[2] Maxim Lapan. *Deep reinforcement learning hands-on : apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more.* Packt Publishing, 2018.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[4] Tabish Rashid, Mikayel Samvelyan, Christian Schröder de Witt, Gregory Farquhar, Jakob N. Foerster, and Shimon Whiteson. QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning. *CoRR*, abs/1803.11485, 2018.

[5] Richard S Sutton and Andrew Barto. *Reinforcement learning : an introduction.* The Mit Press, 2018.

[6] Srimanth Tenneti. Reinforcement learning 101 - analytics vidhya - medium, May 2020.

[7] Justin K. Terry, Benjamin Black, Ananth Hari, Luis Santos, Clemens Dieffendahl, Niall L. Williams, Yashas Lokesh, Caroline Horsch, and Praveen Ravi. Pettingzoo: Gym for multi-agent reinforcement learning. *CoRR*, abs/2009.14471, 2020.