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
20/4/2022

Generalisation in Multi-Agent Reinforcement Learning

Optional Subtitle

Hadrien ENGLEBERT

RMA Brussels
Royal Military Academy

Acknowledgements

Preface

1 Motivation and use for Defence

Machine learning is a very trending topic that is currently contributing greatly to the evolution of the way our societies function. This subject is thrilling because the programming challenges it offers and the advances that are still to make in the field of artificial intelligence. This is furthermore a domain that offers many opportunities to innovate and discover new concepts while being at the centre of attention in many domains. We believe that it still can contribute a great deal for the modernisation of Defence although the evocation of this possibility often encounters scepticism and raises ethical debates.

The subject of this thesis is the generalisation of a multi-agent reinforcement learning algorithm that was developed by Cdt Koen Boeckx, ir in the framework of another thesis linked to the (Intelligent Recognition Information System) project. The IRIS project's goal is to develop a software to help armoured vehicle crews to execute their tasks by providing them an intelligent aid to decision making. The goal of this research is to evaluate the operationality of such a reinforcement learning algorithm at its actual stage of development and to find a solution to enhance its usability and thus improve it mainly regarding its adaptability.

2 Description of the research question

The research that will now be lead is aimed at testing this algorithm in various situations that differ regarding the environment in which the agents operate and the tactics implemented by the adversaries. The adversaries modelling actual combat tactics and the different types of environments will test the generalisation ability of this algorithm. All in all, we will try to answer the following question: "How does our algorithm fare in different environments, against various types of adversaries and how can we improve its performance ?".

Contents

List of Figures

List of Tables

List of Abbreviations

Adam	Adaptive Moment Estimation
AEC	Agent Environment Cycle
ANN	Artificial Neural Network
API	Application Programming Interface
DQN	Deep Q-learning
GRU	Gated Recurrent Unit
IQL	Independent Q-learning
IRIS	Intelligent Recognition Information System
MDP	Markov Decision Processes
MLP	Multilayer Perceptron
ReLU	Rectified Linear Unit
RL	Reinforcement learning
RNN	Recurrent Neural Network

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, using different mathematical and statistical tools, to make machines able to learn to perform tasks or optimise their working without being explicitly programmed to do so. In this sense, it looks very much like we enable machines to learn, like humans do, through experience and a lot of data. For example, one might use sequences of `if ... else` statements to make a program follow an arbitrarily determined behaviour, while in machine learning we would rather program an algorithm that will allow our agent to learn a behaviour that enables it to accomplish the given task.

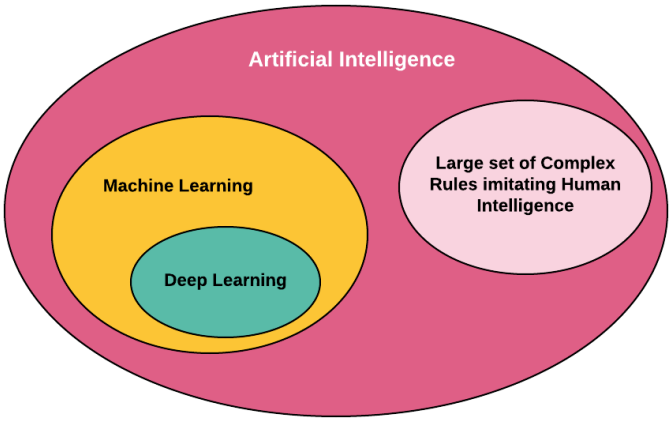


Figure 1.1 Artificial intelligence overview, retrieved from vitalflux.com

Reinforcement learning has been a trending topic since the last century but the improvements in computer technologies that the last two decades brought along allowed to discover and exploit the true potential of various techniques that were already discovered dozens of years ago. and neural networks are perfect examples of this. Reinforcement learning teaches agents to perform tasks by letting them try solutions and rewarding them positively or negatively in real time, while its counterpart, supervised learning, makes use of labelled datasets, that is in other words, tasks and a type solution to teach the agent to approximate the model that maps one onto the other. This last approach is typically used for recognition problems.
In the next sections, we will cover the notions of reinforcement learning and more precisely the sub domains 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 .

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.

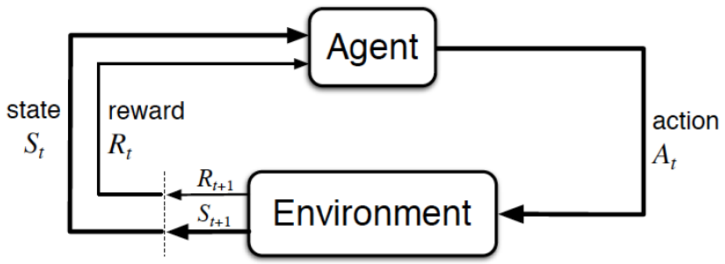


Figure 1.2 Reinforcement learning framework, retrieved from [11]

However, reinforcement learning has a certain amount of pitfalls, well described in [6], 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 state

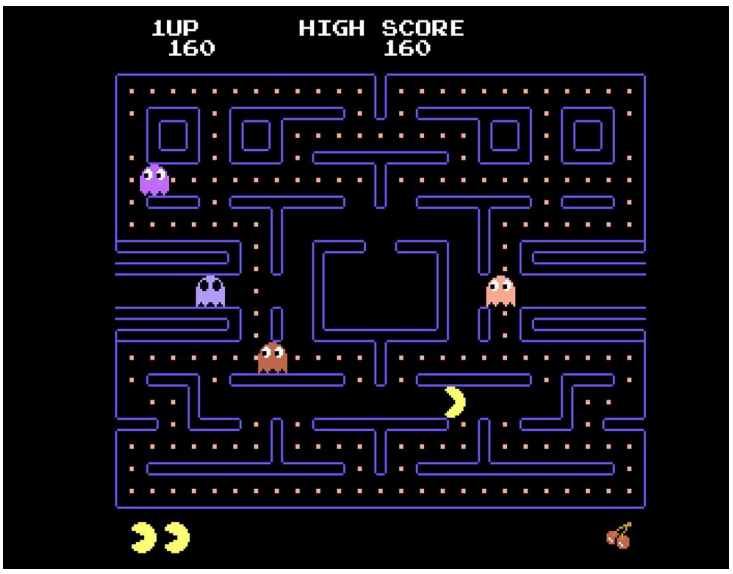


Figure 1.3 Pac-Man game, retrieved from ns-businesshub.com

depend not only on the action taken by the agent but also are stochastic variables. A Markov Decision Process is characterised by ([3]) :

- 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'.
- 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), [11] define the value function $v_\pi(s)$ as following:

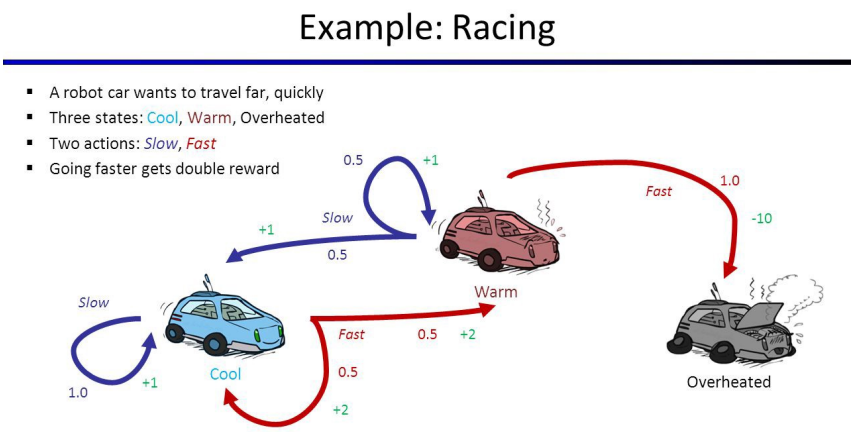


Figure 1.4 MDP example, retrieved from Berkeley's C188 AI Course, Spring 2014

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

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. In the same fashion, [11] 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 ([11]).

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.

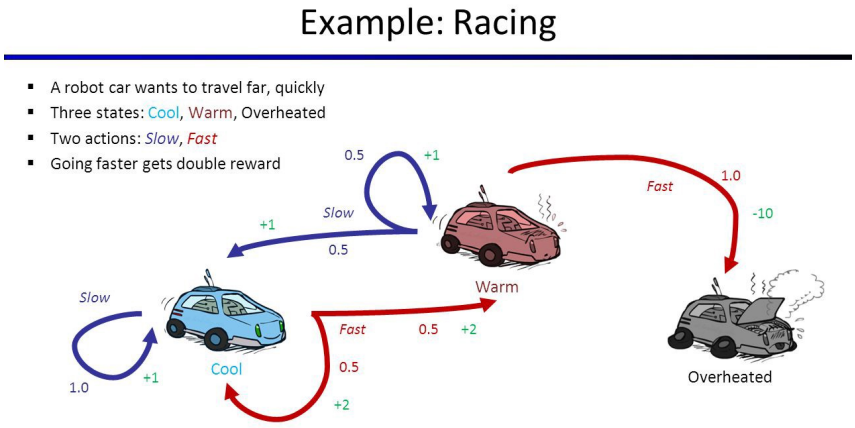


Figure 1.5 Q-table example, retrieved from blog.j-labs.pl

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.

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

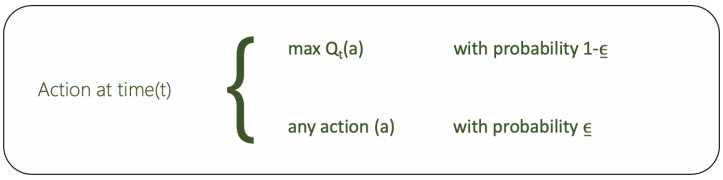


Figure 1.6 Epsilon-greedy, retrieved from geeksforgeeks.org

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. The adjective deep refers to the several layers of the neural networks. The network can be divided in layers disposed parallel to each other and fully connected to the adjacent layers, in other words, each neuron of every layer is connected to every single neuron in the previous layer as well as in the next one. With an exception for the input and output layers, the rest are called hidden layers. 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 1.8

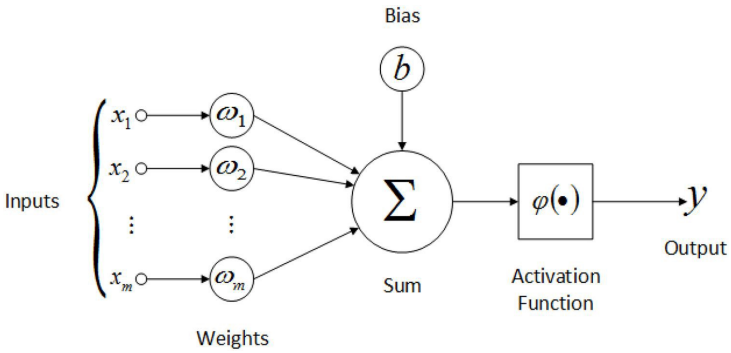


Figure 1.7 Artificial neuron diagram, retrieved from medium.com

Activation functions are essential to the good working of an ANN, they add the non-linearity to the model and allow to normalize the output of each neuron. Without activation function, a neural network would be nothing more than a linear regression model. A bad activation function design can negatively influence the ANN's convergence or even prevent it from converging altogether. The activation functions can be of various nature.

The Sigmoid is one of the most used activation functions because of its range that makes it ideal to approximate probabilities or for binary classification problems and the fact that it is differentiable and has a smooth gradient. However for input absolute values above 3, its gradient tend to be extremely small, leading to the *Vanishing gradient* problem. The tanh function is a shifted version of the Sigmoid and often works better for hidden layers of neural networks, as it centers the output value around 0, between -1 and 1, making the learning easier for other layers. The ReLU (Rectified Linear Unit ,is used in similar cases as the tanh function and is less computationally intensive than the Sigmoid or the

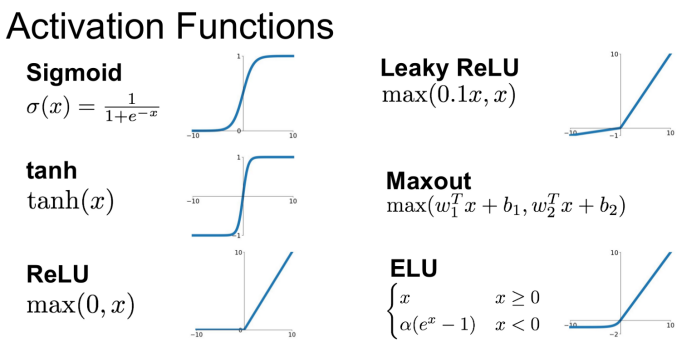


Figure 1.8 Basic activation functions, retrieved from medium.com

tanh but tends to create dead neurons during backpropagation due to the negative side that makes the gradient null. The leaky RELU solves this problem making the negative part non null. ELU is another of ReLU's variants that avoids some of RELU's problems by introducing the log curve in the negative part of its domain but in doing so also adds some computational complexity. Finally the Maxout function is a function that selects the maximum value from a set of linear functions of the input and that will be trained by our model.

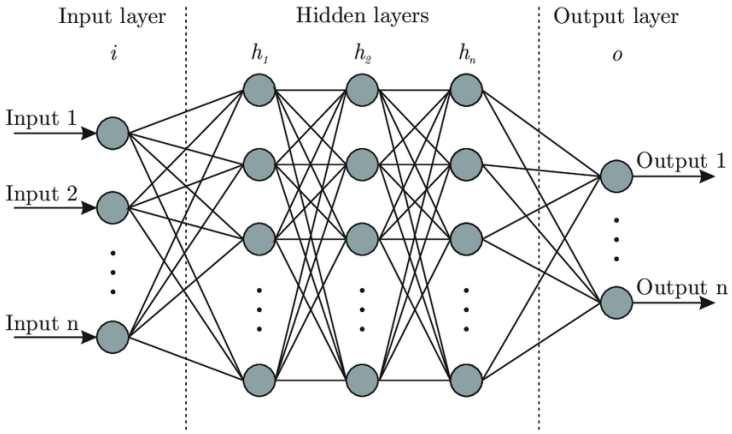


Figure 1.9 Neural network diagram, retrieved from medium.com

The output of the artificial neural network 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**.

Now the main problematic of neural networks is to compute the weights and biases that allow the network to approximate the desired model. To this end, one must define a loss function that will quantify how close the neural network is approximating the model at every step of the learning process and the goal will be to minimise this cost function. In order to optimise the weights and biases of the network to minimise the cost function, we calculate its gradient in function of all the parameters of the network. This is done using **backpropagation**.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. Since the loss function indicates which outputs should be greater or smaller, one can deduce which weights how the weights need to be modified in the last layer to adapt the output. But since the inputs from the last layer are function of the weights of the layer

before it, we can also adapt these to reach the same effect. This reasoning can be applied iteratively over all the layers of the network until we reach the input layer.

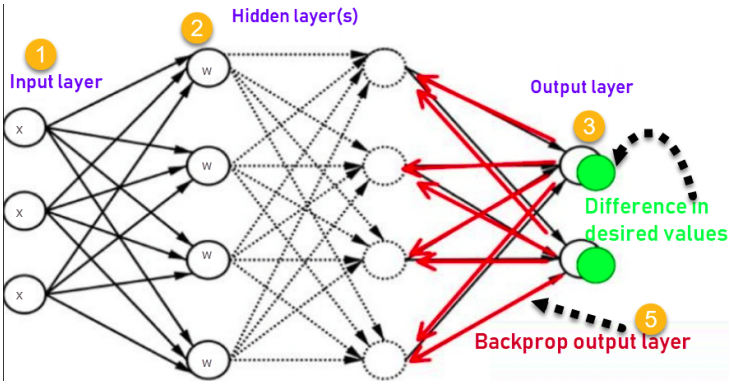


Figure 1.10 Backpropagation, retrieved from mygreatlearning.com

We can then use this gradient in various ways to minimize the loss function, some of the most common ones are **stochastic gradient descent** and the **Adam algorithm**. 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.

However, the optimiser we will be using in this work is the Adam optimiser (Adaptive Moment Estimation), which performs best in general. Adam uses Momentum and Adaptive Learning Rates in order to speed up the convergence but a full explanation of its working is outisde the scope of this thesis. We refer the reader to the original Adam paper [4] for more in-depth explanations.

1.6 Deep Q-networks

We have seen in the previous sections that Q-learning could create a table that allowed the agent to take actions that would maximise its total cumulative reward over the episode. However, what happens when the state-action space becomes extremely large and hence necessitates a huge Q-table ? **Deep Q-learning**, proposed by Mnih et al. in [8] uses deep learning techniques to propose a solution to this problem. Deep Q-learning is based on the same principles as Q-learning but makes use of neural networks to learn the action-value function Q instead of using value iteration over all combinations of state and action. In this case, the inputs of the network are the observations of the agent for the current state and the outputs are the Q-values for each possible action (see figure 2.2).

The learning of the parameters of the network is done once again in a very similar way to what we have already covered with Q-learning. If we look back at equation 1.8, the target value that the current action-value $Q(S_t, A_t)$ was to approximate was the reward received by the agent for the action taken at time t plus the discounted maximal action-value of the next state, reached at time $t+1$ (the part in the red box in the following equation).

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

The Q value will be computed by a target network, a distinct copy of the DQN () of the agent that will not be trained. The DQN of the agent is thus usually called the online network as opposed to the target

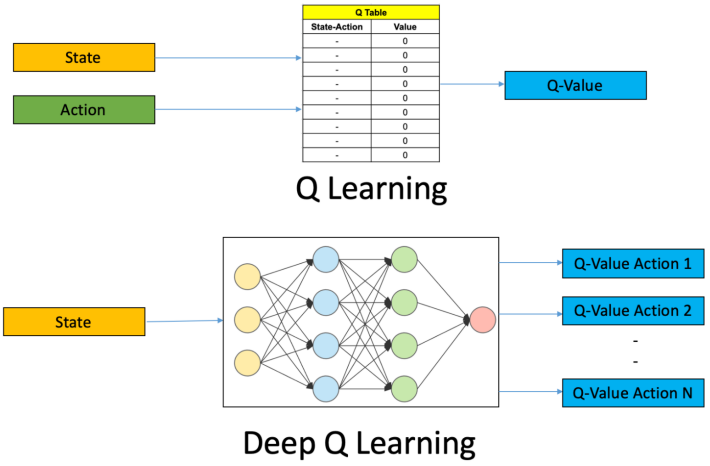


Figure 1.11 Deep Q-network vs Q-table, retrieved from analyticsvidhya.com

network. The parameters of the target network will only be updated periodically to take the value of the weights and biases of the DQN of the agent. From there, we can define the loss as the square root of the target minus the current action-value (cfr [10]).

$$\mathscr{L}(\theta) = \left((R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \theta^{-}\right)) - Q(S_t, A_t; \theta)\right)^2 \tag{1.9}$$

Where θ and θ^{-} represent respectively the parameters of the online network and the target network. This loss can finally be used to train the network of our agent, one can minimise the loss using algorithms such as stochastic gradient descent. In practice, one will store the n past transitions in a replay buffer and will randomly sample b out of these n transitions such that the loss will be the sum of the losses of all the transitions

$$\mathscr{L}(\theta) = \sum_{i=1}^{b} \left[\left(\text{target}_i^{\text{DQN}} - Q(S_t, A_t; \theta)\right)^2\right] \tag{1.10}$$

With $\text{target}_i^{\text{DQN}} = (R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \theta^{-}\right))$

We now have seen how to train an agent to solve a reinforcement learning problem using Q-learning and how to improve this Q-learning by using deep learning and turning it into deep Q-learning to make it more scalable to big environments. But we still have not seen an aspect that is key to the original project this work is contributing to, the IRIS project. Indeed, the goal is to provide an algorithm that would train multiple agents and enable them to cooperate efficiently in order to get a maximum reward in various environments. However, the algorithms we have studied thus far do not offer ways to adapt them to cooperating agents. One could try to use but this approach does not offer very good results as independent agents take longer to learn or might even not converge as their respective explorations interfere with each other. Besides, adding communication between agents greatly augments the complexity of their observation space and this aspect is not the main subject of this work. [12]

Therefore, the approach that we selected here is QMIX.

1.7 QMIX

QMIX is an RL algorithm that can train cooperative agents in a centralised way and enables them to then exploit their training independently. This paradigm is called ”centralised training with decentralised execution” ([10])

The working of a QMIX network can be seen on figure 2.3.

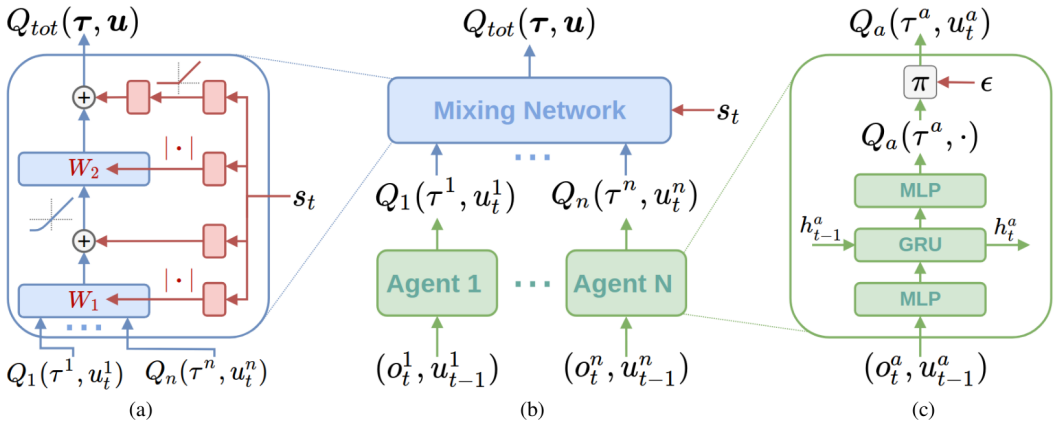


Figure 1.12 QMIX diagram [10]

In QMIX, each agent disposes of their own DQN composed of a , a and another MLP. A MLP designates a basic feed forward artificial neural network, which means that the data goes from the input to the output linearly in one direction. A GRU, on the other hand, takes an input and generates an output but also uses a hidden state as input that it stores in order to use it at the next step. These are not essential to the working principle of QMIX in our application but they allow the algorithm to reuse information from the preceding step to predict the output of the current step. In this manner, the agent can take advantage of any sequential patterns in the opponents' strategy.

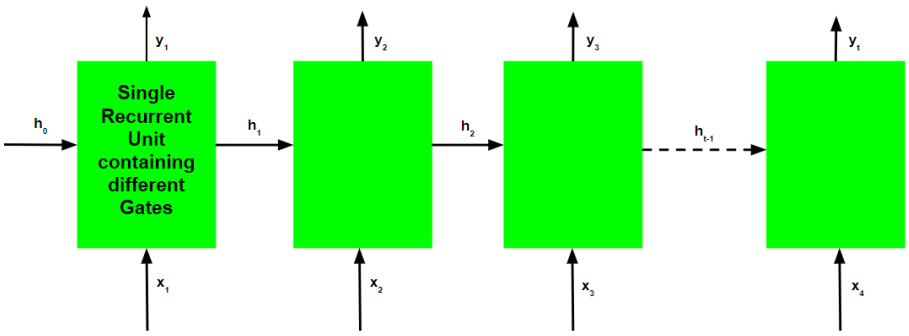


Figure 1.13 Gated Recurrent Unit diagram, retrieved from geeksforgeeks.org

In order to train these DQNs jointly, a mixing network is used that takes as input the action-values of all the agents. And derives a global action value from these inputs that allows the centralised training of the agent networks in the same way as we have seen for deep Q-networks.

$$\mathscr{L}(\theta) = \sum_{i=1}^{b} \left[\left(y_i^{tot} - Q_{tot}(\boldsymbol{\tau}, \mathbf{a}, s; \theta) \right)^2 \right] ?? \tag{1.11}$$

Where $y^{tot} = r + \gamma \max_{\mathbf{u}'} Q_{tot}(\boldsymbol{\tau}', \mathbf{u}', s'; \theta^-)$. The reward in this case is a global reward that is common to all the agents and the target $Q_{tot}(\boldsymbol{\tau}', \mathbf{u}', s'; \theta^-)$ value is the Q_{tot} value obtained when using the maximal Q-values of the agents as input to the mixing network. This means that in case the agents choose an action that does not maximise the Q-value, in case the $\epsilon - greedy$ policy makes it act randomly, the target Q-value will still be computed using the maximal Q-value.

In this equation, $\boldsymbol{\tau}$ represents a *joint action-observation history* for all the agents of the QMIX and a can be seen as the vector of all the actions taken by the agents at a given timestep.
Nonetheless, in this case, the loss function will be calculated using the global action-value Q_{tot}. So we first need to calculate the parameters of the mixer. These parameters will be computed by a hyper-

network that takes as input the global state of the environment and outputs the weights and biases of the mixing network. Then the mixing network takes as input the action-value $-Q_a$ of each agent, every agent sending only the action-value of the action that it took, be it the action that maximises its action-value or a random action due to the implementation of the epsilon-greedy policy. Finally, the parameters of the hypernetwork and the weights and biases of all the agent networks will be optimised in a common gradient descent step to minimise this loss function.
In order for the QMIX method to yield decentralised policies that are in accordance with the centralised ones, [10] states that a sufficient condition is that maximising Q_{tot} globally over the joint action provides the same results as performing the same task over each agent's Qa individually. This way, during execution, agents can participate to a centralised strategy in a decentralised way by taking greedy actions over their own action-value. In the case of QMIX, the authors based their thinking on the observation that they could use this mixing network strategy to approximate any monotonic functions. Henceforward, to satisfy the monotonicity constraint, a relation between Q_{tot} and each Q_a has to be enforced:

$$\frac{\partial Q_{tot}}{\partial Q_a} \geq 0, \forall a \in A \tag{1.12}$$

This condition is satisfied if the weights of the mixing network are bounded to be strictly non-negative. In practice, we will ensure that this condition is respected by performing by passing the concerned outputs of the hypernetwork through an absolute value function before using them in the mixing network.

1.8 Overfitting / Underfitting

[1]

When training an agent to learn a model in machine learning in general, one of the main pitfalls is to find a balance between **overfitting** and **underfitting**.

Overfitting happens when the agent learns the model trying to fit all the data it received during training and ends up memorising it instead of learning the general model. This leads to poor generalisation performance during the testing phase when the model is confronted to environments that present differences with the one the model was trained on. Several situations can lead to overfitting. If the model has a high variance or contains outlier values, we might end up observing the noise and taking it into account in the training. Additionally, the lack of training data, can cause the algorithm to be trained over only part of the model over several iterations. Neural networks, as such, often take much time to be trained and tend to overfit the training data. Finally, the tuning of the hyperparameters of the training algorithm is paramount to obtaining satisfying results in deep learning.

Underfitting is the opposite phenomenon and must also be taken into account when training a RL algorithm. Underfitting means that our algorithm cannot approximate the model we are trying trying to let it learn, it fails for example, in the case of DQN, to map observations to correct $Q(S_t, A_t)$ action-values. It is generally due to the training data containing too much noise or outliers, preventing the model to distinguish the relevant data amongst it, the chosen algorithm being too simple for the model we want to approximate (for example choosing a linear model to approximate a non-linear function), or once again a bad choice of training hyperparameters. Underfitting is more obvious to discover than overfitting as the training error will already be high in case of underfitting.

The goal in machine learning is to find the right balance between overfitting and underfitting.

1.9 Literature Review

In this section we will go over the works that have already been published in relation with the topic of this work, that is the generalisation ability of RL algorithms. A recurrent problem with RL algorithms, as opposed to supervised learning algorithms is that most were designed to perform on popular benchmarks such as the popular Atari games. While the paradigm in supervised learning was from the

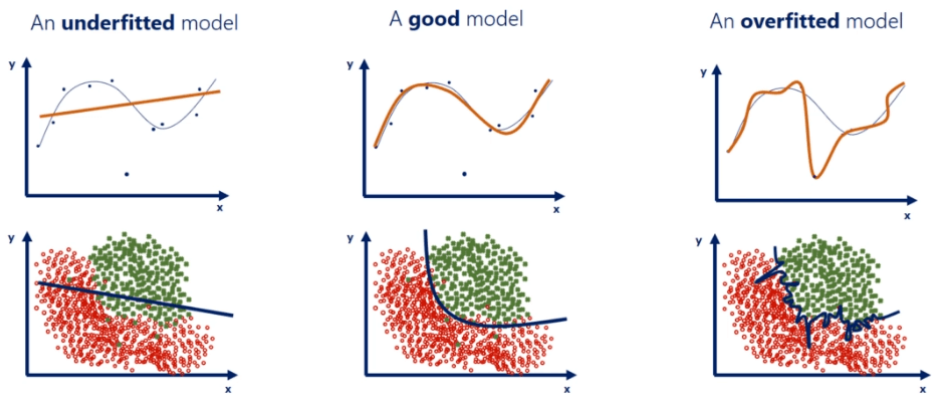


Figure 1.14 Underfitting vs overfitting, retrieved from skillplus.web.id

beginning that the learning set and the test set are to be clearly distinct. Resulting in many researchers developing RL algorithms that tend to work very well on their training environment but perform very poorly on even slightly modified environments.

An interesting contribution has been made with [5], which discusses *zero-shot policy transfer*. This is particularly interesting for us as this means that the paper focuses on evaluation RL algorithms over test environments which are different from the training ones, without allowing them to retrain over a few episodes on the test environment nor to use any additional data over the test environment. It gives an overview of the current benchmarks and protocols for generalisation assessment in reinforcement learning as well as methods for improving generalisation like increasing the similarity between the training and testing by learning efficient environment generation policies (to avoid training our agent on environments that are unsolvable or lead to trivial solutions), or handling the difference in environments by adapting online. This last way consists in adapting the agent's policy during testing within a single episode, since we test it in the framework of zero-shot policy transfer. The adaptations thus needs to happen very fast. At last, generalisation can obviously be improved by avoiding overfitting as much as possible during training. This paper is an interesting starting point to have an overview of the work that has already been done on our subject. However, this paper only discusses single agent reinforcement learning problems while we also train multiple cooperating agents in this work.

Another interesting contribution is [2], which comes up with the notions of *policy similarity metric* and *contrastive metric embeddings*. These notions exploit the fact that in reinforcement learning, contrary to supervised learning, the agent executes its decision-making process in a sequential manner. In this sense it is innovative as most previous approaches were inspired by what has already been done in supervised learning.

Taking advantage of the sequential aspect of reinforcement learning, the authors thought of designing a metric that would allow the agent to assess how similar two situations are based on the action sequence it is going to follow according to the policy. That is, if the agent's solution for a problem is similar in two given situations, then even if the observation of its current state is different, the similarity metric would allow the agent to realise that the two situations are comparable. The authors name this behavioural similarity.

This way, the agent will rather learn a representation that will bring two states close to each other when they are behaviourally similar and push them apart when they are behaviourally dissimilar rather than base itself purely on observations. To do so, they developed contrastive metric embeddings, based on contrastive learning to use their state-similarity metric to train their algorithms. This contribution presents very promising principles that could be of interest for our work.

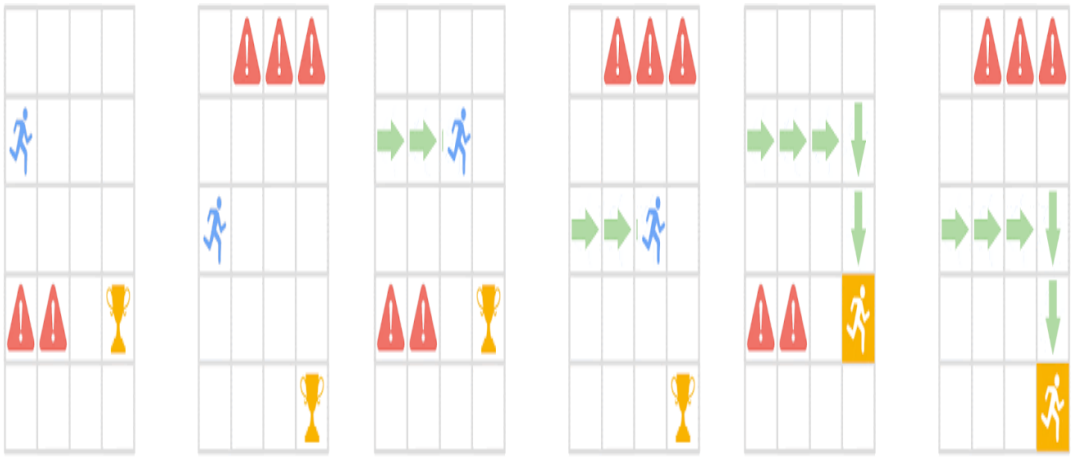


Figure 1.15 Behavioural similarity in observationally dissimilar states, retrieved from ai.googleblog.com

2 Methodology

In this chapter, the methods used to implement the algorithms we have seen in the previous chapter and evaluate our designs will be described. All of our work was programmed using the Python language and can be found on GitHub at the following location GitHub/metaxxxa/tfe. The library we used to work with neural networks elements was the Python library `PyTorch`.

2.1 The Environment

As mentioned earlier, this thesis is based on the work done by Cdt, ir Koen Boeckx for the IRIS project. Hence, our research has been done on the environment used in the said contribution. This environment is the `Defense_v0` environment coded in Python too. This environment is based on the Agent Environment Cycle (AEC) mathematical model for games introduced in [14] with the `PettingZoo` API. PettingZoo is a popular API which contains many environment, single or multi-agents designed for research in reinforcement learning, for multiple agents problems in particular.

The `Defense_v0` environment is a fully observable environment, meaning that agent can observe all the characteristics of the state of the environment at all times. This environment can be represented by a matrix or a grid of coordinates which can either contain an obstacle, an agent representing a tank, or nothing. The environment's input parameters are the number of agents per team, the maximum range and the maximum number of steps. The agents are distributed in two opposite teams of equal numbers. At each step, agents can take an action from 0 to 6 or more depending on the number of agents per team:

- 0: do nothing
- 1: move left
- 2: move right
- 3: move up
- 4: move down
- 5: fire
- 6: aim at agent 1 of opposing team
- (7: aim at agent 2 of opposing team)
- ...

If the opposing agent is within the maximum range parameter upon firing, it is destroyed. At each step all agents receive a reward. This reward equals 1 if the agent destroys an opposing agent by firing on it, it equals -1 if the agent is itself destroyed by an opposing agent or -0.1 in the other cases. This way, agents are individually rewarded for ending an episode quickly and destroying opposing agents.
The agents can observe the current state of the environment at any moment and get an observation that is composed of :

- An observation array containing:
  - The state of the agent: its coordinates, status (alive or not), ammo level and whether it is aiming or not
  - State information of its teammate agents
  - State information of the agents of the opposing team
  - Coordinates of all the obstacles of the environment

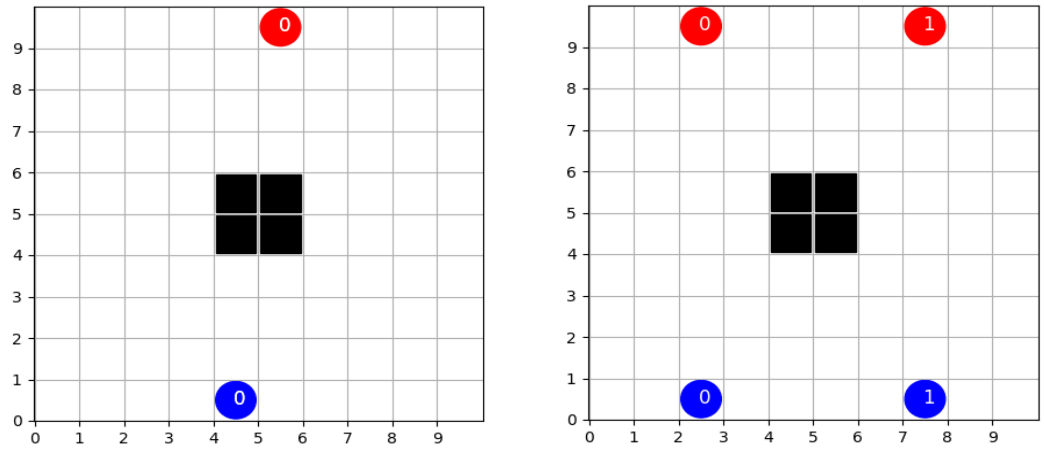
- An action mask: a Boolean array indicating which actions are available to the agent in the current state. Actions can be forbidden in a number of situations:
  - When the agent tries to move outside of the grid or to coordinates already occupied by another agent or an obstacle
  - Firing in case there is no ammunition left, the target is beyond the maximum range or an obstacle or an agent blocks the line of sight
  - All actions when the agent is not alive anymore

When all agents of one team have been destroyed or when the maximum number of steps is reached, an episode is terminated but no extra reward is distributed.

As a consequence of the environment being fully observable and the agent receiving at each time an exhaustive description of the environment, the size of the observations will vary in function of the environment in which the agent is immersed. Hence, we either have to design algorithms that are able to handle variable input size or make the observation size fixed. We will opt for the second option in this work to focus our attention on other aspects of the learning.

2.2 Benchmarking

A benchmark simple environment was chosen to test out all of our models and be able to compare their performance over a number of episodes. This environment is a 10 by 10 grid containing a central 4 by 4 obstacle with 1 or 2 cooperating agents on each extremity for DQN and QMIX respectively. In order to assess the performance of the agents in an unbiased way, they will be evaluated against agents that act totally randomly (within the actions allowed by their action mask at all times).



(a) Benchmark 1 vs 1 environment

(b) Benchmark 2 vs 2 environment

Figure 2.1 Benchmark central obstacle environments

This is the basic environment that will be used first to train the algorithms and assess their performance as such before moving on to the generalisation part.
To evaluate the generalisation performance of our algorithms, the first step we will take is to evaluate their performance on environments that totally differ from the one they were trained in, generating state situations and thus observations the algorithms most likely have not been exposed to during their training. In order to do so, the algorithms that were trained on the benchmark environment will be tested on sets of randomly generated environments. Therefore, we wrote functions that generate environments given a ratio of number of obstacles to number of points in the grid. The number of obstacles and agents being always the same, the size of the observations does not change across the environments of the same batch, allowing us to assess the performance of one algorithm across all the

environments generated. Then we can either place the agents randomly or onto strategic points in order to test certain characteristics of the algorithm.

The first RL algorithm we implemented was deep Q-learning.

2.3 Deep Q-learning

To implement a DQN, a buffer has to be initiated to store the replay memory, which we call a replay buffer. To this aim, we first run the environment while letting our agent act randomly until we stored enough transitions in the replay memory to reach its minimal size. Then, we can start the actual learning. At every step, with probability ϵ, the agent will act randomly, with ϵ decreasing linearly or exponentially trough the training (ϵ-greedy). In the other case, the agent will input its last observation into the neural network that will compute the Q-values for each action. The action that maximises the Q-value while being legal according to the action mask will be selected and executed by the agent that will receive in exchange a new observation from the environment.

A new transition tuple (s_t, a_t, r_t, s_{t+1}) is then stored in the replay memory. After this, we randomly sample the replay memory to get a batch of N transitions that we will use to perform another optimiser step. The loss is computed using equation 1.10. The optimisation is in our case carried out using the Adam algorithm with the `Pytorch` function Adam optimiser. This whole process is repeated until the episode terminates, when this happens, the environment is simply reset and the training resumes until we reach the desired number of training transitions.

2.3.1 Parameters choice

In machine learning, fine-tuning the hyperparameters of the algorithm is essential to the proper functioning of the algorithm and its convergence. By default, we used a classical sequential network structure of the input layer, counting 10 observation inputs, followed two linear layers of 32 and 64 neurons respectively and lastly the output layer which counts 7 Q-values for the agent can only take 7 actions as there is only one agent on the opposing team.

The replay memory contains 2000 transitions, the batch size N is set to 64 and the learning rate equals $\alpha = 0.5e - 4$. The choice of the learning rate is important as a too big learning rate could prevent the algorithm from converging correctly while on the other hand a too small learning rate could make the algorithm too slow to converge in the allotted number of steps. The learning rate recommended in the Adam paper is $1e - 3$, after some trials we decided on $1e - 4$ for our application. The discount rate γ is set to 0.95.

For the $\epsilon - greedy$, we worked with a linear ϵ decay starting from 1 and decreasing until 0.1 over 400 000 steps for a training that lasts 800 00 steps. This is once again a value that cannot be mathematically estimated in advance with precision and requires trial and error to find the right compromise between exploration and exploitation. Finally, the target network is synchronised with the online network every 200 steps.

2.4 Q-Mix

We implemented the DQN algorithm to model strategies for 1 on 1 scenarios. In order to study scenarios that confront several agents against each other, we implemented QMIX. Which, we recall, is founded on a centralised training and decentralised execution approach based on DQNs.

First, we implemented the agents' DQNs which actually are Recurent Neural Networks (RNN) due to the fact that they contain a Gated Recurrent Unit cell. In practice, following the original paper's

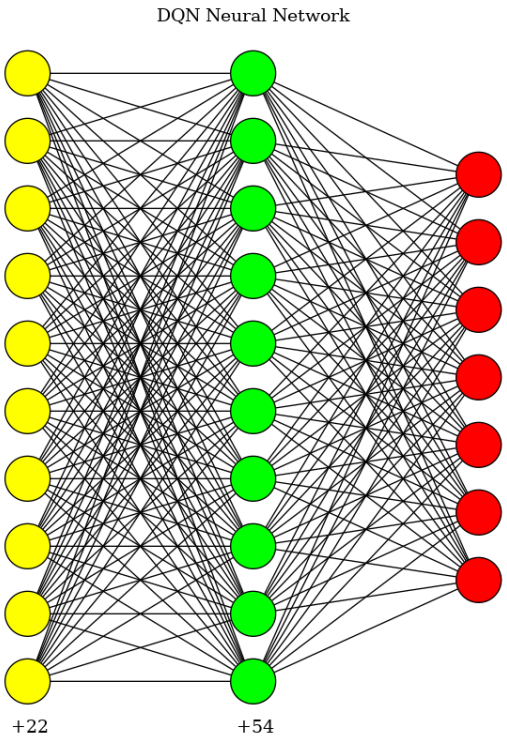


Figure 2.2 DQN neural network representation

recommendations, we programmed them with a linear ANN layer, followed by a ReLU function after which we directly placed the GRU cell as hidden layer and then a final linear ANN output layer.
The mixing network on the other hand is implemented in a different way. The mixing network is composed of two linear layers separated by an ELU function. However, the weights and the biases of these layers are calculated at each step and depend on the total state of the environment, which takes in all the observations of the agents from the team. The parameters are each estimated by an independent ANN that takes as input the entire total state. The weights of the two layers are computed by a single layer ANN followed by an absolute value function. On the other hand, the biases of the first layer are computed by a single layer ANN while the biases of the second layer are calculated by a two layer ANN with a ReLU function in-between the two layers. This way the outputs of the weights network simply have to be reshaped in a matrix of *nxm* with n being the number of inputs of the layer and m being the number of inputs of the following layer, to then perform a matrix multiplication in order to pass the information through the mixer layer.

At every step, each agent computes their Q-values based on their observations and current hidden state and takes an action following the $\epsilon - greedy$ policy while the transition for each agent is stored in the replay memory in the same way as for the DQN. The agents also transfer the Q-value of the action they took, in other words either the maximum Q-value for this state or the $\epsilon - greedy$ random action. The mixing network the takes as input the Q-value of each agent through its network as well as the total state, which is nothing more than all the observations of the agents concatenated into one large fixed size vector. Using the total state, the mixing network calculates its parameters and computes the $Q_{tot}(\boldsymbol{\tau}, \mathbf{a})$.

We then can compute the target value and the error using equation **??**. The optimisation is performed at each step and on all the networks that calculate the parameters of the mixing network as well as on the agents' network. To speed up the learning and following the methods applied in the paper, we let all the agents of the team share the same network. This should not cause any problems as all the agents have the same goal and range of actions and thus should apply the same strategies to cooperate.

The optimisation is executed in the same way as explained in the previous section, using `Pytorch`'s

Adam optimiser over the desired parameters. These steps are repeated at each timestep until all the agents of a team are eliminated or until the maximum number of timesteps is reached.

Nonetheless, other subtleties needs to be taken into account here as we work with teams of agents and not single agents anymore. Firstly, each time an agent dies, the total observed state changes in size somehow as the observations of this agents are not available anymore. Except if any transition where an agent is done is to be used, we need to fill observations for this agent anyway since the mixing network takes as input all the observations and the Q-values of the agents.

To compensate for this we take the last observations of the dead agents for the rest of the episode and set their Q-values to 0 for all the transitions during which the agent is done. Notwithstanding, when the game ends and a team wins, the reward is still distributed to all agents, negative or positive, in order to allow agents to learn actions that can lead to the death of one or more agents but still lead to victory. So the main goal in our framework is for the agents to win the game by all means. An encoder would have also been an interesting way to solve this problem. However this represents a significant amount of supplementary coding and requires much more computing time so it is not part of the scope of this work.

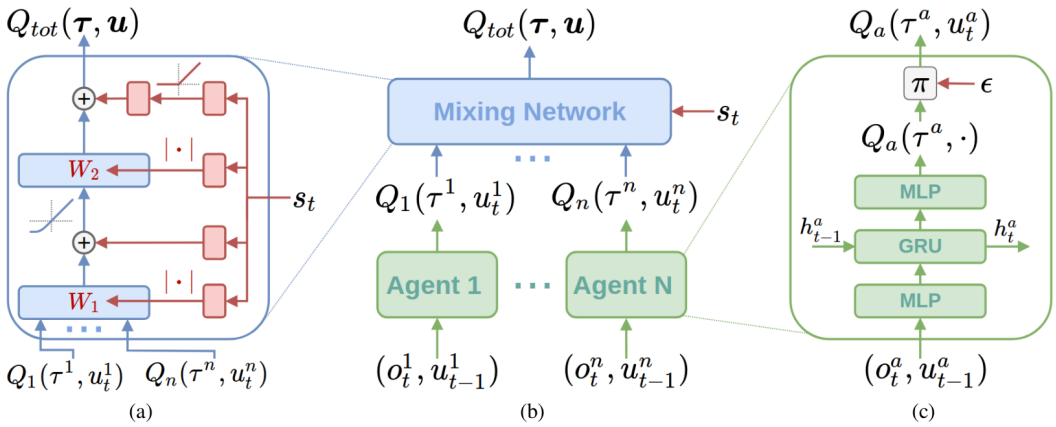


Figure 2.3 QMIX diagram [10]

2.4.1 Parameters choice

Firstly, the agents' RNN is composed of the input layer which takes in the 10 observation inputs, then a linear layer of dimension 32 followed by the ReLU function and the GRU cell of 32 neurons again. The final output layer contains the 8 possible actions.

Moreover, the mixing network contains layers of 32 neurons as well.

The learning rate is set to $0.5e-4$, the replay memory buffer size is 2000 with batches of 64 transitions and a discount rate γ of 0.95. The $\epsilon - greedy$ policy is based on a linear epsilon decay going from 1 to 0.1 over 500k steps when the learning lasts a million steps. Finally the target networks are updated every 200 steps with the values of the online network.

2.5 Training on randomly generated environments

The first technique we tried to improve the adaptation capacity of out algorithms rests on the principle proposed in [5], which proposes to make the training more similar to what the algorithm will face during testing. Therefore, our solution is to train the algorithms over the same number of steps as during benchmark and using the same parameters but changing the environment in which the agents operates every N episodes.

This solutions however already presents limitations due to the way we implemented our DQN and QMIX, namely without encoder and only accepting fixed size observations. Hence, we only can work with environments that contain the same number of obstacles as every observation contains each obstacle's coordinates. Changing the environment periodically while keeping the total number of training steps will also inevitably lead to less training over the same observations and hence most likely worse performance on the benchmark environments than algorithms that were exclusively trained on those.

3 Results

3.1 Benchmarking

3.1.1 DQN

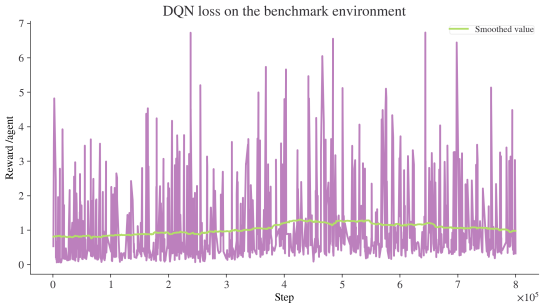


Figure 3.1 DQN loss during benchmark training

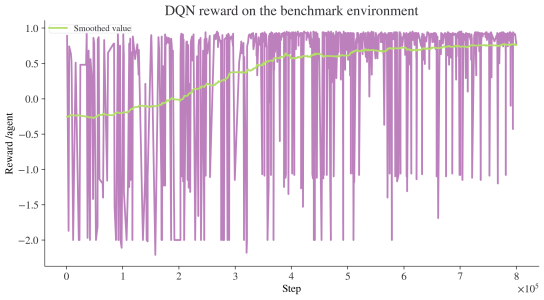


Figure 3.2 DQN reward during benchmark training

3.1.2 QMIX

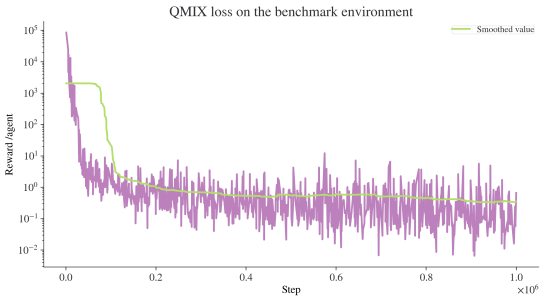


Figure 3.3 QMIX loss during benchmark training

3.2 Evaluation over randomly generated environments

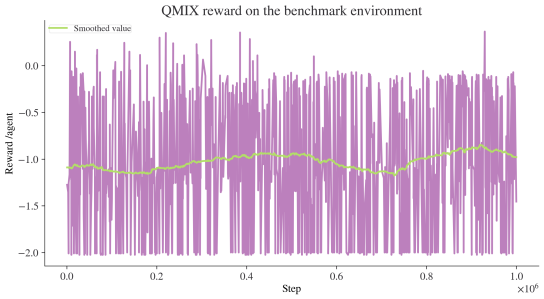


Figure 3.4 QMIX reward during benchmark training

4 Conclusion

Bibliography

[1] Overfitting vs. underfitting: What's the difference?, 2022.

[2] Rishabh Agarwal, Marlos C. Machado, Pablo Samuel Castro, and Marc G. Bellemare. Contrastive behavioral similarity embeddings for generalization in reinforcement learning. *CoRR*, abs/2101.05265, 2021.

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

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[5] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *CoRR*, abs/2111.09794, 2021.

[6] 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.

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

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. , 518(7540):529–533, February 2015.

[9] Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Xiaodong Song. Assessing generalization in deep reinforcement learning. *ArXiv*, abs/1810.12282, 2018.

[10] 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.

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

[12] Ming Tan. Multi-agent reinforcement learning: Independent versus cooperative agents. In *ICML*, 1993.

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

[14] 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.