ROYAL MILITARY ACADEMY

172nd Promotion POL

Promotion Captain Maxime Séverin

Academic year 2021 – 2022

2nd Master

Generalisation in Multi-Agent Reinforcement Learning

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

19/5/2022

Generalisation in Multi-Agent Reinforcement Learning

Hadrien ENGLEBERT

RMA Brussels
Royal Military Academy

Acknowledgements

First and foremost I would like to thank my supervisor Cdt Koen Boeckx, ir for the help, advice and support. Thanks to my reader Dr Nita Cornelia for the follow-up. The 172^{nd} class of Polytechnics for these 5 rich in experiences years and the whole staff of the Royal Military Academy for their commitment. Finally I thank my family and friends for always being supportive.

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 to 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 multi-agent reinforcement learning algorithms on an environment 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 tool 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 performed is aimed at testing this algorithm in various situations that differ regarding the environment in which the agents operate and the information that is available. The various 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
CNN	Convolutional Neural Network
DDQN	Double Deep Q-Network
DQN	Deep Q-learning
GRU	Gated Recurrent Unit
IQL	Independent Q-learning
IRIS	Intelligent Recognition Information System
MDP	Markov Decision Processes
MLP	Multilayer Perceptron
PER	Prioritized Experience Replay
ReLU	Rectified Linear Unit
RL	Reinforcement learning
RNN	Recurrent Neural Network
SSIM	Structural Similarity Index
VDN	Value Decomposition 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 functioning 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.

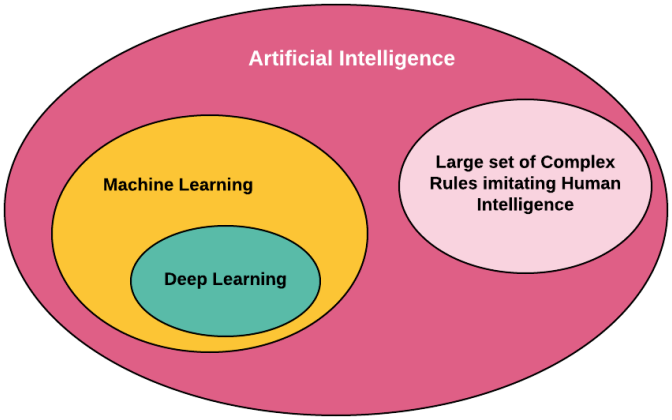


Figure 1.1. Artificial intelligence overview, retrieved from vitalflux.com

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. Reinforcement learning (RL) and neural networks are perfect examples of this. Reinforcement learning teaches agents to perform tasks by letting them try out 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. Finally, unsupervised learning is at the other extreme end of the supervision spectrum. In the case of unsupervised learning, the algorithm is fed unlabelled data and must find patterns in it to classify it. One of the most occurring examples of this is clustering.
In the next sections, we will cover the notions of reinforcement learning and more precisely the subdomains of reinforcement learning that we will use 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 [16].

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 most importantly what the **reward** of an agent when executing 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 to maximise the total **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 encourage 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 expected accumulated reward it would get as from the next state until the end of the episode, 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.

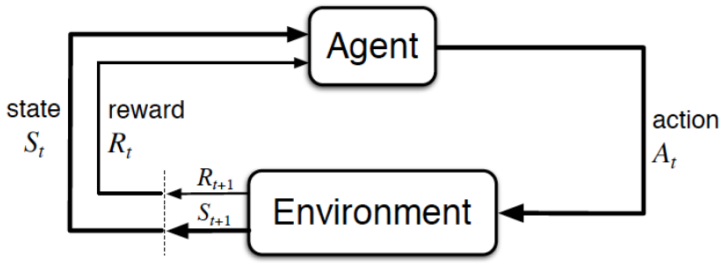


Figure 1.2. Reinforcement learning framework, retrieved from [16]

However, reinforcement learning has a certain amount of pitfalls, well described in [9], that one must take into consideration when working with these techniques. As we explained previously, the agent learns a policy by taking actions and receives 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-off 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 [9].
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.

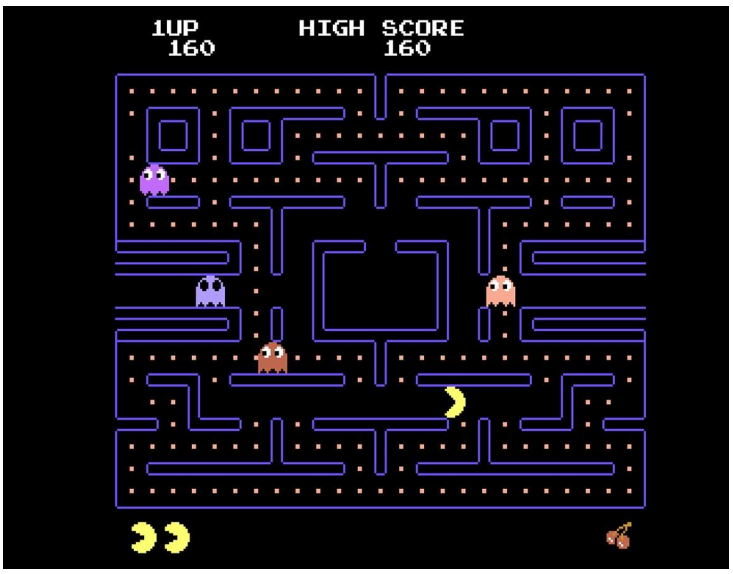


Figure 1.3. Pac-Man game, retrieved from ns-businesshub.com

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 depend not only on the action taken by the agent but also are stochastic variables conditioned on the action taken. A Markov Decision Process is characterised by ([4]) :

- 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 $\mathbf{R(s)} : \mathbf{S} \Rightarrow \mathbf{R}$ 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.

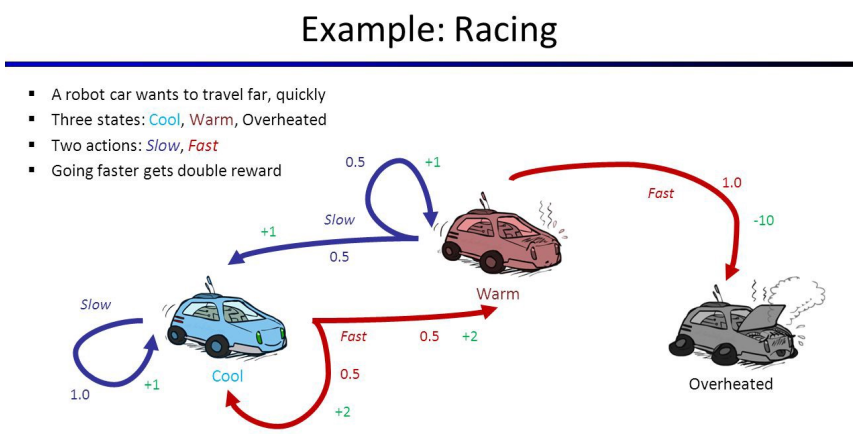


Figure 1.4. MDP example, retrieved from Berkeley's C188 AI Course, Spring 2014

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 policy π (as we defined them earlier), [16] 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 policy π and given that the agent is in state s at time t. This thus reduces to the expectation of the cumulative reward as defined earlier. In the same fashion, [16] 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 [22] and is an off-policy, model free RL algorithm. To explain what these characteristic imply, let's take a look at how it works. 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 ([16]).

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. Below is an example of a Q-table.

Q-Table

	ACTION 1	ACTION 2	...	ACTION N
STATE 1	2.7	3.4	...	-1.3
STATE 2	6.7	-0.5	...	0.0
...	...	...	...	...
STATE N	-1.1	2.4	...	5.9

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 $0 \leq \epsilon \leq 1$ 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.

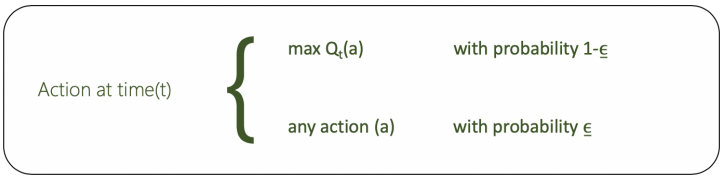


Figure 1.6. Epsilon-greedy, retrieved from geeksforgeeks.org

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 (ANN)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 1.7

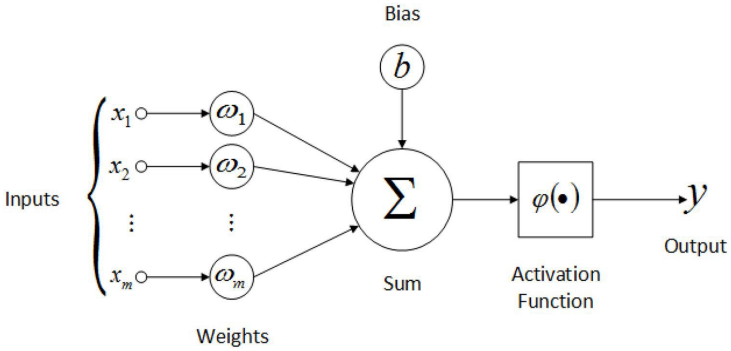


Figure 1.7. Artificial neuron diagram, retrieved from medium.com

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

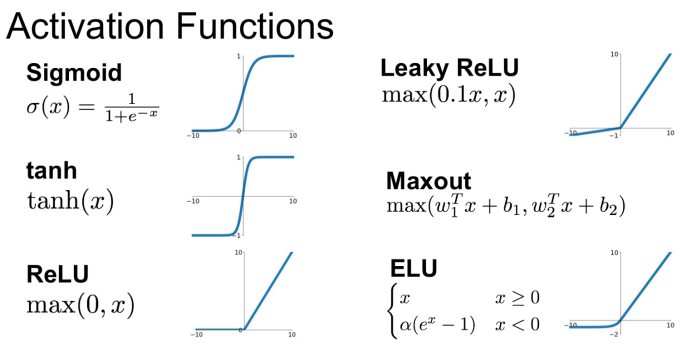


Figure 1.8. Basic activation functions, retrieved from medium.com

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 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. [1] [5]

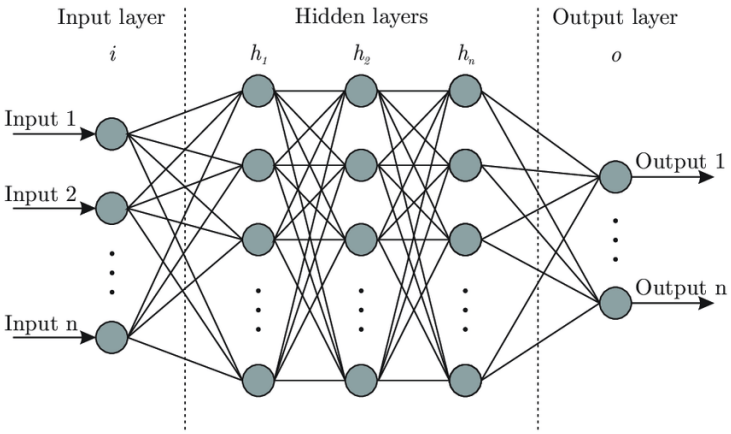


Figure 1.9. Neural network diagram, retrieved from medium.com

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 numerical value for each of the two options. 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 approximating.

Now the main problema 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.

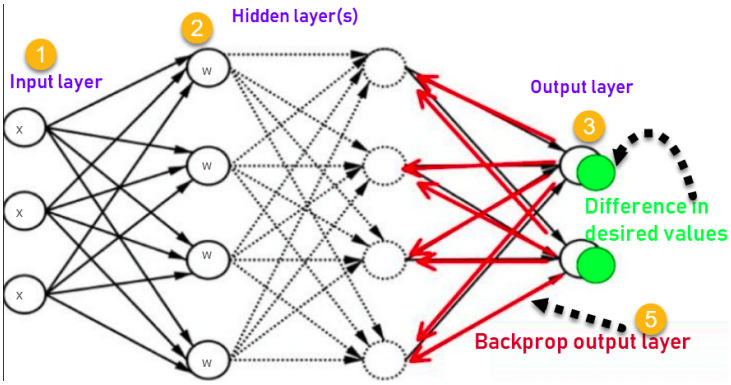


Figure 1.10. Backpropagation, retrieved from mygreatlearning.com

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 [7] 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 [11] 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 2.7).

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).

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

The $\hat{Q}$ value will be computed by a target network, a distinct copy of the Deep Q-Network (DQN) of the agent that will not be trained. The DQN of the agent is thus usually called the online network as

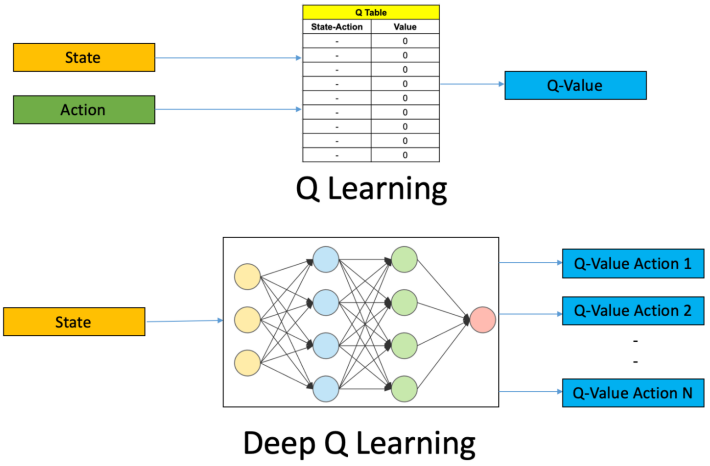


Figure 1.11. Deep Q-network vs Q-table, retrieved from analyticsvidhya.com

opposed to the target 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 of the target minus the current action-value (cfr [13]).

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

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

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. [17]

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” ([13])

The working of a QMIX network can be seen on figure 2.8.

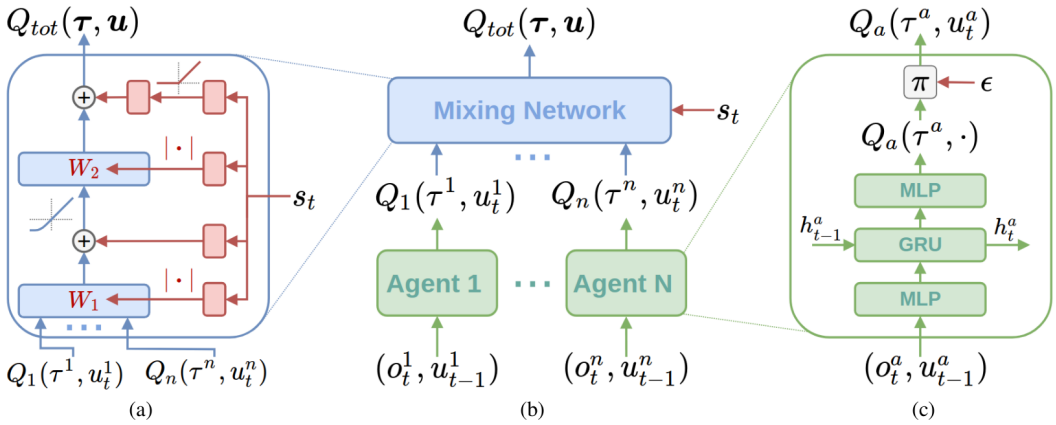


Figure 1.12. QMIX diagram [13]

In QMIX, each agent disposes of their own DQN composed of a Multilayer Perceptron (MLP), a Gated Recurrent Unit (GRU) 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.

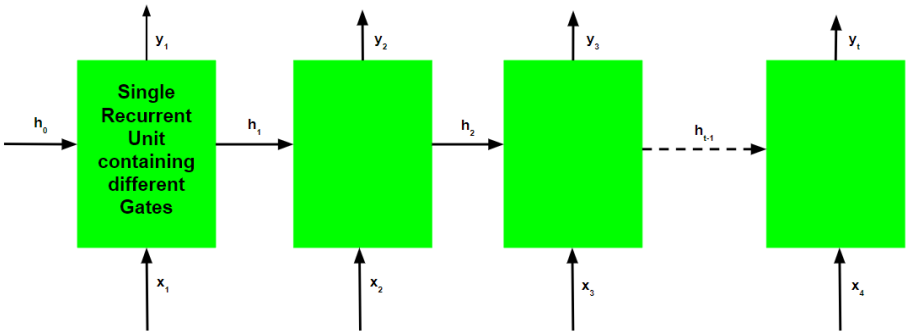


Figure 1.13. Gated Recurrent Unit diagram, retrieved from geeksforgeeks.org

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.

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

(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 observations and 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 hypernetwork 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, [13] 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 Q_a 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:

$$\frac{\partial Q_{tot}}{\partial Q_a} \geq 0, \forall a \in A \leftrightarrow \underset{\mathbf{a}}{\operatorname{argmax}} Q_{\text{tot}}(\boldsymbol{\tau}, \mathbf{a}) = \begin{pmatrix} \operatorname{argmax}_{a^1} Q_1(\tau^1, a^1) \\ \vdots \\ \operatorname{argmax}_{a^n} Q_n(\tau^n, a^n) \end{pmatrix} \tag{1.12}$$

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. Convolutional Neural Networks

Convolutional Neural Networks (CNN), are a specific ANN architecture that is particularly efficient for tasks linked to image or even language processing. Their characteristic convolutional layers allow them to detect patterns in structured data matrices such as images while pooling layers downsample the data and reduce the risk of overfitting. //

1.8.1. Convolutional Layer

The convolutional layer is the base element of CNNs and can be described by the **filters** and their number, the **stride** and the **padding**. The **filter** corresponds to a matrix of a given size (often three by three but it is not necessarily square) that will slide across the input matrices and perform an element-wise multiplication with the elements of the input matrix it is applied on to compute the output matrix. The number of filters will thus determine the number of output channels or feature maps. The working of the filters is represented on figure 1.14.

The output of a filter is called a feature map because a certain filter will be able to detect a specific feature. An example of this is shown in figure 1.15 with a kernel that is designed as a rudimentary vertical line detector.

After being applied on the first section of the input matrix, the kernel will slide by a stride on the matrix. Finally, the padding is used in the case the input matrix does not fit the size of the kernel. This way, setting extra zeros at the extremities of the input matrix allows the output to have the same size or be larger.

Convolutional layers can exist in one, two or even three dimensions, one dimension convolutional layers are mostly used on time-series data while 2D CNN are more associated to visual applications. In this work we will only make use of 2D CNN. 3D CNN have a kernel that moves in three directions, they are useful for 3D images such as CT scans and MRI.

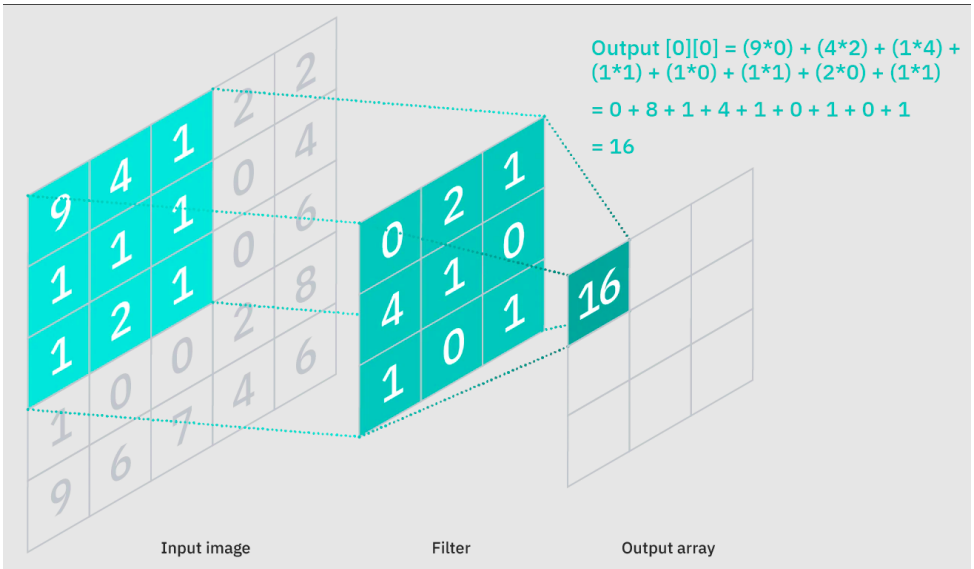


Figure 1.14. Convolutional layer, retrieved from ibm.com

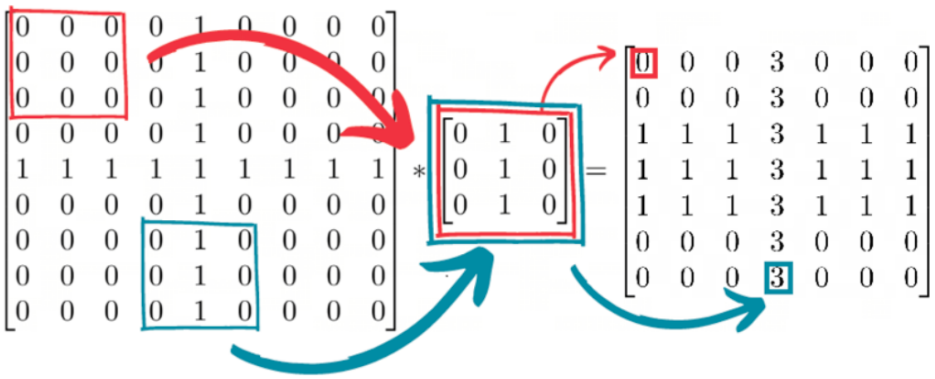


Figure 1.15. Vertical line detector kernel example, retrieved from deepai.org

1.8.2. Pooling layer

The pooling layer is the second most important element of the CNN. Pooling layers execute a down-sampling of the input matrix. The working principle of a pooling layer is the same as the one of the convolutional layer. It consists in a kernel the is slid along the input matrix but this time the kernel does not have any weights. There exist two types of pooling layers:

- Max pooling layer: This type of pooling only selects the maximum value in the window as output. This is the most common approach
- Average pooling: In this case, the output is the average of all the values in the sliding window.

This step leads to the loss of a significant amount of information but also reduces the number of parameters in the following layer and this way makes the network more efficient and reduce the risk of overfitting the data.

1.9. Overfitting / Underfitting

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 with 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 not being adapted to the type of 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.

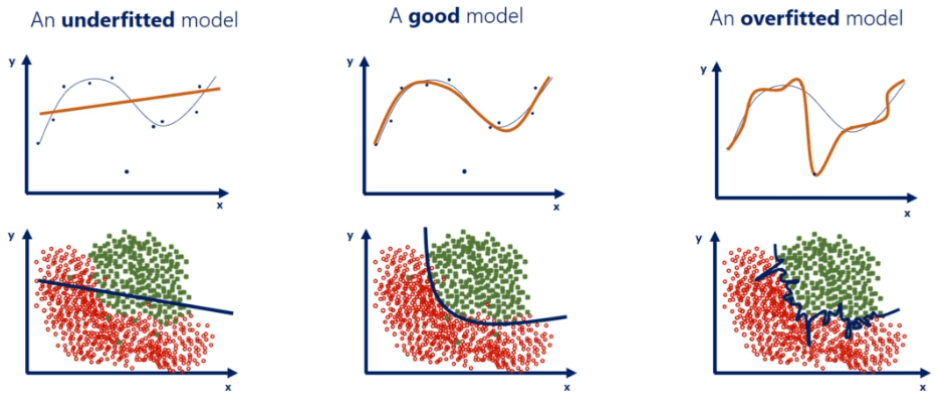


Figure 1.16. Underfitting vs overfitting, retrieved from skillplus.web.id

It is essential in reinforcement learning to find the right balance between overfitting and underfitting.

1.10. 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 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 [8], 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 [3], 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.

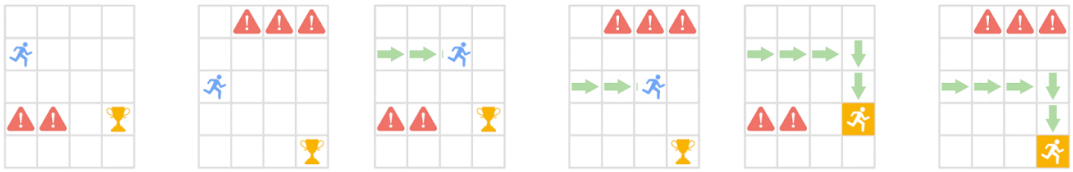


Figure 1.17. Behavioural similarity in observationally dissimilar states, retrieved from ai.googleblog.com

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.

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 done using the Python programming 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 Koen Boeckx for the IRIS project. Hence, our research has been done on the environment used in this contribution. The environment that was designed for it 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 [19] with the `PettingZoo` API. PettingZoo is a popular API which contains many environments, single or multi-agents designed for research in reinforcement learning, for multiple-agent 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 agents have destroyed all adversary agents and thus won the game and it equals -1 for the losing team. In all other cases, the reward is worth -0.01. 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

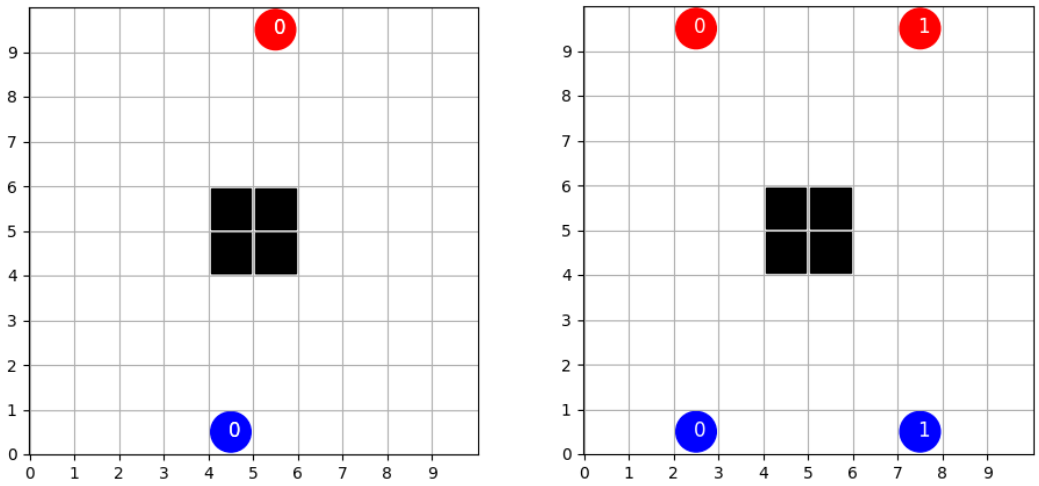
When all agents of one team have been destroyed or when the maximum number of steps is reached, an episode is terminated but no extra reward is distributed.

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

2.2.1. Environment

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 side for DQN and QMIX respectively. In order to assess the performance of the agents in way that does not favour overfitting of the agents against a particular strategy, 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

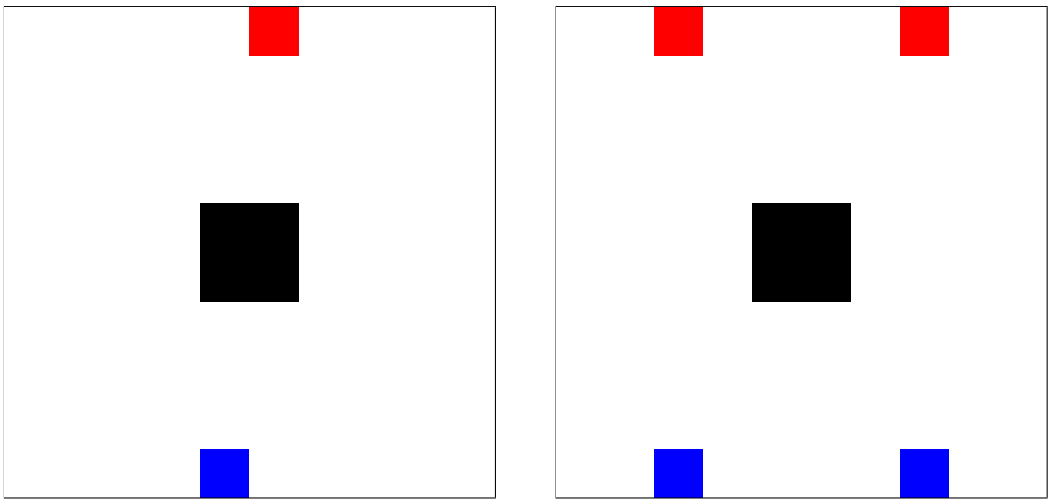
This is the basic environment that will be used first to train the algorithms and assess their performance as such before moving on to the generalisation part.
To evaluate the generalisation performance of our algorithms, the first step we will take is to evaluate their performance on environments that 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 probability that a grid point is an obstacle. 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.

2.2.2. Terrain difference quantification

In order to have an objective measure of the generalisation performance of the algorithms, we must think of a way to quantify the difference between two given environments. Therefore, we designed a method to compute a similarity metric that allows us to assess the terrain generalisation performance of our algorithms quantitatively. This method is based on existing image processing techniques, which makes even more sense since we worked with convolutional networks that consider observations as images as described in the following sections. The first step of the method is to take a terrain int its initial state, with all the agents being alive and on their initial position and to compose an image that represents it. This image represents each point in the grid as a white pixel. In case there is an obstacle at certain coordinates, the pixel in question becomes black. For agents, the pixel becomes blue or red in function of their team.

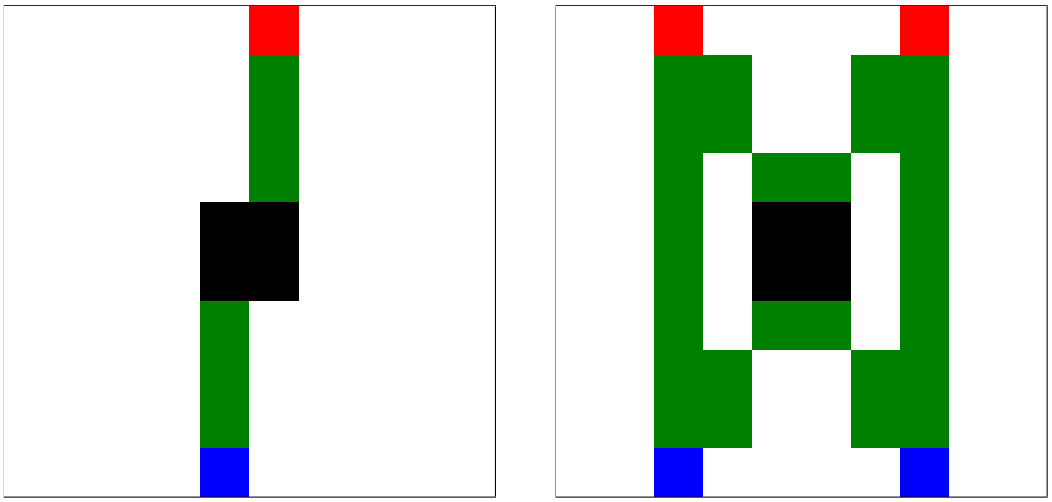


(a) Benchmark 1 vs 1 environment **(b)** Benchmark 2 vs 2 environment

Figure 2.2. Benchmark central obstacle environments in pixel RGB representation

To make the characteristic patterns of an environment even more distinguishable for the processing, an option is added that draws green lines-of-fire between each agent and all adversaries. This way, the similarity between environments that differ in obstacle positioning but have a very similar disposition of the agents can be better detected (see figure **??**).

These representations are then compared with each other using the Structural Similarity Index (SSIM). This is a metric that has been proposed in 2004 in [21] by Zhou Wang et al. This method, goes in another direction than most traditional image processing methods that evaluate perceptual image quality based on errors between images, exploiting properties of the human visual system. This new framework is based on estimating the degradation of structural information.[21] This property makes this method very well adapted to our application. The main assumptions of the authors of this method are that most other quality assessment techniques use the difference between a reference and a sample image. They often perform for example a *Mean Squared Error* on the difference between the values of corresponding pixels in a reference and a sample image. Secondly, since the human visual perception system performs very well in identifying structural similarities between images, a method that could replicate this will



(a) Benchmark 1 vs 1 environment **(b)** Benchmark 2 vs 2 environment

Figure 2.3. Benchmark central obstacle environments in pixel RGB representation with lines-of-fire

probably perform well at identifying similarities or differences between two images. [6] The metric is based on three criteria:

- Luminance: Luminance is measured as an average over the values of all pixels
- Contrast: It is the standard variation of the values of all pixels
- Structure: It basically is the value of the input normalised with the standard deviation

The value of the index can vary from -1 to 1 for images that are respectively very different to totally similar.

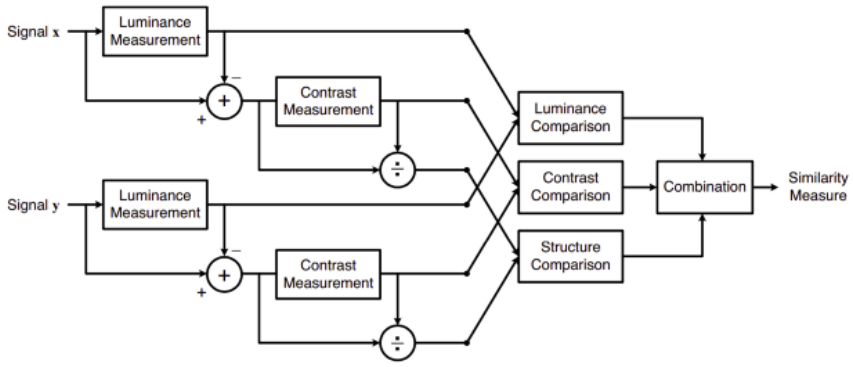
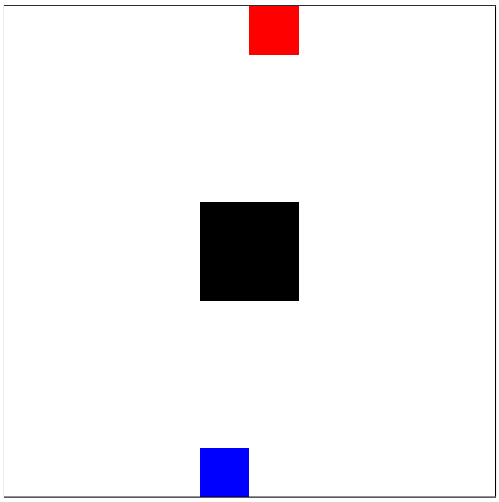
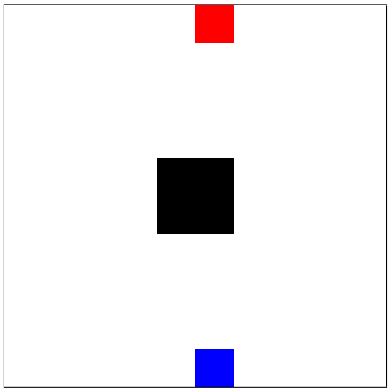


Figure 2.4. SSIM calculation diagram

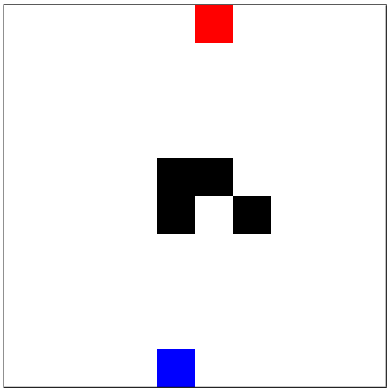
We computed this index using the `ssim` function of the Python library `scikit-image`. An example of the results of this function can be seen on figures 2.6 without the lines-of-fire option and on figure 2.6 with the lines option.



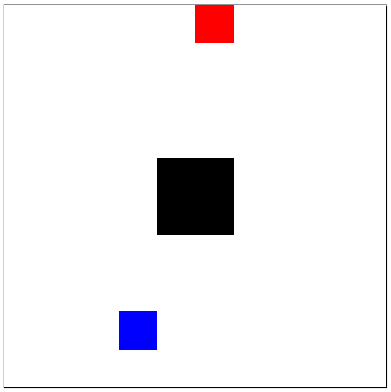
(a) Benchmark 1 vs 1 environment



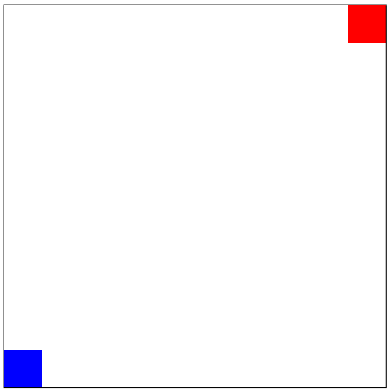
(b) Sample environment SSIM: 0.94



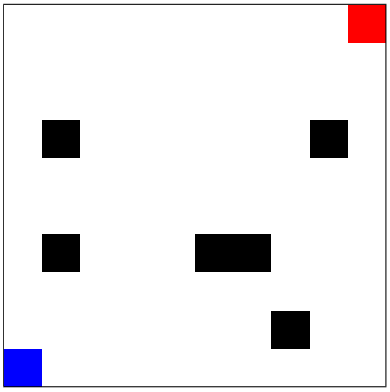
(c) Sample environment SSIM: 0.88



(d) Sample environment SSIM: 0.89

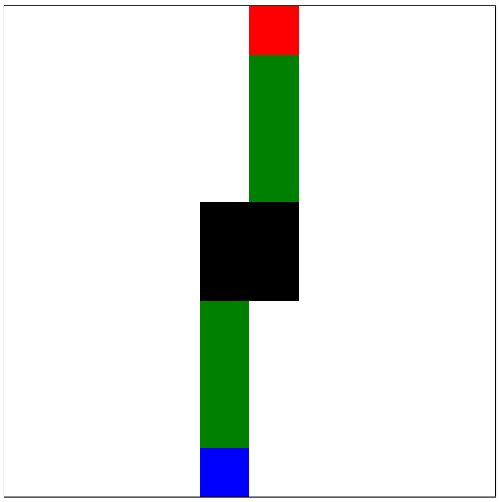


(e) Sample environment SSIM: 0.63

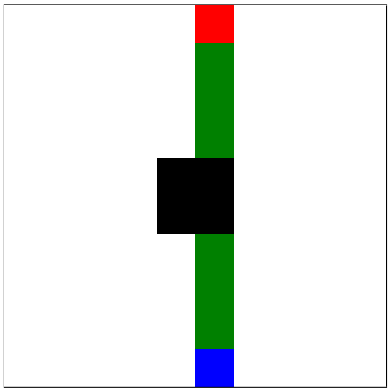


(f) Sample environment SSIM: 0.17

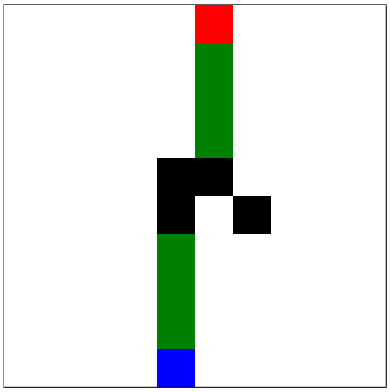
Figure 2.5. SSIM index computation on reference environment without lines-of-fire



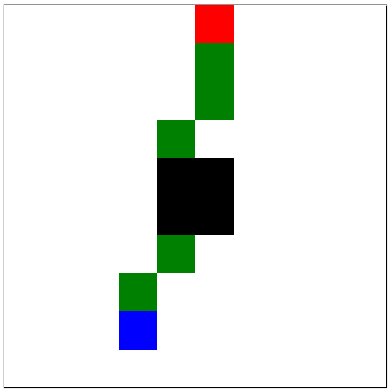
(a) Benchmark 1 vs 1 reference environment



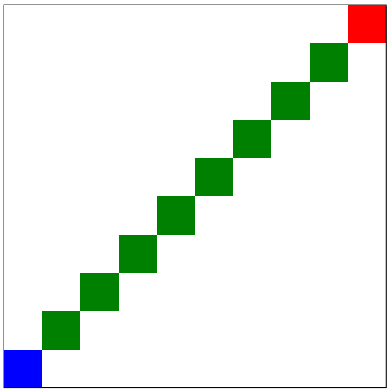
(b) Sample environment. SSIM: 0.79



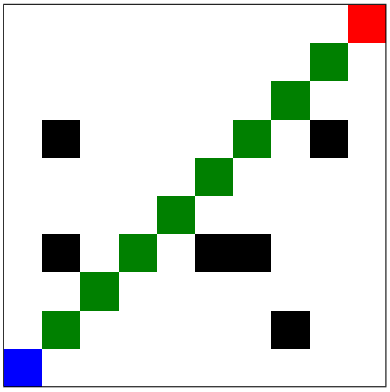
(c) Sample environment. SSIM: 0.88



(d) Sample environment. SSIM: 0.77



(e) Sample environment. SSIM: 0.38



(f) Sample environment. SSIM: 0.09

Figure 2.6. SSIM index computation on reference environment with lines-of-fire

As can be observed from both of the previously mentioned figured, the lines option allows to give more weight in the computation of the similarity index to the relative position of the agents rather than the placement of the obstacles. Indeed, we can see that in figure 2.6, without the lines, sample environment (b), where only the position of the blue agent has been shifted by one pixel to the side is the most similar to the reference figure. While terrain (c), which only has an obstacle shifted to the side is less similar to the reference environment according to the metric without lines. Nonetheless, when the line option is enabled, the shift of the agent to the side modifies the line-of-fire, making environment (b) less similar to the reference environment than sample terrain (c). It is also interesting to note that environments (e) and (f) go from a SSIM index of 0.47 without lines to 0.62 with the lines option enabled. Showing once again the same characteristic. This feature thus allows us to fine-tune the analysis of our algorithms' performances according to our preferences. For the rest of our work, we will by default use the metric with the lines option disabled if not mentioned otherwise.

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 end, we first run the environment while letting our agent act randomly until we have 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 since 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 $\alpha = 10^{-4}$. 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 10^{-3}, after some trials we decided on 10^{-4} for our application. The discount rate γ is set to 0.9.

For the $\epsilon - greedy$ action selection, we worked with a linear ϵ decay starting from 1 and decreasing until 0.01 over 100 000 steps for a training that lasts 300 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.

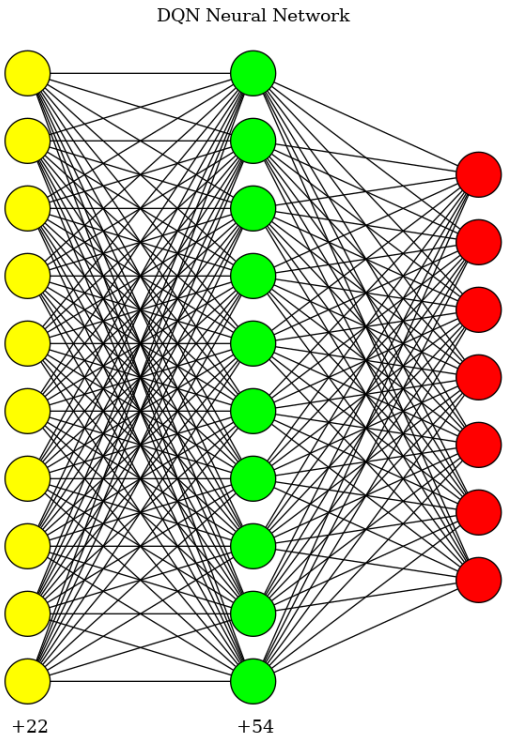


Figure 2.7. DQN neural network representation

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 based 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 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. The total state of the environment is composed of all the observation of the agents of a team at each timestep. 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 $n \times m$ 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 mixing network.

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 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 1.7. The optimisation is performed at each step and on all the hypernetworks 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 we work in a fully-cooperative setting, 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. Besides, if any transition where an agent is terminated has 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.

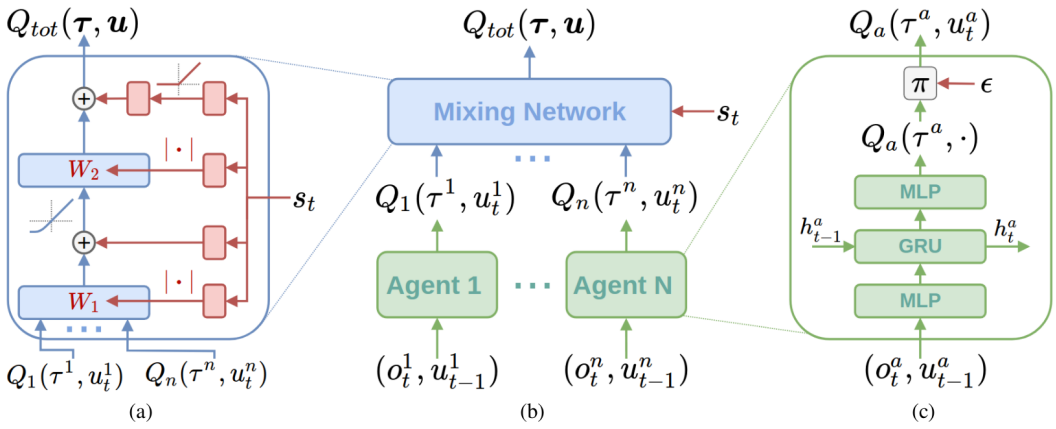


Figure 2.8. QMIX diagram [13]

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. Value Decomposition Network

The Value Decomposition Network (VDN) is another multi-agent reinforcement learning algorithm presented by P. Sunehag et al in [15]. This approach, similarly to QMIX is a centralised training, decentralised execution approach that works in an analogous way. In the case of VDN, a total action value is calculated as well but this time as a sum of the action values of the agents. The gradient is then back-propagated on this sum to train the agent networks to achieve a common policy. This algorithm is efficient as well but the simple representation of the Q_{tot} action value drastically limits the complexity of the functions that can be approximated.

2.5.1. Implementation

To implement the VDN, the code of the QMIX can be reused with very few changes to be made. The hypernetworks of the mixer can be deleted, which speeds up the code. Then the mixer function can be adapted to simply sum the Q values over all the agents of the team. The rest of the implementation is the same.

2.6. Training on randomly generated environments

The first technique we intuitively designed to improve the adaptation capacity of out algorithms rests on the principle proposed in [8], which proposes to make the training more similar to what the algorithm will face during testing. Therefore, a solution is to train the algorithms over the same number of steps as during the benchmark and using the same parameters but periodically 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. Hence it will most likely display worse performance on the benchmark environments than algorithms that were exclusively trained on those.

However, we found a way to circumvent the fixed observation size limitation with the next feature we implemented for both our algorithms: using a conventional neural network to process the observations. Implementing this feature, we are now able to use a randomly generated training library of 10 terrains that contains the benchmark terrain on which the algorithms will be trained.

2.7. Conventional Neural Network as input layer

2.7.1. Converting observations to images

This section is one of the most important and interesting parts of this work. By changing the observation format, we allowed our network to take as input observations containing a variable number of obstacles or even agents. Indeed, as explained in the previous chapter, CNNs are very efficient for image processing. Hence, the idea is to convert the observation array that the original environment feeds the agents into multiple images, using the same principles as image processing with RGB images. This is already a widely studied field and examples of this type of neural network are numerous.

In our case, we decided to create 8 2D matrices for each observation that represent the grid of the environment. These matrices are all filled with zeros and each contain a certain type of information under the form of a numerical value at the coordinates of the agent or obstacle in question. The 8 images represent respectfully:

- The obstacles as 1s
- The observing agent with a 1 if alive and a 0 otherwise
- The teammates of the observing agent if alive
- The adversaries if alive
- The amount of ammunition left on each teammate (including the observer)
- The amount of ammunition left on the adversaries
- Whether the teammates are aiming (including the observer)
- Whether the adversaries are aiming

This choice of representation is the result of much trial and error and is probably perfectible. Nonetheless, it proved to still allow the algorithms to converge during training although in more steps than using the basic representation. Ameliorating this representation might be a key to a much better generalisation of the algorithm.

2.7.2. Designing CNNs

The part of this solution is to design an efficient CNN model to process the transformed image input. We opted to use the same agent network designs as used in DQN and QMIX, namely an linear DQN and an RNN respectively preceded by a CNN to process the input. The CNN is composed of a 2D convolutional layer with 8 input channels, corresponding to the 8 layers of 2D information coming from our transformation function, and 16 output channels. The kernel is of size 1 with stride 1 since information is in our case contained in separate pixels of low values. We did not add any padding at this layer. This layer is followed by a max pool layer and another 2D convolutional layer with a 4 by 4 kernel, stride 1 and 64 output channels. Finally, a max pool forms the lasts layer after which the data is flattened to be transferred to the agent's actual DQN.

2.8. Improving DQN performance

The limitations of the method we presented in previous section, namely training on randomly generated environments mostly have to do with the limited amount of training the algorithm ends up getting. Henceforth, an intuitive way to compensate for this weakness in our proposed solution would be to increase the performance of the DQN such that it is possible to train it up to the same level with a lower number of steps. To this end, we surveyed the literature for techniques that can make a DQN more efficient. In this section we will review some of them and how we will implement them in our specific setting.

2.8.1. Prioritised Experience Replay

The `defense_v0` environment presents a few particularities that we need to take into account when implementing our RL algorithms. Firstly, it has a big observation and action space already with our benchmark 10 by 10 terrains with 1 agent on each team which makes that it needs a significant amount of steps to explore the (*state*, *action*) space in a decent manner. Indeed, each agent can find itself in dozens of different positions and perform many different actions such as moving in all directions, aiming at any of the opposing team's agents or firing. So for the agent to randomly take at least once, each of these actions, it will statistically take an important number of steps. Moreover, when the agent manages to randomly take a sequence of actions that allows it to get a significant reward, namely a +1 reward for aiming at and then shooting an agent, which happens rarely compared to all the other possible transitions, this transitions still has to be selected from the replay memory during the random sampling. This phenomenon is even more acute in environments that contain several agents per team as the agents have to repeat this action several times to eliminate all the agents on the opposite team,

making the probabilities of occurring some orders of magnitude smaller. With a memory buffer of limited size, hence acting as a sliding window, those probabilities become almost negligible.

With all these probabilities combined, we end up seldom training our model on these most interesting transitions while these are the only ones that lead to winning the game.

This is where Prioritised Experience Replay (PER), introduced by T. Schaul in 2015 in [14], becomes interesting.

Prioritised experience replay is based on this idea that some transitions are more relevant than others in the training of our model. In PER, this importance is expressed as a priority value from which we can derive the probability, for each transition, of being chosen during sampling. The probability of transition i being sampled from a k transitions batch thus becomes:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.1}$$

With the priority being $p_i = |\delta_i| + \epsilon$. Here δ_i represents the error of the DQN or the QMIX as seen in equations 1.9 and 1.7. ϵ is an arbitrarily chosen small constants that is used to ensure that the priority of a transition cannot be null and hence all priorities have a chance in theory to be sampled, it is fixed at 0.01 in our implementation. α, on the other hand is a parameter that allows to reintroduce randomness in the process by dampening the priority differences between the transitions. Indeed, an α equal to 0 would make all the transitions equiprobable while a value of 1 would not make any difference. We chose a value of 0.6 in our case.

Nonetheless, by using prioritised experience replay, we will tend to use the same transitions over and over for training, increasing the risk of overfitting the model to those experiences. To compensate for this non-uniform distribution and anneal the bias, weights are assigned to transitions in addition to the priority values. The weights, calculated according to the following formula are smaller for transitions with higher probabilities:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{2.2}$$

N stands for the number of transitions in the replay memory. Once again, β is a parameters chosen to damp the effect of the weights similarly to α for the probabilities. We chose β equal to 0.4. In the original paper, they also recommend that β be linearly annealed from an initial value to 1 to make the correction more aggressive only towards the end of training, in a way similar to the $\epsilon - greedy$ policy. This way, towards the end of the training, the algorithm totally compensates for the bias. In practice, weights are always normalised with $1/\max_i w_i$ so that they only scale the parameters correction downwards. Finally, in the paper, they implemented their prioritised experience replay buffer with a Double DQN, which is discussed in next section.

Implementation

The implementation of PER in our code was realised by using a priorities buffer and a weights buffer in parallel to the replay memory buffer. During the replay memory initialisation phase, all priorities are set to 1 when storing a new transition. After this, during the training of the model, each time a new transition is stored, it takes the value of the maximum priority in the buffer. At each step, the replay memory is also sampled according to the distribution derived from the priorities buffer (we use the `numpy.random.choice` function with the probabilities array as distribution to get the transition indexes) and the error is calculated for each transition as in the normal DQN or QMIX implementations. Nonetheless, in this case the error is first used to update the priority of the sampled transitions and then multiplied with the normalised weights before being used to optimise the model over the loss. This way, recent transitions automatically get a high probability, which will ensure that they probably get sampled and thus updated within the next training steps and get a priority corresponding to their

relevance. This way of implementing also avoids the need to loop through all the transitions at each step to update the priority values and directly assigns priorities that ensure that useless transitions do not get sampled again while important transitions will be resampled.

2.8.2. Double DQN

The last method that we implemented in order to improve the performance of the algorithms is the Double DQN (DDQN). The idea behind the Double DQN, which was introduced in paper [20], is that the Q-values as calculated in the original form of the Q-learning algorithm is systematically subject to significant positive overestimation. This is mostly due to noise in action values and to the max function that is used in equation 1.8. Indeed, this operator increases the probability to select overestimated Q-values. These overestimation errors were already a known phenomenon however the authors demonstrated that they actually were very frequent and had a negative impact on performance and even could lead *asymptotically to suboptimal policies*. Yet they also proved that it was possible minimise them with Double DQN. The principle of Double DQN is to decouple the selection of the action and the evaluation of the action value. In basic DQN, both are performed by the max operator on a given network. In double DQN, the online network will determine the greedy policy while the target network will evaluate the action value. The target value can hence be mathematically represented as following:

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname{argmax} Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right); \boldsymbol{\theta}_t^{-}\right) \tag{2.3}$$

Implementation

Starting from our code, implementing the double DQN is quite straightforward. Instead of performing an `argmax` operation over the output of the target network, we selected the action that maximised the Q-value as calculated by the online network. For the QMIX, a similar adaptation was performed on the Q_{tot} function. Then we kept periodically updating the target network with the online network as already implemented. This additional feature is expected to dramatically improve the DQN performance in all algorithms even if we have not found any literature describing this implementation in QMIX or VDN.

2.9. Partially observable environments

A last aspect that is extremely interesting to study the operationality of a model is to test it in partially observable environments. In all the previous sections, we considered that each agents had access to all information about the environment at all times when observing. In the case of a partially observable environment, only part of the information of the environment is available to the agents. In our case, to make the environment partially observable, we will delete the information that indicates to the observing agent whether the adversaries are aiming and the ammo an adversary agent has left. This will be done in the convolutional representation by leaving all the matrices in question equal to zero. We will then study to what extent deleting this information impacts a model and hence which algorithms are most robust to this.

3. Results

In this chapter, the results of our work are presented.

3.1. Benchmarking

First, the algorithms were trained on our benchmark environments according to the methods described in previous sections. The curves shown on the plots are the data on which a uniform 1-D filter has been applied to smoothe the curve and better show its tendency. This filter takes for each point the arithmetic average over a specified window. To better show the stability of the curve, the interval formed by this average plus and minus a multiple of the standard deviation is plotted around each curve.

3.1.1. DQN

The loss does not augment much and diminishes very fast in the case of the DQN to oscillate around very small values (comprised between 0.02 and 0.001). This can also be attributed to the simple architecture that we chose for our DQNs. We can further observe that the convolutional DQNs have lower loss values that the regular implementation. The double DQN alone seems to raise the mean value of the loss but combined with the prioritised experience replay, a clear diminution can be observed. Which is even more pronounced when annealing the β value of the PER.

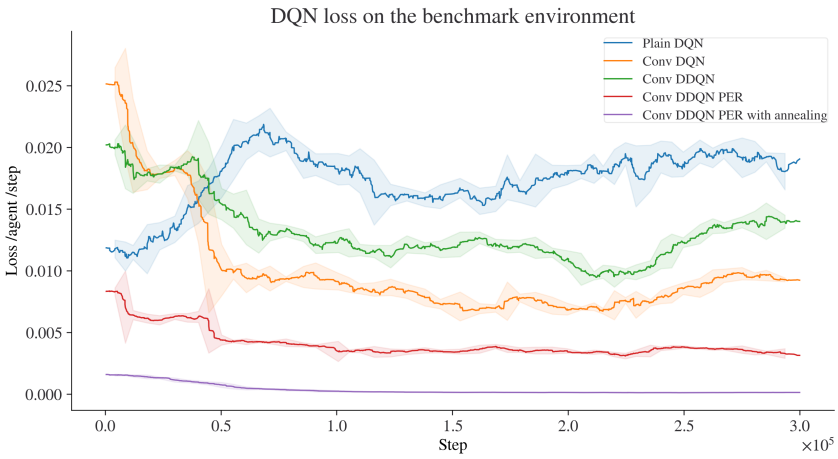


Figure 3.1. DQN loss during benchmark training

The reward in function of the step in the learning process gives a more representative idea of the performance gain that is achieved using convolutional networks, Double DQN and PER. The basic DQN implementation serves here as a baseline to compare the loss or gain that is made at each step. This first implementation stabilises quite fast while the basic convolutional DQN takes much more steps to reach similar levels of mean reward. The double DQN makes the convolutional implementation significantly faster but is less stable and tends to oscillate more. Finally, we see that the PER without annealing does not deliver satisfying performances and is so unstable that it does not necessarily converge. We will thus not focus ourselves on this implementation for the rest of this work. On the other hand, the PER with annealing coupled with the Double DQN shows a significant performance improvement with

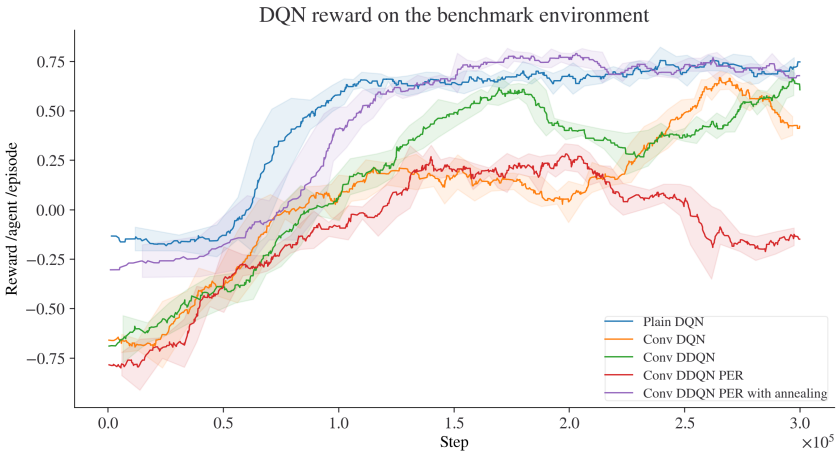


Figure 3.2. DQN reward during benchmark training

a reward that takes more time to converge towards the ideal policy but quickly catches up to similar mean reward levels and is much more stable than previous convolutional implementations.

The plot of the number of steps per episode during training shows the same tendencies as the reward and even shows that with the DDQN and the PER, the convolutional DQN reaches even better performance than the plain DQN after a third of the training is performed. This step actually corresponds to the end of the $\epsilon - greedy$ decay and of the annealing of the PER's β value. This can explain the sharp slope of the curve when reaching this step.

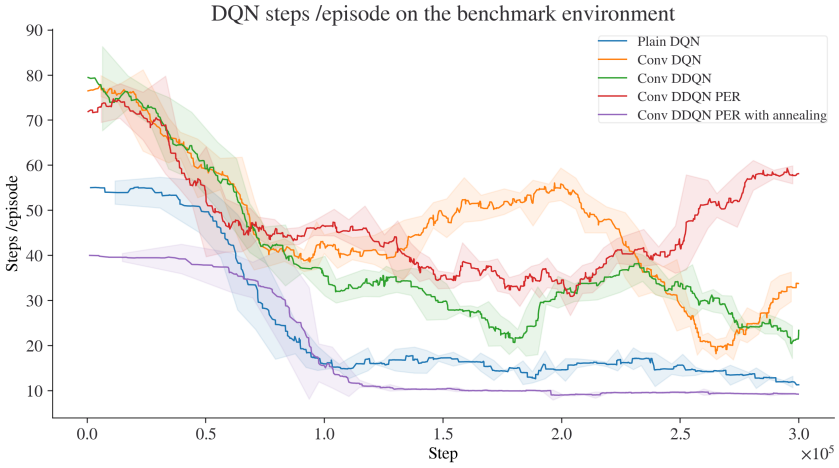


Figure 3.3. DQN n° of steps during benchmark training

Finally, the win rate plot is very similar to the reward plot as these are very much linked and thus show the same tendency.

To conclude, the Double DQN and the PER dramatically increase the learning speed. However, as a result, we noticed that the performance of the algorithm over the training environment in test setting, without the $\epsilon - greedy$, in some cases started to drop after 100k steps, that is at the end of the $\epsilon - decay$. This is most likely due to the fact that we kept the same learning rate for all the training processes. Therefore, to test out the DQN improved with the DDQN and PER, we used the parameters that were obtained by using a 10^{-5} learning rate during training instead of 10^{-4}. An ideal solution would be to implement a scheduler for the learning rate so that it would be adapted during training to optimise the the efficiency of the training.

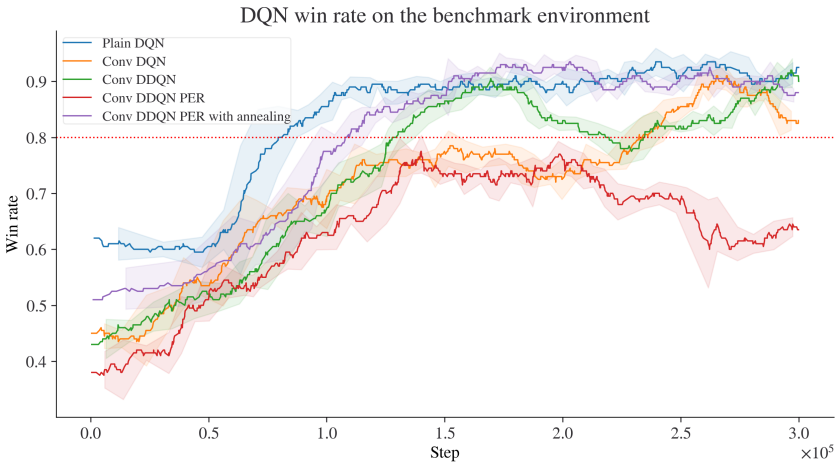


Figure 3.4. DQN win rate during benchmark training

3.1.2. QMIX & VDN

The QMIX and VDN implementations did not yield good results on terrains that contain more than a single agent on each team. Indeed, the algorithms did not converge towards an optimal solution in this case on the `defense_v0`. The reasons behind this phenomenon are not known and must be further investigated in a follow up work. It could be due to a need for parameters optimisation as this process is mostly comprised of trial and error and these algorithms take a lot of time to train. Another possible cause would be that the algorithms need more time to train than we tried using the resources at hand. Training the algorithms for a higher amount of steps would require too much time or faster computers. Therefore, we present in this section the result of the training of VDN and QMIX and the performance increase we obtained by adapting DDQN and PER to these algorithms and explain how these performance gains can be exploited using the simpler case of the DQN which is an essential component of both algorithms.

3.2. Evaluation over randomly generated environments

3.2.1. DQN

Linear vs convolutional

The first test we performed was intended to have a first idea of the generalisation capacity of the basic linear DQN implementation. To this aim, we generated a small library of terrains based on the benchmark training terrain. Due to the fixed input size limitation of the linear DQN, we could only generate terrains containing the exact same number of agents and obstacles. For this reason, the test library, composed of 10 environments is relatively small. The algorithms were tested on each of these environments and the performances plotted are an average of the results obtain over 50 episodes to have a representative sample. The standard deviation on each environment is represented with the error bars.

The results of this test are shown in figure 3.5, 3.6 and 3.7. T The first observation we make is that there is a clear tendency for the performance to improve with the similarity index. This confirms our hypothesis regarding training and the relevance of our metric. This tendency is a lot more pronounced for the basic DQN implementation which was expected to have a worse generalisation performance by design. The curve of the convolutional DQN implementation, on the other hand, is much less steep as expected as well. The tendency of the curve has almost disappeared in with the convolutional DQN. Our similarity metric might be better exploited in bigger, more complex environments as the environments used in the scope of this study contain so few elements that they do not require complex or much

different policies to be solved.

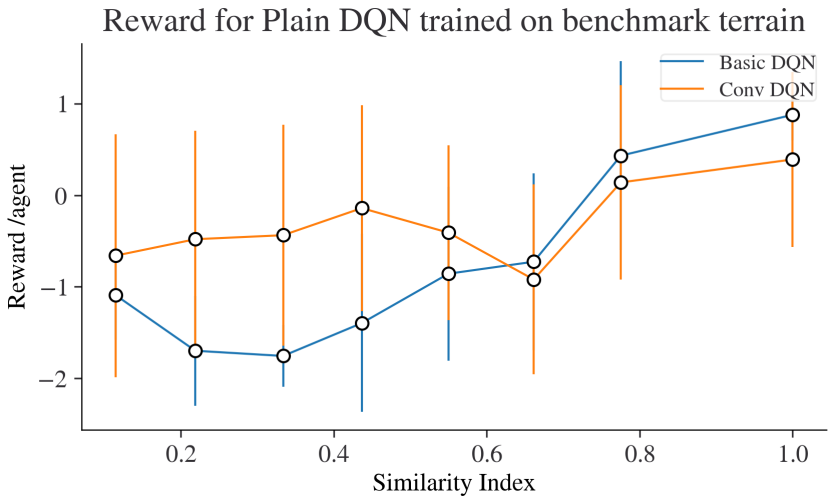


Figure 3.5. Plain DQN reward over small terrains library

We still notice a few outliers around 0.66 or 0.

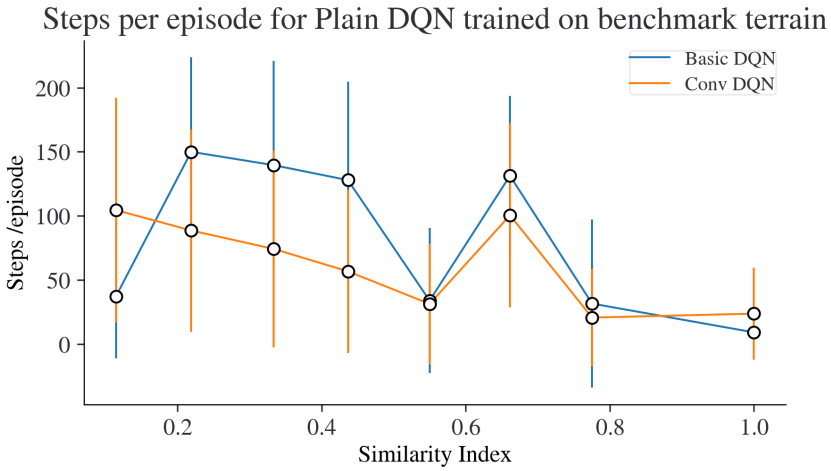


Figure 3.6. Plain DQN n° steps over small terrains library

The steps curve does seem to be more unstable but this can be explain by the fact that the duration of an episode does not necessarily indicate the performance of the algorithm as the agent can also be killed quickly by applying its learned policy. This is why the outliers are a lot less significant in the reward and win plots.

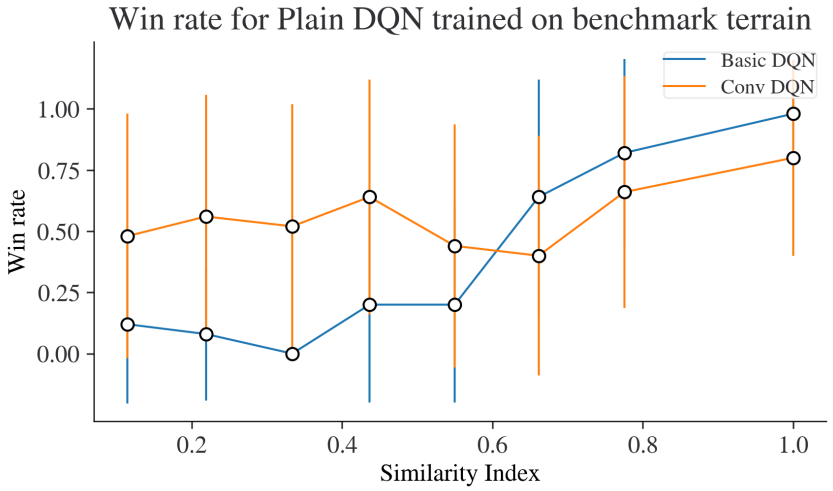


Figure 3.7. Plain DQN win rate over small terrains library

On average, the convolutional DQN shows a superior overall performance over the whole test set (see table ??). The basic DQN has an overall win rate of 38% and thus becomes even worse than a random policy that would yield a 50% win rate for a 1v1 scenario with an opponent applying a random strategy. While the convolutional version retains an advantage.

	Average reward	Average episode duration (in steps)	Win rate
Linear DQN	-0.78	82.56	38.0%
Convolutional DQN	-0.31	62.54	56.24%

Table 3.1. Plain DQN performance statistics over fixed obstacles number test set

This first test confirms our hypothesis concerning our similarity metric and the gain that the convolutional DQN implementation represents for our research. Now we will further investigate the different means we implemented to improve our convolutional implementations.

Improving the convolutional DQN

As we previously demonstrated, the best way we developed to improve our convolutional DQN implementation was to combine the Double DQN with a prioritised experience replay on which β annealing was performed. Therefore, and also for time limitations, we will in the following section only consider this last version to assess the gain it yielded in comparison to the plain DQN implementation. The training of these more and more complex algorithms is indeed very demanding in terms of processing. From this step, we used as test set a library of 241 randomly generated environments with similarity index relative to the benchmark environment going from 0 to 1 to study the evolution of the performance in function of the similarity index. We also fixed the maximal probability of a grid point in the terrain to be an obstacle to 0.3 to avoid considering terrains that are too clogged for the agents to apply a non-trivial strategy. We first compared the performance of the plain convolutional DQN to the one of the improved version (see figures 3.8, 3.9 and 3.10).

On the first graph, we can confirm our previous observations as the upwards tendency of the curves is almost nonexistent here in comparison to what the linear DQN yielded. Furthermore, both curves here seem very close to each other so no clear generalisation performance gap can be distinguished.

The same observations can be made for the last two figures.

It appears that the improvements in learning speed our last version of the DQN achieved, lead to overfitting on the benchmark environment and thus do not improve the generalisation performance as such. Although the performance on highly similar environments stays superior. This hypothesis is confirmed by testing both models on the benchmark environment for 200 episodes (see table ??).

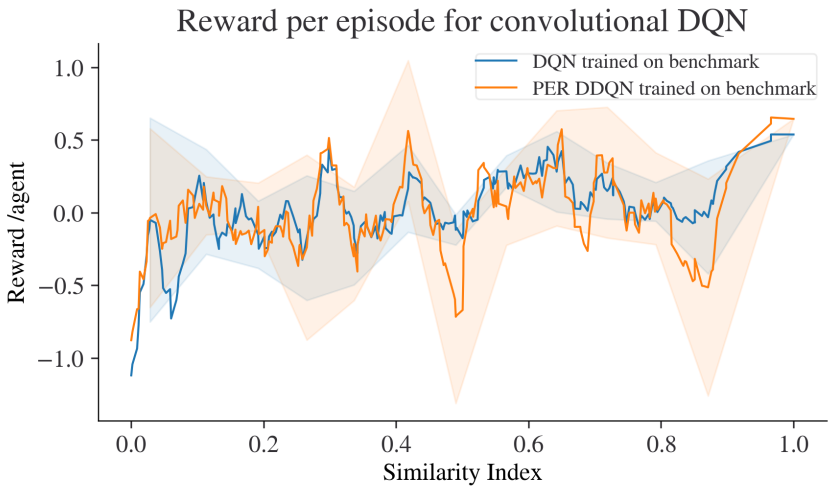


Figure 3.8. Convolutional DQN reward over terrains library

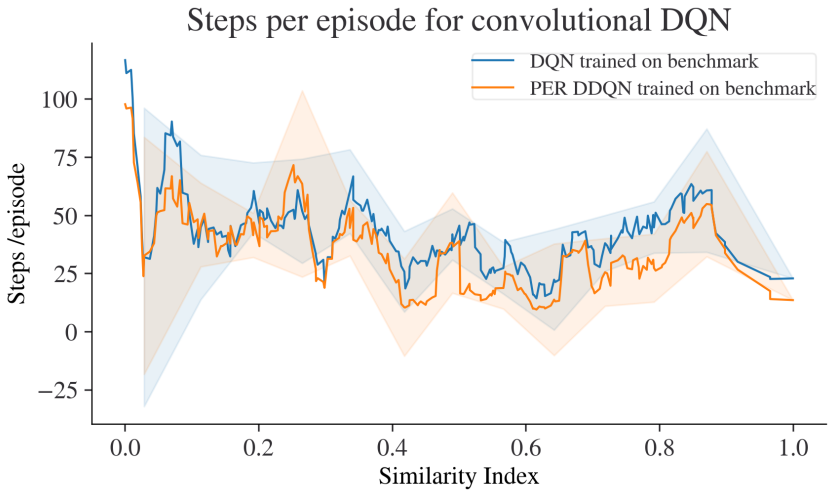


Figure 3.9. Convolutional DQN number of steps over terrains library

	Average reward	Average episode duration (in steps)	Win rate
Plain Conv DQN	0.47	25.46	83.50%
PER Conv DDQN (with β annealing)	0.73	8.37	90.0%

Table 3.2. Convolutional DQN trained on benchmark performance evolution over benchmark terrain

The PER DDQN does perform significantly better on the benchmark environment than the plain DQN implementation. The reward as well as the win rate are largely superior with this enhanced algorithm. Its performance on the entire test set (see table ??) is clearly worse than the performance of the basic DQN. Those two results combined confirm our hypothesis of overfitting of the benchmark environment with the PER DDQN.

	Average reward	Average episode duration (in steps)	Win rate
Plain Conv DQN	0.01	43.50	66.19%
PER Conv DDQN (with β annealing)	-0.03	35.07	60.12%

Table 3.3. Convolutional DQN trained on benchmark performance evolution over terrains test set

Nonetheless, the learning speed of the PER DDQN can be exploited in a more ingenious way. Indeed,

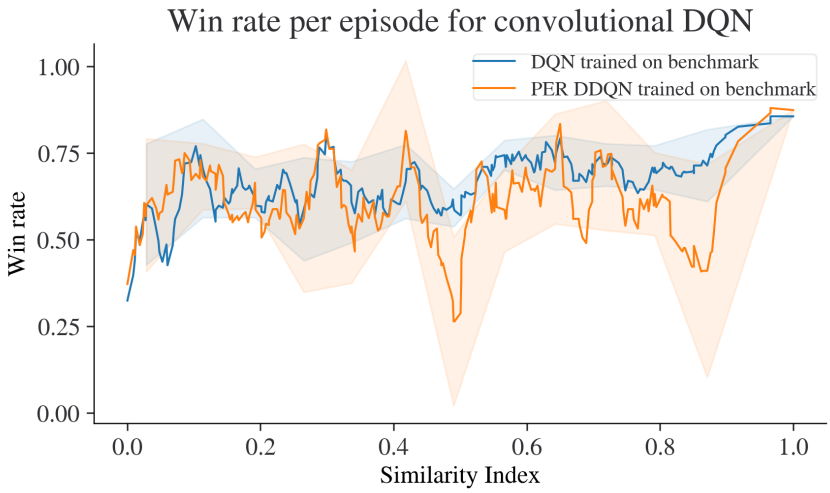


Figure 3.10. Convolutional DQN win rate over terrains library

given it learns faster, it should eventually yield better performance than the plain DQN when trained on the terrains training set. This is indeed the case as can be clearly observed on graphs 3.11, 3.12 and 3.13.

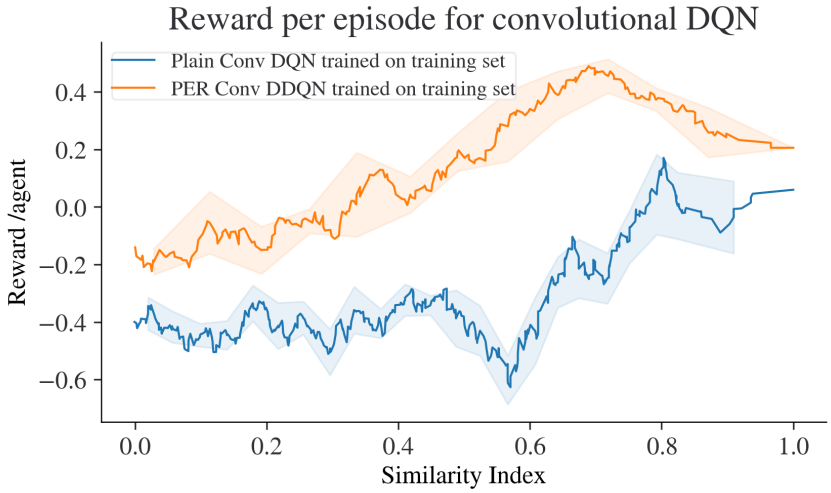


Figure 3.11. Convolutional DQN reward over terrains library

We see a clear performance increase with the PER DDQN when training on the entire training set. The learning speed indeed makes a dramatic difference when training over rapidly changing terrains as we expected. Another interesting observation is that the same upwards tendency of the curve as with the plain linear DQN can be observed again. Although the reason for this is not totally clear, it is certainly in significant part due to the fact that the benchmark environment is part of the training set.

We then compare the performance of all the variations together on graphs 3.14, 3.15 and 3.16.

As already stated, the plain DQN trained on the training set has the worse overall performance over the test set. This is in part due to the amount of training it received. Over many more training steps, it would most likely catch up with the PER DDQN which trains a lot faster. Next, the PER DDQN trained on the benchmark terrain has the second worst performance due to the overfitting of the model to its training terrain. Then there is a clear performance gap between these two models and the plain DQN trained on the benchmark terrain and the PER DDQN trained on the training set. These last two models perform the best with very similar win rates but the fourth implementation's efficiency makes it finish episodes much faster and thus collect significantly higher rewards.

The results are summed up in table ??.

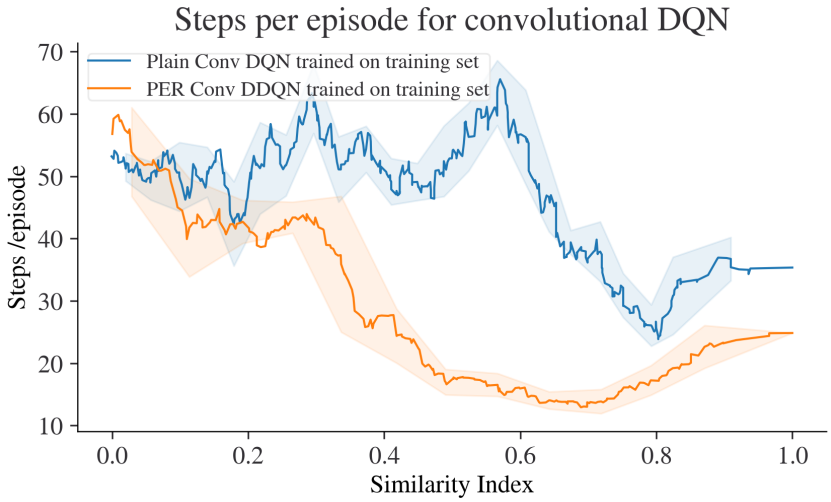


Figure 3.12. Convolutional DQN steps over terrains library

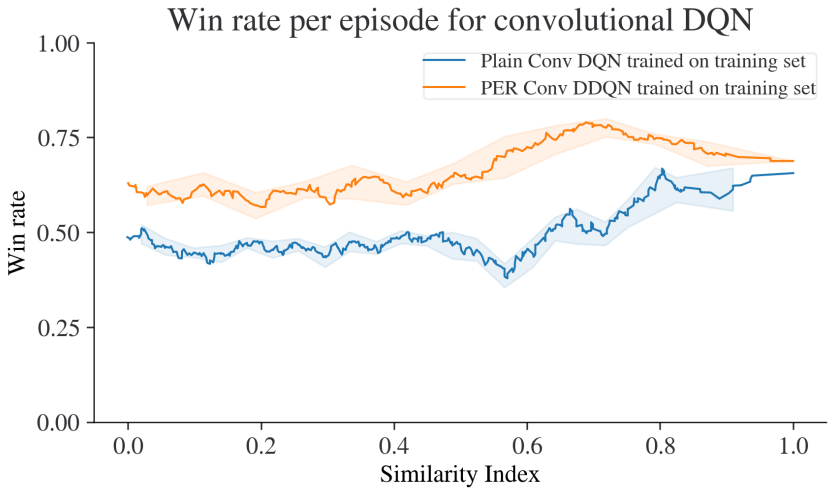


Figure 3.13. Convolutional DQN win rate over terrains library

	Average reward	Average duration	Win rate
Plain Conv DQN trained on benchmark	0.01	43.50	66.19%
PER Conv DDQN (with β annealing) trained on benchmark	-0.03	35.07	60.12%
Plain Conv DQN trained on training set	-0.32	47.47	48.98%
PER Conv DDQN (with β annealing) trained on training set	0.13	28.61	66.41%

Table 3.4. Convolutional DQN performance evolution over terrains test set

The results of these tests confirm the assumptions of previous chapters and allow us to conclude that training the algorithms over different environments does indeed improve the generalisation performance of the model. However, the amount of model in the training set and the duration of the training must be carefully balanced out taking into account the training speed of the algorithm. Implementing training speed enhancing features such as PER and Double DQN allow us to push towards bigger training set and get even higher generalisation performance gains.

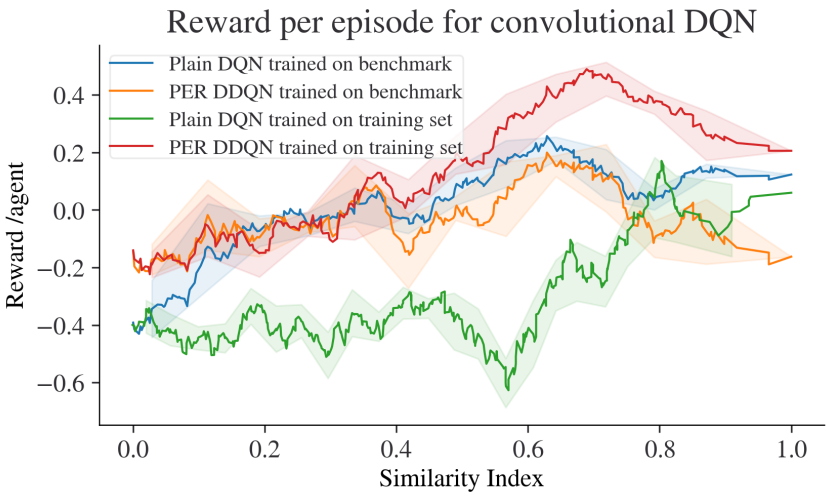


Figure 3.14. Convolutional DQN reward over terrains library

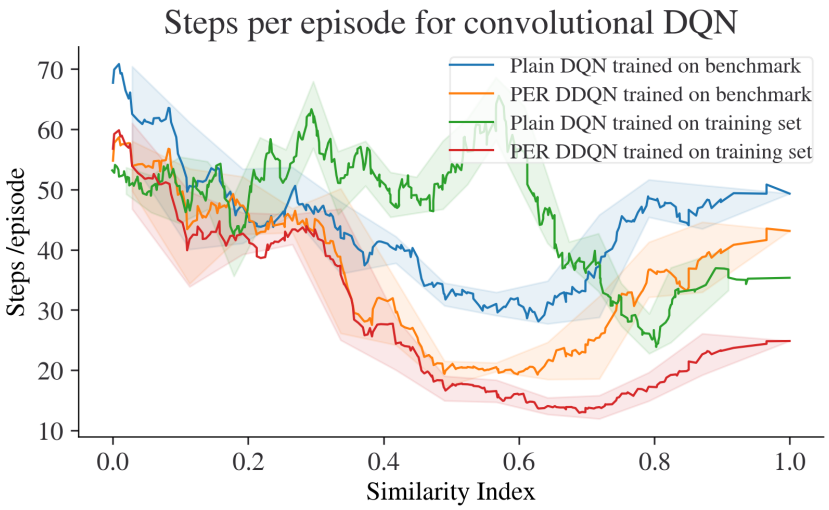


Figure 3.15. Convolutional DQN steps over terrains library

3.3. Robustness to partially observable environments

The last aspect we studied in this work is the consistency of a model's performance in an environment that becomes partially observable.

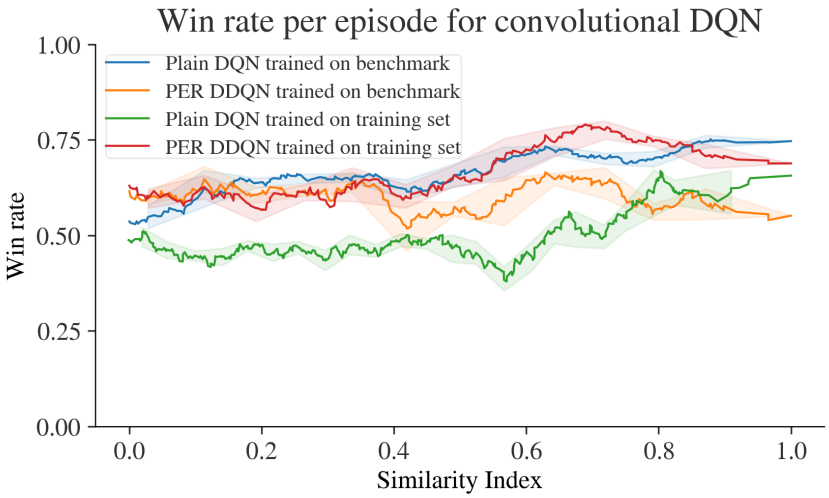


Figure 3.16. Convolutional DQN win rate over terrains library

4. Conclusion

The main goal of this research was to study the generalisation capacity of multi-agent reinforcement learning algorithms in the scope of the `defense_v0` environment, developed in the framework of the IRIS project.

To this aim, we developed a metric that allowed us to quantitatively compare various terrains in order to study the performance of the algorithms implemented. This metric was based on the existing Structural Similarity Index and yielded good results for our applications. We suggest however that it would be even more accurate when used on bigger and more complex environments than the ones seen in this work. This could thus be seen as a proof of concept for this metric.

The algorithms we used were the DQN, VDN and QMIX. The next step was to study the performance of these algorithms on a set of randomly generated environments. To be able to use our model in many different environments, we implemented a conversion function that changed the observation vectors yielded by the environment in images that we processed by convolutional neural network layers before being fed to the algorithms. This representation is one of the most important aspects of this research. Then we implemented certain features that enhanced the learning speed of the algorithms, namely the Double DQN and a prioritised experience replay. This speed increase lead to overfitting when training the model on only one terrain for the same number of steps as the basic implementations but allowed us to, when train it on varying environments, obtain a significantly improved generalisation performance. Indeed, varying the environments with the basic implementations prevented the model to learn the optimal policy on any of the environments. The main lesson learned from this is the trade-off between the size of the training set and the number of learning steps, carefully taking into account the learning speed which can be dramatically increased with the presented techniques.

We also presented a way to apply the known techniques of DDQN and PER to the VDN and QMIX algorithms and by doing so, to greatly improved its learning speed. This is something we have not seen described in literature yet.

Finally, we studied the robustness of the various algorithms to a change in the environment's observability during testing. From the results of these tests it appears that This is an interesting features for models as they need this kind of robustness to be operational in uncertain environments. In the paradigm of military applications, this definitely is an important characteristic.

4.1. Motivation and use for Defence

We believe there is still much to investigate in the same direction for the sake of Defence. Developing models that can efficiently analyse various types of environments in which some elements of information might lack and quickly give options to the operator of a system would definitely be an asset. Computers are now far more efficient than humans in many tasks, therefore I believe they could be used to assist in taking quick decisions in a tactical context.

4.2. Future recommendations

For future work, we would recommend to search ways to optimise the representation of the information to make the convolutional input layers even more efficient. We believe that this is an essential component of making a robust algorithm. Would a more complex representation allow for better poli-

cies or would a simpler and more compact representation be more robust to observability changes and faster to train ? In the same framework, the CNN component of the algorithms could also be improved.

Of course, further working to solve the convergence problems of the VDN and QMIX algorithms would be necessary to complement this work.

Lastly, it would be interesting to train and evaluate the algorithms against adversaries that implement various tactics. These could use trained models or manually programmed strategies.

A. Supplementary Information

A.1. Adam alogrithm

Algorithm 1 Adam algorithm

Require: : α : Stepsize
Require: : $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: : $f(\theta)$: Stochastic objective function with parameters θ
Require: : θ_0 : Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / \left(1 - \beta_1^t\right)$ (Compute bias-corrected first moment estimate)
$\hat{v}_t \leftarrow v_t / \left(1 - \beta_2^t\right)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / \left(\sqrt{\hat{v}_t} + \epsilon\right)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Bibliography

[1] 12 types of neural network activation functions: How to choose?

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

[3] 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.

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

[5] Sukanya Bag. Activation functions — all you need to know! - analytics vidhya - medium, Feb 2021.

[6] Pranjal Datta. All about structural similarity index (ssim): Theory + code in pytorch, Sep 2020.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2016.

[15] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinícius Flores Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning. *CoRR*, abs/1706.05296, 2017.

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

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

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

[19] 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.

[20] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[21] Zhou Wang, Alan Bovik, Hamid Sheikh, and Eero Simoncelli. Image quality assessment: From error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13:600 – 612, 05 2004.

[22] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 2004.