ROYAL MILITARY ACADEMY

172nd Promotion POL

Captain Maxime Séverin

Academic year 2021 – 2022

2nd Master

A Variable-length Observation Encoder for Multi-Agent Reinforcement Learning

Ensign at sea second class Officer Cadet

Maximilien REMACLE

Royal Military Academy

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 1.72

24/5/2022

A Variable-length Observation Encoder for Multi-Agent Reinforcement Learning

Maximilien REMACLE

Preface

1. Abstract

This thesis is a follow-up to the work of Senior Captain Koen BOECKX [BOE20], who first developed a simulated tanks battlefield environment. He then implemented some reinforcement learning algorithms to train agents in it and see if they could develop interesting strategies in the simulated environment. The final goal of it being to apply the learned strategies in real life applications.

One of the limitations of the previously mentioned reinforcement learning algorithms is that they only work with environments where the observation of the state is a vector of fixed length. In this thesis, we present a method that allows to encode variable-length observations to fixed-length observations with a minimum loss of information. This technique allows to implement classical reinforcement learning algorithm on a wider set of environments. It might allow, for instance, to develop and train on more realistic environments, where the learned strategies are more applicable or provide better results in real life situations.

We first developed an environment that can provide both variable-length and fixed-length observations. We then successfully managed to design encoders using recurrent neural networks, that can encode variable-length observations into fixed-length observations. Afterward, we trained some agents in our environment using reinforcement learning algorithms. Those agents only had access to the observations encoded by the previously designed encoders . Finally, we compared the results with some agents learning from the same environment, but from the raw fixed-length observations provided by the environment.

2. Acknowledgments

First and foremost, I would like to give my warmest thanks to Senior Captain Koen BOECKX, for the help and guidance throughout all the stages of the writing (and coding) process of this thesis.

Furthermore, I would like to thank the second reader of my thesis, Mr. Sanjoy BASAK, for the advice and for the time he gave me.

Also, I would like to thank two of my friends and colleagues: the second lieutenants Yemen RHOUMA and Hadrien ENGLEBERT. We wrote our respective theses in parallel, about very similar subjects, and supervised by the same supervisor. We exchanged some of our findings and ideas, and I am sure that this collaboration has been beneficial for all of our theses.

Besides, I want to thank my class at the Royal Military Academy, the 172nd class of Polytechnics, for these five years together, rich in experiences, emotions and which made me grow enormously.

Next, I want to express my gratitude to my family, for their love, admiration and support. More particularly my parents, who have always encouraged me to be curious and interested about everything. And my sister Jade, who is a caring listener and a comforting counsellor.

Finally, I would like to thank my sweet Eloïse, who has been a great support to me, for this particular work and in all other aspects of my life, and without whom this thesis would certainly have been of inferior quality.

3. About the author

I am a 21 year old student at the Royal Military Academy (RMA). I started studying there in 2017. In 2021, I chose to specialize in network enabled capabilities, one of the two masters in polytechnics proposed at the RMA.

I am passionate about programming since I discovered it when I was in high school. I spent hundreds of hours late in the night learning to program on my own. Since then, my passion for it has only gotten bigger. Later on, I discovered the broad field of artificial intelligence and machine learning and I have become obsessed by it and, to me, reinforcement learning is the most beautiful part of it: machines performing in an environment, repeating actions that allow them to get higher rewards, and avoiding the actions that penalize them. Almost like a little child learning to live and make the most of its environment.

When I heard that Senior Captain Boeckx proposed to write a thesis in this area, I could only accept. I knew it was the perfect opportunity for me to dive into this part of artificial intelligence and better understand its working. And I was not disappointed. The whole writing process of this thesis, the hundreds of hours of coding, the research and discovery of state-of-the-art algorithms in this domain have all been a real pleasure.

Contents

Preface i
1. Abstract . . . i
2. Acknowledgments . . . i
3. About the author . . . ii

List of Figures vii

List of Tables ix

List of Abbreviations xi

1. Introduction 1
1.1. Background . . . 1
1.2. Literature Review . . . 2
1.2.1. (Deep) reinforcement learning . . . 2
1.2.2. Variable-length observation MDPs and encoder . . . 2
1.3. The code behind the words . . . 2
1.4. Notation . . . 2
1.4.1. Code in text . . . 2
1.4.2. References to equations . . . 3
1.4.3. Use of the plural form . . . 3

I. Models and algorithms 5

2. Reinforcement learning 7
2.1. Markov Decision Processes . . . 7
2.2. Concepts and definitions . . . 8
2.2.1. Transition function . . . 8
2.2.2. Reward function . . . 8
2.2.3. Discounted factor and discounted cumulative reward . . . 8
2.2.4. Policy . . . 8
2.2.5. Value and Q-value . . . 8
2.2.6. Episode and transition . . . 9
2.2.7. Bellman equation . . . 9
2.3. Tabular Q-learning . . . 9
2.3.1. The Algorithm . . . 10
2.3.2. The epsilon-greedy policy . . . 10
2.3.3. The exploration/exploitation dilemma . . . 11

3. Deep Learning and neural networks 13
3.1. Machine Learning . . . 13
3.2. Working Principle . . . 14
3.2.1. Single Neuron . . . 14
3.2.2. Neuron Layer . . . 14
3.2.3. Adding Non-linearity . . . 15

3.2.4. Deep Learning and Deep Neural Networks . . . 16
3.2.5. Supervised Learning . . . 17
3.3. Recurrent Neural Networks . . . 19
3.3.1. Classical Recurrent Neural Networks . . . 19
3.3.2. Long Short-Term Memory cells . . . 19
3.3.3. Gated Recurrent Units . . . 21
3.4. Deep Reinforcement Learning . . . 21
3.4.1. Limitations of tabular learning . . . 21
3.4.2. Deep Q-Learning . . . 22
3.4.3. Replay buffer . . . 22
3.4.4. Target network . . . 23

4. Multi-Agent Reinforcement Learning **25**
4.1. Independent Q-Learning . . . 25
4.2. Value-Decomposition Networks . . . 25
4.3. QMix . . . 26

5. A solution to variable-length observation spaces **29**
5.1. POMDPs and variable-length observation spaces . . . 29
5.2. RNN-based encoder . . . 30
5.2.1. Concept . . . 30
5.2.2. Training . . . 31
5.2.3. Encoder of other agents . . . 32

II. Environment **33**

6. *tanksEnv* **35**
6.1. Tanks . . . 35
6.2. Actions . . . 35
6.3. Observations . . . 35
6.3.1. Four parts observation . . . 35
6.3.2. Encoded observations . . . 36
6.3.3. Full state observations . . . 36
6.4. Reward . . . 37
6.5. Parameters . . . 37
6.6. Rendering . . . 37

III. Implementation and results **39**

7. Encoders training **41**
7.1. Parameters . . . 41
7.1.1. Enemy tanks encoder . . . 41
7.1.2. Allied tanks encoder . . . 42
7.1.3. Obstacles encoder . . . 42
7.1.4. Full state encoders . . . 42
7.2. Results . . . 43
7.3. Representation of the results . . . 43
7.4. Analysis of the results . . . 44
7.5. Threshold value . . . 48
7.6. Obstacles encoders . . . 52
7.7. Discussion . . . 52

8. Reinforcement learning **55**
8.1. Setup 1 . . . 55
8.1.1. Parameters . . . 55
8.1.2. Results . . . 56
8.1.3. Discussion after the first setup . . . 58
8.2. Setup 2: random IDs . . . 59
8.3. Setup 3: obstacles . . . 60
8.4. Setup 4: random obstacles . . . 62
8.5. Multi-Agent algorithms . . . 63
8.5.1. Parameters . . . 63
8.5.2. VDN . . . 64
8.5.3. QMix . . . 65
8.6. Discussion . . . 65

9. Conclusion and discussion **67**
9.1. Conclusion . . . 67
9.2. Belgian Defense . . . 67
9.3. Ideas for further improvements . . . 68

A. Supplementary Information **69**
A.1. Probability of kill as a function of range . . . 69

B. Computing time and hardware **71**
B.1. Comment on the computing times . . . 71
B.2. Hardware . . . 71
B.3. Profiling of the learning programs . . . 71
B.3.1. Simple DQN . . . 71
B.3.2. Encoder and RNN . . . 72
B.4. Improvements . . . 73
B.4.1. The *is_visible* method . . . 73
B.4.2. Supervised learning of the single-agent obstacles encoder . . . 74

C. Time series data smoothing **75**

D. Deep Q-Network algorithm **77**
D.1. Implementation . . . 77

E. Hypothesis tests **79**
E.1. Null hypothesis . . . 79
E.2. p-value . . . 79

F. RNN agents **81**
F.1. DQN with RNN . . . 81
F.1.1. Exploitation . . . 81
F.1.2. Learning . . . 81
F.1.3. Storing the transitions . . . 81

Bibliography **83**

List of Figures

2.1. Agent-environment interactions in an MDP [AA22a] . . . 7
2.2. Example of a Q-table in Q-learning [Bae21] . . . 10

3.1. Schematic views of a biological and an artificial neuron [Dis20] . . . 14
3.2. Schematic view of a two neurons layer with five inputs . . . 15
3.3. Widely used activation functions . . . 16
3.4. Schematic view of a deep neural network . . . 16
3.5. Stochastic Gradient Descent with and without momentum [Mus22] . . . 18
3.6. Recurrent Neural Network[AA22b] . . . 19
3.7. Long Short-Term Memory layer [Che18] . . . 20
3.8. Gated Recurrent Unit[Phi20] . . . 21

4.1. (a) Mixing network (red: hypernetworks, blue: mixing network layers). (b) Global QMix architecture. (c) Agent Q-Network. [RSS$^+$18] . . . 27
4.2. Win rates for IQL, VDN and QMix, learning to play the StarCraft II video game on six different maps. [RSS$^+$18] . . . 27

5.1. Different types of RNN architectures[Kar22] . . . 30
5.2. Architecture of the RNN encoder . . . 31
5.3. Training architecture of an RNN encoder . . . 32

6.1. P_k as a function of $\frac{R}{R50}$ . . . 36
6.2. Two rendering modes of a 20x20 grid with two blue agents and two red agents . . . 38

7.1. Enemy tanks encoder-decoder result for an input sequence of length 1 . . . 45
7.2. Enemy tanks encoder-decoder result for an input sequence of length 2 . . . 46
7.3. Enemy tanks encoder-decoder result for an input sequence of length 3 . . . 47
7.4. Enemy tanks encoder-decoder result for an input sequence of length 4 . . . 47
7.5. Result for an input sequence of length 1 with a threshold value of 4, ID=2 . . . 48
7.6. Result for an input sequence of length 2 with a threshold value of 4 . . . 49
7.7. Result for an input sequence of length 3 with a threshold value of 4 . . . 50
7.8. Result for an input sequence of length 4 with a threshold value of 4 . . . 51
7.9. Obstacles decoder output . . . 53

8.1. Setup 1, one red agent . . . 57
8.2. Setup 1, two red agents . . . 57
8.3. Setup 1, three red agents . . . 58
8.4. Sum of rewards per episode for setup 2 . . . 60
8.5. Examples of start states for setup 3 . . . 61
8.6. Sum of rewards per episode for setup 3 . . . 61
8.7. Sum of rewards per episode for setup 4 . . . 63
8.8. Results for VDN training . . . 64
8.9. Results for QMix training . . . 65

A.1. Standard deviation of the ballistic dispersion as a function of range . . . 70

C.1. Smoothing applied on a reward curve . . . 76

D.1. DQN Runner diagram 78

F.1. DRQN exploitation architecture 82
F.2. DRQN training architecture 82

List of Tables

6.1. Parameters of the *tanksEnv* environment 37

7.1. Enemy tanks encoder parameters 41
7.2. Enemy tanks encoder parameters 42
7.3. Enemy tanks encoder parameters 42
7.4. Accuracy and recall of the RNN encoders 43

8.1. Expected sum of rewards for setup 1, one red agent 57
8.2. Expected sum of rewards for setup 1, two red agents 57
8.3. Expected sum of rewards for setup 1, three red agents 58
8.4. Expected sum of rewards for setup 2 60
8.5. Expected sum of rewards for setup 3 62
8.6. Expected sum of rewards for setup 4 62

A.1. Standard deviation of the ballistic dispersion as a function of range 69

List of Abbreviations

α Learning rate

γ Discount Factor

π Policy

π^* Optimal Policy

G_t Cumulative Discounted Reward

$J(\boldsymbol{\theta})$ cost function

$Q_*(s, a)$ Q-value of state-action pair (s,a) under the optimal policy

$Q_\pi(s, a)$ Q-value of state-action pair (s,a) under policy π

$R(s, a, s')$ Reward function

$T(s, a, s')$ Transition function

$V_*(s)$ Value of state s under the optimal policy

$V_\pi(s)$ Value of state s under policy π

AI Artificial Intelligence

CLDE Centralised Learning/Decentralised Execution

CTDE Centralised Training/Decentralised Execution

DQN Deep Q-Network

DRL Deep Reinforcement Learning

DRQN Deep Recurrent Q-Network

EMA Exponential Moving Average

FC Fully Connected

GRU Gated Recurrent Unit

i.i.d. independent and identically distributed

IQL Independent Q-Learning

LSTM Long Short-Term Memory

MARL Multi-Agent Reinforcement Learning

MDP Markov Decision Process

ML Machine Learning

NN Neural Network

POMDP Partially Observable MDP

RL Reinforcement Learning

RNN Recurrent Neural Network

SGD Stochastic Gradient Descent

VDN Value-Decomposition Networks

1. Introduction

> Machine intelligence is the last invention that humans will ever have to make.
>
> Nick Bostrom

1.1. Background

This thesis has been written in the continuation of [BOE20], which has been written within the framework of the Intelligent Recognition Information System (IRIS) project. This project aims to develop a software that can assist the crews of deployed armored vehicles. It is separated into three central parts:

1. Detection and recognition of objects on the terrain based on on-board sensor data (image and IR).
2. Classify the recognized objects according to a certain threat level.
3. Develop a strategy to handle the different threats. This strategy can be proposed to the vehicle's crew as an action or series of actions to undertake.

In this thesis, we only focus on this third part. The battlefield in which the strategies are developed is a simulated environment. [BOE20] has thus two goals.

Firstly, develop a model of a battlefield that is simple enough to work with but complete enough to capture some essential elements from real environment, so that the developed strategies make sense and are useful in real life situation.

Secondly, develop strategies within the modeled battlefield with some multi-agent algorithms, using recent advances in the domain of deep learning and reinforcement learning.

In this thesis, those two principal objectives are kept, but the idea is to re-implement the already developed algorithms and environment, and then bring some contribution that could improve the quality of the work and the obtained results.

The major contribution of this thesis is the proposition of a new method to address a problem that can be encountered when working with deep reinforcement learning algorithms: variable-length observations. The idea is to encode this variable-length observation to a fix-length vector without (or as little as possible) loss of information.

The main matter of this thesis is separated into three parts:

1. The first part is rather theoretical. It describes the different notions and algorithms that are used in this thesis. The last chapter of this part describes the main problem addressed in this thesis, as well as the proposed solution: the variable-length observation encoder.
2. The second part describes the working and possibilities of the modeled battlefield: *tanksEnv*. It is largely inspired by the battlefield environment developed in [BOE20].
3. The last part shows and discusses the results obtained.

1.2. Literature Review

This section present some of the literature that has been useful in the writing of this thesis. For some of the cited papers, more detailed explanation will be given in the first chapters of this thesis in order to help better understand the different concepts that we used in this thesis.

1.2.1. (Deep) reinforcement learning

The topic of reinforcement learning really started in the 80's. The first tabular Q-learning method [Wat89] was proposed in the late 80's. It is only in the 2010's that some deep reinforcement learning algorithms (i.e reinforcement learning combined with deep learning) were implemented [MKS+13] [HS15a] [vHGS15].

Besides, some other interesting algorithms have been developed in the field of multi-agent reinforcement learning. Amongst them, two are of interest for us. VDN [SLG+17], that proposes a cooperative multi-agent reinforcement learning method with a single joint reward, where the agent still have their own individual policies. This was the first idea of a centralised learning/decentralised execution algorithm. A few years later, another paper proposed a similar cooperative multi-agent learning algorithm: QMix [RSS+18], which allows to learn more complex joint Q-value functions.

1.2.2. Variable-length observation MDPs and encoder

We didn't find scientific literature about solving the problem of Variable-length observation MDPs in deep reinforcement learning. The solution that we propose in this thesis was an in-house idea. However, we have been inspired by the concept of autoencoders [BKG20]. The idea of training an encoder with a decoder comes from there. In autoencoders, the aim is to compress some input data with an encoder. To train this, the decoder will try to retrieve the original input data by decompressing the output of the encoder. Both the encoder and the decoder will be training by trying to minimize the difference between the original input and the decompressed output. Once trained, the encoder can be used on its own to compress data without (or with low) data loss.

1.3. The code behind the words

The different algorithms presented in this thesis have all been (re)implemented. All the values and graphs presented in the last part of the thesis are the result of an important programming effort and numerous simulations, some of them being quite heavy computation-wise, particularly for a laptop. The code is mainly written in Python, with PyTorch used as deep learning library. As for the different plots and statistical analysis, the code is written in Matlab. All this code is to be found on the GitHub repository linked to this thesis [Rem22].

1.4. Notation

1.4.1. Code in text

To indicate that some words are code words, they will be written in italic. Those words can be functions, variable names, parameters, ... used in the implementations of the algorithms and/or the environment developed within the framework of this thesis. For instance, *tanksEnv*, *use_encoder* and *shoot* are all code words.

1.4.2. References to equations

In order to refer to an equation in the text, the reference of the equation is written between brackets. The word ”equation” is not explicitly mentioned. For instance, we’ll write:

> At each transition, the agent ”learns” by updating its Q-table using (2.11)

where 2.11 refers to equation 2.11.

1.4.3. Use of the plural form

Although I am the only author of this thesis, I deliberately chose to express myself in the plural. In french, this is called ”le nous de modestie” (the ”us” of modesty). It is a reminder that the author does not started from scratch, but rather builds on prior works and knowledge. This is a way to honour my teachers and other mentors, as well as the authors of the works on which I based myself, and which guided me through the writing of this thesis. This also softens the ego and the individualistic aspect of the ”I”.

Part I.

Models and algorithms

2. Reinforcement learning

This chapter briefly explains the working principles of reinforcement learning (RL). RL is an area of Machine Learning (ML), which is a part of Artificial Intelligence (AI).

2.1. Markov Decision Processes

In RL, an agent performs in and interacts with an environment. This environment is a Markov decision process (MDP)[1][2]. An MDP is defined by:

- A set of states
- A set of actions
- A transition function
- A reward function
- A start state
- Often one or several terminal (or end) state(s)

At every time step, the agent gets an observation of the current state of the environment. The agent then chooses an action among a set of actions authorized by the environment and submits it to the environment. This action is chosen according to the agent's policy (see section 2.2.4). Finally, the environment reacts to this action by updating its state and providing the agent with a reward that depends on the quality of the action chosen by the agent. This process then repeats itself until the environment ends up in an end state. An illustration of the agent-environment interactions in an MDP is shown on figure 2.1.

Figure 2.1. Agent-environment interactions in an MDP [AA22a]

[1] In this thesis, the terms "MDP" and "environment" are used interchangeably.

[2] In MARL (chapter 4) and in POMDPs (chap 5), the environment is not an MDP in the strict sense. The slight differences will be discussed later in this thesis.

2.2. Concepts and definitions

2.2.1. Transition function

The transition function $T(s, a, s')$ gives the probability that taking action a in state s will lead to state s', i.e.:

$$T(s, a, s') = P(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.1}$$

Note: The lowercase notation refers to generic values, whereas the uppercase notation refers to specific instances, sampled at specific time steps (indicated in subscript). For instance, S_t is the state observed at time step t.

2.2.2. Reward function

The reward function $R(s, a, s')$ gives the reward granted when the agent takes action a in state s and ends up in state s'. In some MDPs, the reward only depends on the state the agent ends up in and is thus noted $R(s)$.

2.2.3. Discounted factor and discounted cumulative reward

In RL, the goal of the agent is to find the sequence of actions that will maximize the cumulative reward, i.e. the sum of all rewards received. However, we often introduce a discount factor γ $(0 \leq \gamma \leq 1)$. We then talk about the cumulative discounted reward G_t, which is expressed as follows:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

$$= R_{t+1} + \gamma G_{t+1} \tag{2.3}$$

The agent must then find the sequence of actions that optimizes G_t. This factor models the uncertainty of the future rewards and helps our algorithm to converge.

2.2.4. Policy

The policy $\pi(s)$ is a function that returns the action a to take in state s. In a more general way, it can also return a probability distribution over all available actions in state s. If that is the case, it will be denoted $\pi(a|s)$. The optimal policy, i.e. the one that optimizes (2.2) is noted $\pi^*(s)$.

2.2.5. Value and Q-value

The value of a state $V_\pi(s)$ is the expected (discounted) cumulative reward when starting from state s and choosing action with policy π.

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.4}$$

$V_{\pi^*}(s)$ or $V_*(s)$ is the value of state s under the optimal policy.

The Q-value of a state-action pair $Q_\pi(s, a)$ is the expected (discounted) cumulative reward when starting from state s and taking action a, and then acting according to policy π.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{2.5}$$

In the same way as for the state value, $Q_{\pi^*}(s,a)$ or $Q_*(s,a)$ is the Q-value of state-action pair (s,a) under the optimal policy. Sometimes, a state-action pair is also called a Q-state.

The link between value and q-value is then:

$$V_\pi(s) = Q_\pi(s, \pi(s)) \tag{2.6}$$

Or, with the optimal policy:

$$V_*(s) = \max_a Q_*(s,a) \tag{2.7}$$

2.2.6. Episode and transition

An episode in an MDP is one specific sequence of states, actions and rewards that begins with the start state of the environment and ends with a terminal state.

In a MDP, moving from one state to another is called a transition. More specifically, when we talk about a transition in RL, it can be (and will be for the rest of this thesis) a sample of one time step of a particular episode. The transition at time t contains the following data:

- S_t the state (or local observation) at time step t.
- A_t the action performed by the agent at time step t.
- R_t the reward received by the agent at time step t.
- S_{t+1} or S'_t the state (or local observation) at time step t+1.
- *is_done* a Boolean value that is true only if S_t is terminal. If that is the case, S_{t+1} will then be *None* or 0.

2.2.7. Bellman equation

It can be shown that the Q-value function of the optimal policy $Q_*(s,a)$ obeys a recursive relationship that is called the Bellman equation:

$$Q_*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_*(s',a')] \tag{2.8}$$

$$= \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_*(s')] \tag{2.9}$$

The optimal policy can than easily be retrieved from $Q_*(s,a)$:

$$\pi_*(s) = \arg\max_a Q_*(s,a) \tag{2.10}$$

2.3. Tabular Q-learning

So, in RL, we want to find the optimal policy, that is the one that maximizes (2.2). One algorithm to do so is tabular Q-learning [Wat89]. As the name suggests, the algorithm will try to learn the optimal Q-values for every state and action in the MDP in which it is performing. The optimal policy may then be retrieved with (2.10).

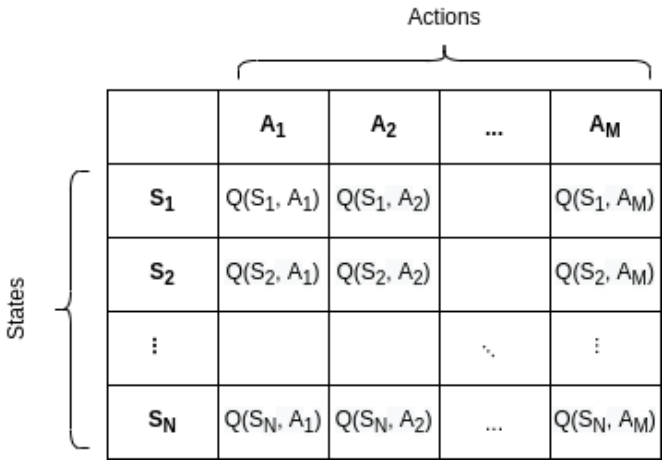


Figure 2.2. Example of a Q-table in Q-learning [Bae21]

2.3.1. The Algorithm

First, a table is created. This table contains the estimations of the Q-values for every possible state and action in the environment. It can be initiated with random or zero values. This table is called the Q-table. An example of such a table for an environment with N states and M actions is shown on figure 2.2.

Then, the agent will perform in the environment, collecting data in the form of transitions. At each transition, the agent "learns" by updating its Q-table using (2.11), inspired by (2.8).

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.11}$$

where:

- α $(0 < \alpha < 1)$ is the learning rate.
- $[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$ is called the temporal difference. It is actually the difference between what $Q(S_t, A_t)$ should be (i.e. $R_t + \gamma \max_a Q(S_{t+1}, a)$) and what it really is.

At every learning step, that is at every transition, $Q(S_t, A_t)$ is incremented by the temporal difference multiplied by the learning rate. If $\alpha = 1$, the agent ignores prior knowledge and $Q(S_t, A_t)$ is replaced by a new value at every learning step. If $\alpha = 0$, the agent is not learning at all. The learning rate actually determines how fast the Q-values in the Q-table will change. A higher learning rate might lead to faster convergence of our Q-values towards the optimal values. However, in probabilistic environment, a too high learning rate might also lead to no convergence at all, because taking too big steps may result in an oscillation of the estimated Q-values $Q(s, a)$ around the optimal value.

Note: If S_t is a terminal state, 2.11 becomes:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t - Q(S_t, A_t)] \tag{2.12}$$

2.3.2. The epsilon-greedy policy

As explained in previous section, the agent learns by interacting with its environment. While it does this, it updates its Q-table with the data obtained from the environment at every transition. Thus, the different states visited by the agent have a really important impact on the quality of the learning process. Those visited states depend on the actions the agent decides to perform during the training phase,

which are determined by its policy. Hence, besides the learning algorithm, the agent's policy is also really important in the learning process.

At first sight, it might seem a good option to use (2.10), but using the agent's Q-table in the equation instead of the optimal Q-values function $Q_*(s, a)$, which is unknown. However, following that sub-optimal policy might lead to an agent that explores only certain states. If the environment gives some high reward when going to certain states, but the agent never goes to those states, the Q-values for those states will never be learned.

Thus, the agent shall explore every state during the learning. A solution to that is the epsilon-greedy policy. At every transition, the probability that the agent will choose an action using (2.10) with its own Q-table is $1 - \epsilon$. And there is an ϵ probability that the agent will take a random action from the set of all available actions in the environment. It is obvious that ϵ is a really important parameter in the epsilon-greedy policy and that $0 \leq \epsilon \leq 1$.

The epsilon-greedy policy is shown in algorithm 1.

Algorithm 1 Epsilon-Greedy Policy

procedure EPS-GREEDY($S; Q(s, a), \epsilon$)
  $x \sim \mathcal{U}(0, 1)$ ▷ Generate a random number between 0 and 1 (uniform law)
  if $x < \epsilon$ **then**
    $A \longleftarrow$random action
  else
    $A \longleftarrow \arg\max_a Q(S, a)$
  end if
  return A
end procedure

2.3.3. The exploration/exploitation dilemma

The exploration/exploitation dilemma is a problem that is inherent to RL. As already stated in previous section, we have to add some randomness in the chosen actions in order to allow the agent to explore all of the possible states. That is what is called exploration.

However, in some more complex environments, there are states in which the agent is very unlikely to find itself by performing only random actions. Let's take the example of a video game where there are several levels: if the agent acts randomly, it could get some little rewards if it starts to progress in the correct direction. But, if the game is complex, it is very unlikely that the agent will finish the whole level and access the next ones by playing randomly. Hence, the agent will never observe the more advanced states, further in the game. So, at a certain moment, the agent has to exploit what it has already learned in order to access some states that are more difficult to reach, and explore the results of action undertaken in those states. This is called exploitation.

The dilemma is thus to find a good equilibrium between exploration and exploitation, in order to be sure to explore every possibility, but still advance to the states that are more difficult to access. One solution to this problem is to implement an epsilon decay. The idea is to decrease the value of the epsilon parameter of the agent's epsilon-greedy policy as the episodes progress. This way, the agent will act more randomly in the start of the training, when it hasn't learned much, but will act more and more according to its Q-table, as the values of that table converge to the optimal solution.

3. Deep Learning and neural networks

This chapter presents the basic concepts of deep learning and neural networks, as well as how they can be used in reinforcement learning .

3.1. Machine Learning

Machine Learning (ML) is a field of AI that studies algorithms that can establish models based on sample data, called training data, and can then make predictions on new data, similar, but not identical to the training data. Some Machine Learning algorithms use neural networks. For the rest of this thesis, ML only refers to those algorithms.

A neural network (or NN) is a function that takes a tensor (a multi-dimensional vector) as input and computes an output, which is also a tensor of values. This computation involves a lot of parameters that are used to compute the output from the input. It can be noted as follows:

$$\mathbf{Y} = NN(\mathbf{X}; \boldsymbol{\theta}) \tag{3.1}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are respectively the input and output tensors, NN the neural network function and $\boldsymbol{\theta}$ the neural network parameters.

The parameters of a neural network can be tuned in such a way that the neural network approximates any given function. This is called the learning. Those parameters are learned using some data composed of both some inputs and the outputs that we want our neural network to produce when fed with those inputs.

For instance, if we want our neural network to be able to make the distinction between an apple and a banana, we will give him some data that could be, for instance, photographs of apples and bananas (the inputs) as well as the corresponding labels for each photograph (the outputs). The network will then learn from this data by tuning its parameters. Then, the neural network should be able to classify some new images of apples and bananas that it has never been presented with during its training. In this example, the input tensors of the network are the values of all the pixels in the images, and the output will only have two values: the probability that the network thinks the input is a picture of an apple and the probability that it thinks it is a banana. This is called a classifier neural network. Other applications of NNs are:

- Image up-scaling (resolution increase)
- Text translation
- Image generation
- Approximation of the Q-values in function of an agent's observation (see section 3.4)
- ...

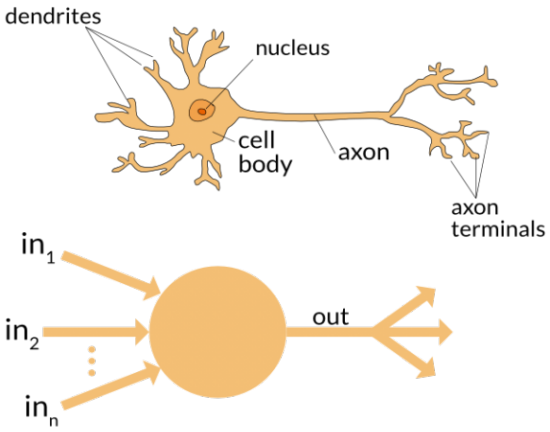


Figure 3.1. Schematic views of a biological and an artificial neuron [Dis20]

3.2. Working Principle

3.2.1. Single Neuron

The basic building block of an artificial neural network is an artificial neuron. The name comes from the real neurons in our brains, because the technology has been inspired by it. Figure 3.1 shows a schematic view of a biological and an artificial neuron. An artificial neuron is a mathematical function that takes several values as input an computes one output. The output is a weighted sum of the inputs to which is also added a fixed value, called the bias. For n input values, the output y is thus calculated as follows:

$$y = b + \sum_{i=1}^{n} w_i x_i \tag{3.2}$$

where x_i is the i-th input, w_i its corresponding weight and b the neuron's bias. This can also be written in matrix form:

$$y = b + \vec{w}^T \vec{x} \tag{3.3}$$

where $\vec{w}$ is a vector containing all the weights and $\vec{x}$ is a vector containing all the inputs.

3.2.2. Neuron Layer

Furthermore, to obtain more than one output, multiple neurons can be stacked and form what is called a neuron layer. Figure 3.2[1] shows a two-neurons layer fed with five inputs. The output of such a neurons layer may be calculated as follows:

$$\vec{y} = W\vec{x} + \vec{b} \tag{3.4}$$

This is almost the same as (3.3), but now we have $\vec{y}$ a vector containing multiple outputs, $\vec{b}$ a vector containing the biases of every neuron in the layer and W a two-dimensional matrix containing every weight of every neuron, that is all the horizontal vectors $\vec{w}^T$ from (3.3) of all the neurons, stacked on top of each other.

[1] Figure generated with NN-SVG [ale22]

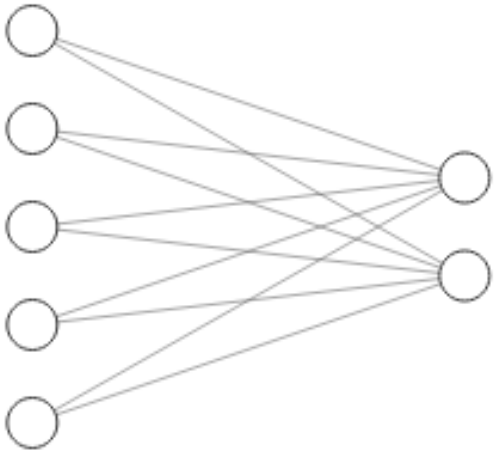


Figure 3.2. Schematic view of a two neurons layer with five inputs

3.2.3. Adding Non-linearity

Earlier in this chapter, we said that the aim of a neural network is to approximate a given function. However, (3.4) clearly shows that the relation between input and output is linear. Thus, only linear functions can be approximated, which is very limiting. To overcome this problem, some non-linearity is added into the neural network. This is done by applying a non-linear function to the output of each neuron. In the field of machine learning, those functions are called activation functions. Figure 3.3 shows four activation functions that are widely used in machine learning: sigmoid, Rectified Linear Unit (ReLU), hyperbolic tangent (tanh) and Exponential Linear Unit (ELU). Equations (3.5) to (3.8) show the formula of those activation functions. Introducing non-linear activation functions into the neural network allows to approximate a much wider set of functions.

$$\sigma(x) = sigmoid(x) = \frac{1}{1 - e^{-x}} \tag{3.5}$$

$$ReLU(x) = max(0, x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.6}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.7}$$

$$ELU(x) = \begin{cases} e^{x} - 1 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.8}$$

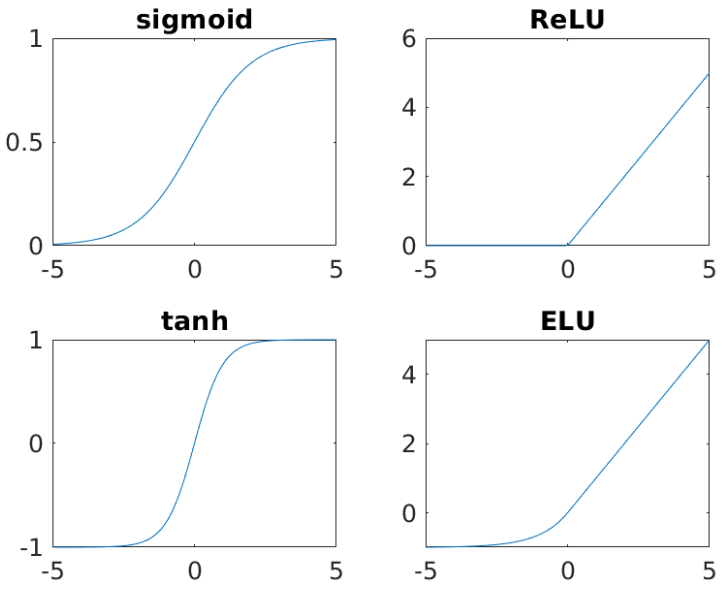


Figure 3.3. Widely used activation functions

3.2.4. Deep Learning and Deep Neural Networks

In order to approximate more complex functions, deep neural networks are often used. They consist of several neuron layers stacked horizontally, where the output of each layer is fed as input to the next layer. Of course, a non-linear function is applied at the output of every layer. Such a neural network, where every output of a layer is fed to every neuron in the next layer, is said to be fully connected (FC). Figure 3.4 shows a deep neural network with five inputs and two outputs. In this example, the data first goes through three layers of 10 neurons (called hidden layers). The output of the last hidden layer is then fed as input to the output layer. The activation functions are not shown on the figure, because they are considered to be part of the neurons, i.e. each neuron performs (3.3) then applies its activation function. Deep Learning (DL) is the part of machine learning that uses such deep neural networks.

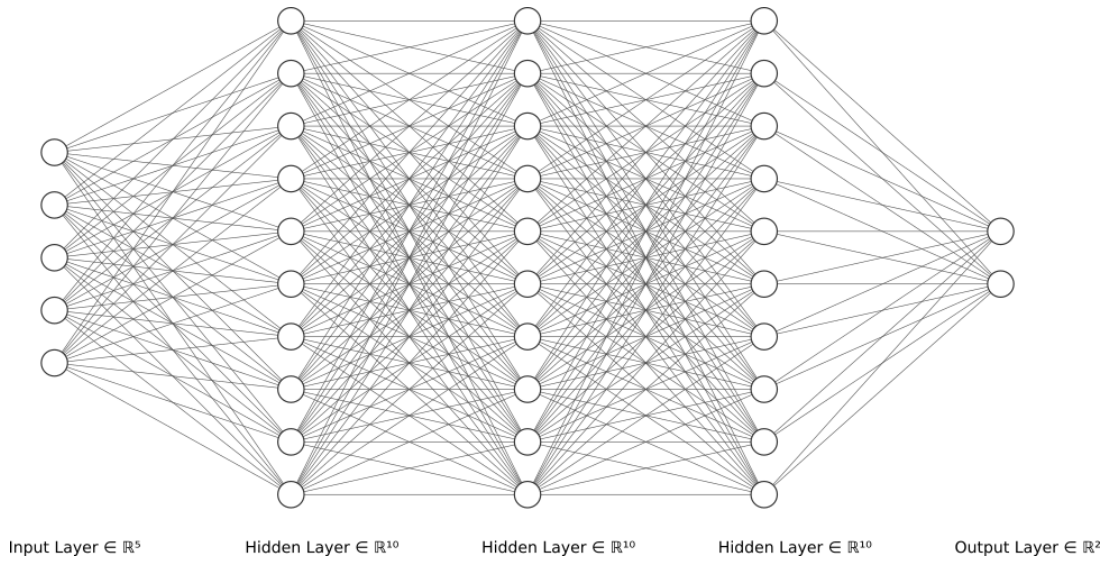


Figure 3.4. Schematic view of a deep neural network

3.2.5. Supervised Learning

In a neural network, the parameters ($\boldsymbol{\theta}$ in equation 3.1) are the weights and biases of all the neurons in the network. Learning the weights and biases using a database composed of both inputs and their corresponding outputs is called supervised learning. This is opposed to another field in machine learning called unsupervised learning, where the training data is only composed of inputs. The objective is then to detect patterns in the data and classify it into categories.

Loss function

In supervised learning, the parameters are optimized by minimizing a cost function $J(\boldsymbol{\theta})$:

$$J(\boldsymbol{\theta}) = \mathcal{L}(\hat{Y}, Y) \tag{3.9}$$

where $\hat{Y} = NN(X;\boldsymbol{\theta})$ and $L(\hat{Y}, Y)$ is called the loss function and quantifies the difference between the expected output from the learning data Y and the output produced by the neural network $\hat{Y}$. $\hat{Y}$ is often called the prediction and Y the target.

A lot of different loss functions are used in Deep Learning, depending on the application. (3.10) shows the Binary Cross Entropy (BCE) loss, which is used when the outputs of the networks should be either 0 or 1, in classifier networks for instance. (3.11) shows the Mean Square Error (MSE) loss.

$$\mathcal{L}_{BCE}(\hat{Y}, Y) = -\frac{1}{N}\sum_{i=1}^{N} Y_i\, ln(\hat{Y}_i) + (1 - Y_i) ln(1 - \hat{Y}_i) \tag{3.10}$$

$$\mathcal{L}_{MSE}(\hat{Y}, Y) = \frac{1}{N}\sum_{i=1}^{N} \left(\hat{Y}_i - Y_i\right)^2 \tag{3.11}$$

This cost is rather easy to compute for given parameters and training data. However, it might be really heavy computation-wise for big neural networks and/or big data sets. Fortunately, this can be parallelized a lot and the computation time can be reduced very significantly by making this computation on GPUs (Graphics Processing Units), which have a very large number of cores (up to tens of thousands) that can work in parallel, whereas classical CPUs (Central Processing Units) have at most a couple of tens of cores.

Gradient descent

The way the cost function is minimized is by applying a gradient descent algorithm, which is an iterative method. $\boldsymbol{\theta}_t$ will thus denote the value of $\boldsymbol{\theta}$ at iteration t.

First, the gradient of J with respect to $\boldsymbol{\theta}$ (i.e. $\nabla_{\boldsymbol{\theta}} J$) is calculated. This is most often done using backpropagation [RHW86]. The explanation of its working is out of the scope of this thesis.

Then, a small step in the negative direction of the gradient is taken:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} J \tag{3.12}$$

where α is called the learning rate. The discussion about this learning rate in RL (section 2.3) is also valid here.

Those two steps are repeated until some stop criterion is reached (number of iterations, threshold value for the loss, accuracy of the predictions, ...).

Furthermore, the gradient is almost never calculated based on one single sample. Instead, the average over several (or all of the) samples from the training data is computed. Those samples selected from the training data put together are called a batch. The batch size is a very important hyperparameter when training neural networks. A higher batch size often results in faster convergence and better results, at the cost of higher computation times. However, beyond a certain point, increasing the batch size even more won't hardly increase the performance, but will still increase the computation time.

Using only a few samples from the training data is called stochastic gradient descent (SGD) [RM51] [KW52]. The computation time for one step is smaller than for bigger batches, but the direction of the step is less optimal. Hence, with SGD smaller, uncertain steps are taken instead of bigger, but more certain ones. The advantage of it resides in the reduced computation time to generate and train on one batch. However, more batches will be required.

Such an algorithm that calculates the size and direction of the steps to take at every iteration is called an optimizer.

Momentum

Other more complex optimizers than simple gradient descent exist. Momentum [Qia99] is an upgrade of classical gradient descent. In order to increase the efficiency, a momentum term is added. The momentum term takes into account the direction of the previous steps to determine the size and direction of the current step. It works as follows:

$$m_{t+1} = \alpha \nabla_{\boldsymbol{\theta}} J + \mu m_t \tag{3.13}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - m_{t+1} \tag{3.14}$$

where μ is a parameter that determines how much the previous steps influence the current one.

An illustration of momentum applied to a two parameters optimization problem is shown on figure 3.5. The image can be visualized as a plane where the position (in x,y-coordinates) represents the value of the two parameters. The black elliptical curves are iso-loss curves and the red dot is the optimal value of the two parameters for a minimal loss. The saw tooth shaped black arrow represents the different values of the two parameters, that are updated at every time step.

What is illustrated on the figure, is that SGD is less efficient when some parameters have to "move" less than others to reach the optimum for the same loss value. In this case it is represented by the fact that, on the image, for the same loss value, the distance to reach the optimal value are longer along the x-axis than along the y-axis. Momentum will help to work around this problematic. Another advantage of momentum is that, in certain cases, it allows to get out of a local minimum, instead of staying stuck in it.

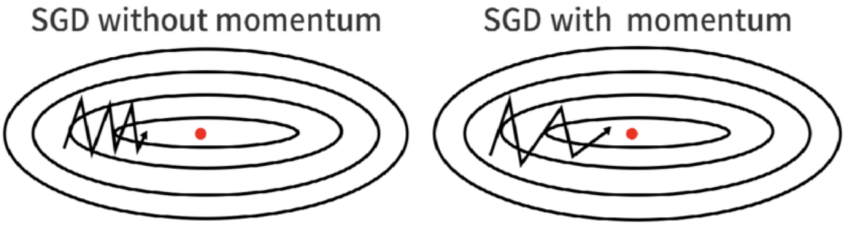


Figure 3.5. Stochastic Gradient Descent with and without momentum [Mus22]

Other optimizers

Even more sophisticated optimizers have been developed in the field of Deep Learning. The goal of every optimizer is to achieve faster convergence and/or to convergence towards better results (lower loss values). Adam [KB14] is maybe the most widely used optimizer in Deep Learning. It estimates the first and second order momenta and computes several adaptive learning rates. The explanation of its working is out of the scope of this thesis.

[Rud16] presents an overview of the most popular optimizers used in Deep Learning. In Deep Reinforcement Learning, the optimizer will rarely influence the final result (i.e. the obtained policy) significantly.

3.3. Recurrent Neural Networks

3.3.1. Classical Recurrent Neural Networks

Besides the classical fully connected neural networks, there exist a lot of other types of neurons that offer some advantages, depending on the application. Recurrent Neural Networks (RNN) are used to process sequences of data, i.e. sequences of inputs of the same dimensions. In most of the cases, the sequence is a time series.

In a FC NN, if two inputs are fed one after the other to the network, both are processed independently. Whereas in RNN, some information from the (processing of the) first input of the sequence is kept in the network and used to process the next input. The information that is kept is actually the hidden states, i.e. the values of the outputs of the neurons in the hidden layers of the neural network. For the first input of the sequence, this hidden state must be initialized to some value. The most common practice is to initialize it to zero or random values.

Figure 3.6 shows an illustration of a RNN. x, a and y are respectively the inputs, hidden states and outputs. Here, the superscript notation expresses the position in the sequence, i.e. the time step.

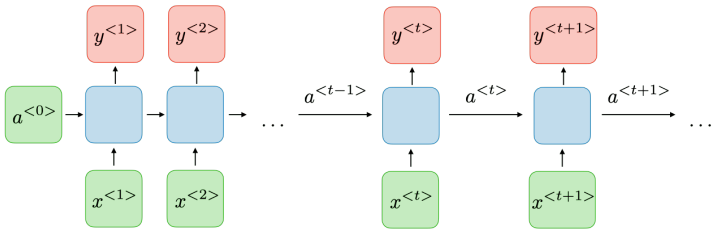


Figure 3.6. Recurrent Neural Network[AA22b]

3.3.2. Long Short-Term Memory cells

Classical RNNs allow to insert some kind of memory in the network. The network "remembers" information from the previous inputs it has received. There is, however, a limitation to this: if the sequence is too long, it is difficult for the network to carry information from early time steps to late ones. Thus, RNNs have memory, but it is short-term. Long Short-Term Memory (LSTM) cells [HS97] are a solution to this.

In LSTMs, not only is the hidden state passed to the next time step, but also what is called the cell state. This cell state is a kind of memory of the neuron that can hold information during a lot of time steps. The working of a LSTM cell is a bit more complicated than that of a simple recurrent neuron. Actually,

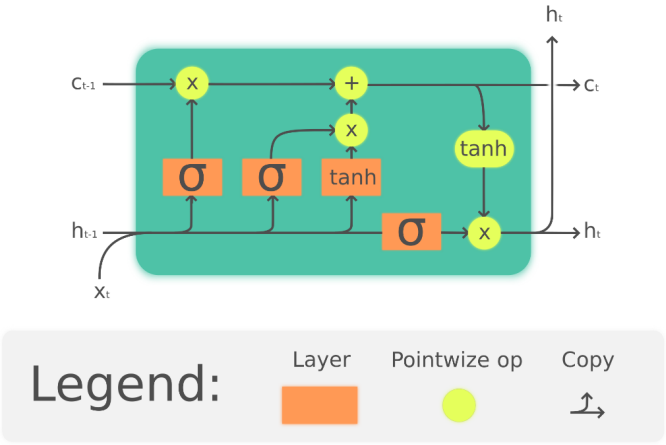


Figure 3.7. Long Short-Term Memory layer [Che18]

an LSTM cell is composed of four classical neurons, similar to those described in section 3.2.1. Those neurons are called gates.

The first gate is the forget gate. Its activation function is a sigmoid, so that its output value is always between 0 and 1. This gate determines whether or not the cell state of the previous time step will be passed on to the next step. A forget value of 1 means the previous cell state is kept, while a forget value of 0 means the previous cell state is forgotten.

The second gate is the cell gate or candidate. This gate has a tanh activation function, which constraints its output value between -1 and 1. This gate computes a candidate value to replace the cell state of the previous time step.

The third gate is called the input gate, it has a sigmoid activation function. It determines if the candidate will actually be added to the current cell state or discarded. It works the same way as the forget gate.

The three previous gates determine the value of the cell state:

$$\vec{c_t} = \vec{f_t} \odot \vec{c}_{t-1} + \vec{i_t} \odot \vec{g_t} \tag{3.15}$$

where $\odot$ is the symbol for the Hadamard product (element-wise multiplication), $\vec{c_t}$ is the cell state at time step t and $\vec{f_t}$, $\vec{i_t}$ and $\vec{g_t}$ are respectively the outputs of the forget, input and cell gates at time step t.

The last gate is called the output gate, and also has a sigmoid activation function. It computes what information should be in the hidden state, in the same way as the forget and input gates. The next hidden state is then computed as follows:

$$h_t = \vec{o_t} \odot tanh(\vec{c_t}) \tag{3.16}$$

where h_t is the hidden state at time t and $\vec{o_t}$ is the output of the output gate at time step t.

A visualisation of an LSTM cell is shown on figure 3.7. Note that the input for all of the gates is the input of the LSTM cell concatenated with the hidden state of the previous time step.

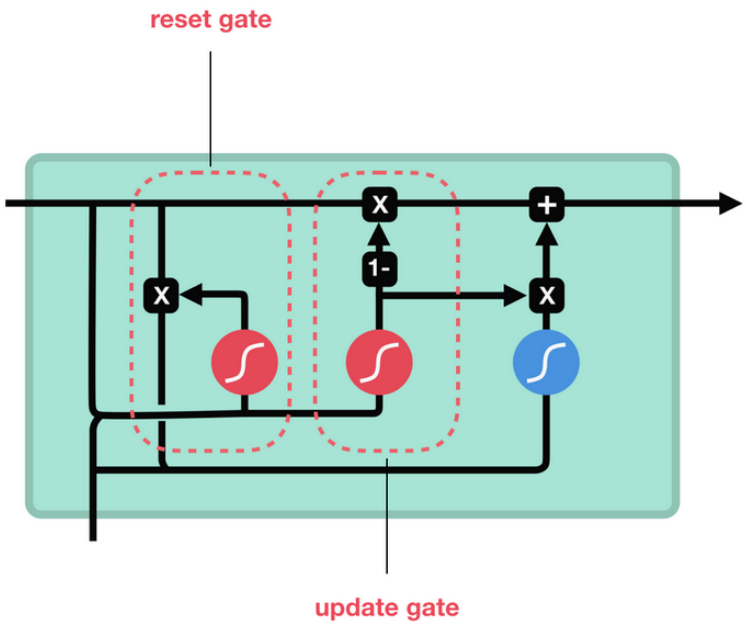


Figure 3.8. Gated Recurrent Unit[Phi20]

3.3.3. Gated Recurrent Units

Gated Recurrent Units (GRUs) [CGCB14] have been developed more recently than LSTMs. They are solution that solves the short-term aspect of the memory in classical RNN cells. GRUs are pretty similar to LSTMs, but a lighter version, hence faster. There is no cell state in GRUs, so only the hidden state is passed on to the next time step. They also only have three gates: a reset gate, an update gate and a candidate.

The reset gate determines how much of the information from the previous time step is used to calculate the new candidate for the next hidden state. The update gate acts like the input and forget gates of an LSTM. It determines to what extent the previous hidden state and the new candidate contribute to the next hidden state. The candidate gives a candidate value to replace the current hidden state.

An illustration of a GRU cell is shown on figure 3.8. The red circles represent neuron layers with sigmoid activation functions, whereas the blue circle (the candidate) represents a neuron layer with a tanh activation function.

GRUs are a bit faster than LSTMs[Muc21], but don't have a real memory (the cell state in LSTMs). None of them has a clear advantage over the other. The best solution is always to try both and determine which one works the best. For this thesis, we tried both of them, and there was no clear difference in the quality of the results. Because the GRU cells tend to train a bit faster and we did not observe any other difference for our application, we decided to use GRUs whenever we needed to use RNNs.

3.4. Deep Reinforcement Learning

3.4.1. Limitations of tabular learning

As seen in sec 2.3, a Markov Decision Process can be optimized using Q-learning, by creating a table that has as many rows as there are different possible states (i.e. the size of the state space) in the MDP, and as many columns as there are possible actions (i.e. the size of the action space). Even though this algorithm might be efficient with simple MDPs, when the complexity increases, it is limited by some

important factors.

First, in some MDPs, the state space (and action space) is so large that the Q-table would be really to big. Let's take the example of a game where the agent's observations are the values of the pixels of the screen of the game. For every pixel, the red, blue and green colours are coded with one byte each, which means there are over 16.5 millions different colours (255^3). Multiplying that by the amount of pixels in an image would yield to at least several tens of billions of different possible states, which means the same amount of rows in the Q-table, which is not feasible.

Second, there are a lot of different possible states that are very similar, like two adjacent positions or a small difference of colour of a few pixels between two observations. Hence, they should have similar (or identical) Q-values. In tabular learning, the agent must observe all of those different states several times in order to approximate correctly the corresponding Q-values. Furthermore, there could also be a lot of possible state that the agent will actually never observe. Taking the same example of a game where the observations are screenshots, not every possible image (every combination of colours) will be displayed, and thus observed.

3.4.2. Deep Q-Learning

A solution to this is to use a neural network to approximate the Q-values. The input of the network is then the observation of the agent and the outputs are the estimated Q-values for every possible action in the environment. This solution works really well with a lot of complex MDPs. It has been used to train an agent that can play better than humans at several Atari® games [MKS+13].

The idea in deep Q-learning, also called Deep Q-Network (DQN), is to let an agent evolve in an MDP and update the parameters of the Q-Network at every transition. Each transition contains the observed state (S), the action taken (A), the reward received (R) and the state observed after the action has been undertaken (the "next observed state", S_{next}).

At each time step t, the loss is computed using some loss function. The first argument of this loss function is the current approximated Q-value, that is the output of the Q-Network when fed with the observed state, for the undertaken action: $Q(S_t, A_t)$. The second argument is the target value:

$$R_t + \gamma \max_a Q(S_{next}, a) \tag{3.17}$$

where R_t is the received reward at time step t, $Q(s)$ is the output vector of the Q-Network for observation s, i.e. the approximation of all Q-values, and $Q(s, a)$ is the output of the Q-Network for a specific action a.

The Q-Network is then trained using stochastic gradient descent or another optimizer with the values of the computed loss.

3.4.3. Replay buffer

The main idea in DQN is to use supervised learning to approximate $Q(s, a)$ a complex and nonlinear function. Given that the parameters of the Q-Network are updated at every transition, and therefore that there is only one sample per batch, SGD is used. However, one important requirement of SGD is that the training data is independent and identically distributed (i.i.d). As the different data samples are consecutive transitions from the same episode, they are very similar, and the training data is therefore not i.i.d at all.

One solution to deal with this is to store the transitions in a large buffer, called replay buffer or experience buffer. Then, the training batches are sampled from this buffer to train the Q-Network. The

size of the replay buffer is an important parameter. On the one hand, if it is too small the data is not i.i.d. On the other hand, if it is too large there are transitions in it that have been sampled from really "old" episodes, where the Q-Network was a lot less accurate and the corresponding policy less efficient. Those transitions are less interesting to learn from.

3.4.4. Target network

Another issue of DQN is that the Q-Network is evaluated in S and in S_{next}. Unfortunately, there is only one step between those two states, which means they are very similar. A neural network will have a hard time to tell the difference between those states. Thus, when the network will be updated for S, the values for nearby states (including S_{next}) will also change. Hence, by improving the estimation for S, the estimation for S_{next} might get worse.

To fix this stability issue, a copy of the Q-Network is made. This copy is called the target network. This network is used to compute the target value (3.17). The trick is to only update the parameters of the Q-Network, and not of the target network. We then copy the parameters of the Q-Network to the target network given period, expressed in number of episodes.

4. Multi-Agent Reinforcement Learning

Up to now, there was only one agent performing in the environment. However, we also want to develop multi-agent strategies in our environment.

This chapter describes some Multi-Agent Reinforcement Learning (MARL) algorithms that are used to train multiple agents in Multi-Agent environments.

4.1. Independent Q-Learning

One of the simplest MARL algorithms is Independent Q-Learning (IQL) [Tan93]. This simple method consists of implementing the basic DQN algorithm for every agent. Then, all agents perform simultaneously in the environment and they all train using the observations and rewards that they get individually from the environment.

Although IQL works well for a lot of environments, faster convergence and/or better performance of the agents can be obtained with more advanced methods that combine the information of the different agents during training. This is called centralised learning. However, for the execution, each agent has still its own Q-Network that approximates the Q-values for its own observed states. This is called decentralised execution. The next two sections present two Centralised Training/Decentralised Execution (CTDE) MARL algorithms.

4.2. Value-Decomposition Networks

Value-Decomposition Networks (VDN) [SLG+17] is a CTDE MARL algorithm. Instead of using individual Q-values, a joint Q-value function is learned: $Q_{tot}(\mathbf{s}, \mathbf{a})$, where $\mathbf{s}$ is the joint observation of the state and $\mathbf{a}$ is the joint action vector, containing the actions of every agent. In VDN, the joint Q_{tot} is simply the sum of all independent Q-values[1], that is:

$$Q_{tot}(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta}) = \sum_{a=1}^{n} Q_a(s_a, a_a; \theta_a) \tag{4.1}$$

where n is the number of agents, Q_a, S_a and a_a respectively the individual Q-Network, observation and actions of agent a and θ_a the parameters (i.e. the weights and biases) of Q_a. The loss function calculated on a batch sampled from the replay buffer is:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{b} \left(y_i^{tot} - Q_{tot}(\mathbf{S}_i, \mathbf{A_i}; \boldsymbol{\theta}) \right)^2 \tag{4.2}$$

where b is the batch size, $\boldsymbol{\theta}$ contains all parameters of all Q-Networks, the subscript i refers to the specific values of the i-th transition of the batch and

$$y_i^{tot} = \mathbf{R}_i + \gamma \max_{\mathbf{a}} Q_{tot}(\mathbf{S}_{next,i}, \mathbf{a}; \boldsymbol{\theta}^-) \tag{4.3}$$

[1] In this case, the term "Q-value" refers to the output of the individual Q-Networks of the agents. However, since only the Q_{tot} is learned, those values are not necessarily the proper Q-values for the agents, but rather utility functions.

where $\boldsymbol{\theta}^-$ are the parameters of the target Q-Networks, as explained in section 3.4.4.

The reward $\mathbf{R}_i$ is also a joint reward. This reward might be directly given by the environment or be the sum or mean of the rewards that the agents receive individually from the environment.

Even though the Q_a functions do not estimate the expected return, performing greedy action selection with the outputs of the Q_a-Networks for each agent will give an individual policy that can be used for decentralised execution and that is consistent with the centralised Q_{tot} function that has been learned.

4.3. QMix

QMix [RSS+18] is another approach that is also a CTDE learning algorithm. In this type of algorithms, the difficulty is to derive the decentralised agents' policies that are fully consistent with the centralised policy that has been learned. In VDN, this is guaranteed by the fact that Q_{tot} is the sum of the Qa values of every agents. Nevertheless, this condition is not necessary to assure the consistency between the centralised and decentralised policies. Instead, we could only ensure that a global argmax applied on Q_{tot} yields the same result as a serie of argmax operations applied on the individual Qa values:

$$\arg\max_{\mathbf{a}} Q_{tot}(\mathbf{s}, \mathbf{a}) = \begin{pmatrix} \arg\max_{a_1} Q_1(s_1, a_1) \\ \vdots \\ \arg\max_{a_n} Q_n(s_n, a_n) \end{pmatrix} \tag{4.4}$$

If (4.4) is satisfied, choosing greedy actions from Q_{tot} or individually from the Q_a functions would yield the same policy. This is already the case for VDN.

However, the idea of QMix is that the relation between Q_{tot} and Q_a can be extended to a larger set of functions, that is all the monotonic functions. This can be guaranteed by enforcing the following constraint:

$$\frac{\partial Q_{tot}}{\partial Q_a} \geq 0, \forall a \in [1, 2, ..., n] \tag{4.5}$$

Imposing (4.5) is a sufficient condition to ensure (4.4). To do this, QMix uses one Q-Network per agent to compute the Q_a values, a mixer network that computes Q_{tot} with the outputs of the Q-Networks and a bunch of neural networks that compute the weights and biases of the mixer network. Such networks are called hypernetworks [HDL16]. The trick to ensure (4.5) is to limit the weights to positive values. This is done by taking the absolute value of the outputs of the hypernetworks that compute the weights of the mixing network.

Figure 4.1 shows the overall architecture of QMix, as well as the mixer and agent networks architectures. The figure comes from [RSS+18], where the notation is different. However, Q_a and Q_{tot} refer to the same values as explained in this thesis. It can be observed that the Q_a networks have only access to o_t^a the local observation of agent a at time step t, while the hypernetworks of the mixing network have access to S_t the full state of the environment at time step t. Moreover, the exact architecture of the Q-networks (Q_a) might be adapted to fit the best the specific application. The use of a GRU layer (or any other RNN layer) is not strictly necessary.

By implementing a more complex mixer network, QMix is able to learn more complex joint Q-value functions, performs better and learns faster (in less episodes) than IQL or VDN, in a lot of different environments. QMix has been implemented and compared with VDN and IQL in [RSS+18]. Figure 4.2 shows some of the obtained results.

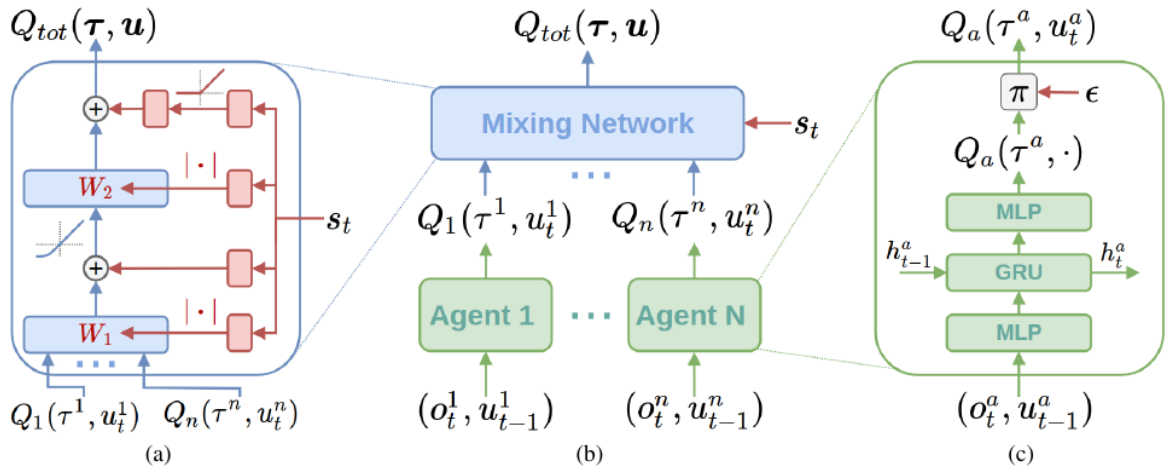


Figure 4.1. (a) Mixing network (red: hypernetworks, blue: mixing network layers). (b) Global QMix architecture. (c) Agent Q-Network. [RSS+18]

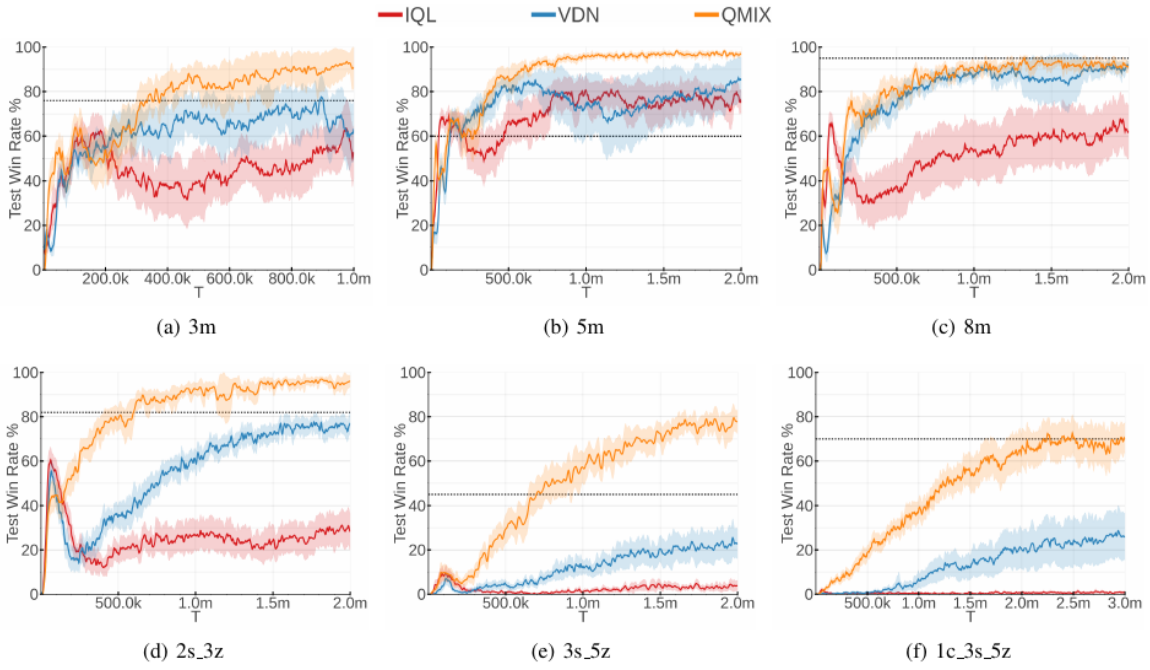


Figure 4.2. Win rates for IQL, VDN and QMix, learning to play the StarCraft II video game on six different maps. [RSS+18]

5. A solution to variable-length observation spaces

This chapter introduces the concept of partially observable MDPs (POMDPs) and a problem that arises in some environments of this type when working with deep reinforcement learning. Finally, we'll describe an RNN-based encoder that offers a solution to this problem.

5.1. POMDPs and variable-length observation spaces

In real life environments, it is often the case that an agent won't be able to observe the full state of the environment. Sometimes, it is even impossible to determine it. In such environments, the Markov property, which states that the state at time step t only depends on the state at time step $t-1$ (and not the full history of states), doesn't hold anymore. An example of this is an environment where some objects move at a certain speed, but the observation only contains the positions of those objects.

This problematic has already been talked about a lot, and multiple solutions have already been proposed to solve this problem. One of them is Deep Recurrent Q-Learning (DRQN) [HS15b]. The idea is to add RNN layers in the agent's Q-Network. The agent has then a memory and will remember information from earlier states.

However, there is another problem that can be encountered in POMDPs: variable-length observations. There could be some states where the agent can't observe some elements of the environment, whereas it can in other states. For instance, in the *tanksEnv* environment (described in chap 6), each tank has a maximum visible range beyond which it is unable to see the other agents or obstacles. Furthermore, the agents are unable to see through the obstacles. It is then clear that the amount of agents and obstacles an agent is able to observe depends on the state of the environment. Moreover, each visible obstacle and each visible agent will be described by a series of values (position, ID, ...) that has a fixed length, and the final observation is a concatenation of those fixed-length vectors describing each visible agent and obstacle. Hence, the observations an agent gets at different transitions might be of different lengths.

Moreover, some classical MDPs could also have variable-length observations. For instance, if the agent fights against multiple adversaries, when one of them dies it cannot be observed anymore. Nevertheless, this can be solved by leaving those adversaries in the observation and putting some Boolean value in the observation vector that specifies whether the observed agent is still "alive" or not.

This variable-length observation space property is an issue if we want to implement a DRL algorithm in an environment that has this characteristic. The problem is that, in DRL, the Q-value of a state is approximated with a neural network. However, classical NNs only accept inputs of the same shape. Thus, we can neither use DQN nor the methods described in chapter 4 when working with such environments.

5.2. RNN-based encoder

5.2.1. Concept

One solution to work around the variable-length observations problem and still use DRL can be to fix the NN's input length at a sufficiently high value. When the observation length is smaller than the NN's input size, the excess entries are put to values that can't be encountered in the observation. Or else, some supplementary inputs are added. Those inputs contain values that specify which portion of the NN's input is useful information and which portion doesn't contain data. Those supplementary inputs could then be seen as flags.

Although it is functional[1], this solution has some limitations. If it is not possible to determine the maximum size an observation can be, then the agent could get an observation that has a larger size that the maximum and that doesn't fit into the Q-Network. Or else, this maximum size could be overvalued, and be much larger that the actual maximum size of the biggest observation. This would work, but would also slow down the training phase, as the Q-Network would be larger.

In this thesis, we propose another method, based on RNNs, that solves this issue. As already explain in section 3.3, RNNs are used to treat sequences of data. The output sequence is then of the same length as the number of elements in the input sequence. But there are other types of RNN architectures. Some of them are shown on figure 5.1.

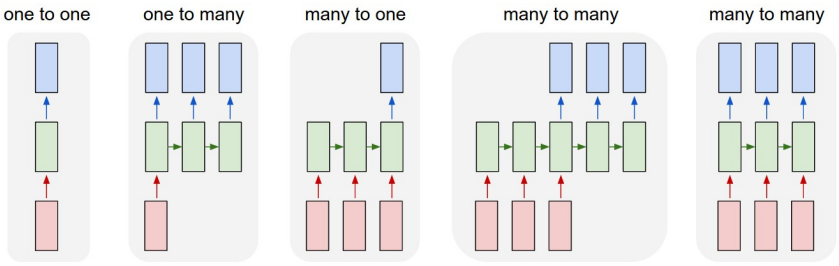


Figure 5.1. Different types of RNN architectures[Kar22]

The idea is to encode the observation data with an encoder that has a many-to-one RNN architecture. In a many-to-one RNN, only the last element of the output sequence is used, but, given the recursive nature of the neural network, the encoder can put information about every element of the input sequence in this last element of the output sequence.

In addition to the main RNN, we can put a classical FC NN that will process every element of the input sequence before they are fed to the RNN. We will call it the pre-processing network. Similarly, We can also put a FC NN that will process the output of the many-to-one RNN. We will call it the post-processing network. Those networks are optional but, if such NNs are used, they are considered to be part of the encoder.

The architecture of the full encoder is shown on figure 5.2. the X_i are the elements of the variable-length input sequence X, and Y is the fixed length output.

In the case of the *tanksEnv* environment, there will be three encoders: One for the allied agents (if in MARL mode), one for the enemy tanks and one for the obstacles. For instance, the elements of the input sequence of the obstacles encoder will be all the coordinates of all obstacles that are visible by

[1]Implemented in part III. Implementation and results.

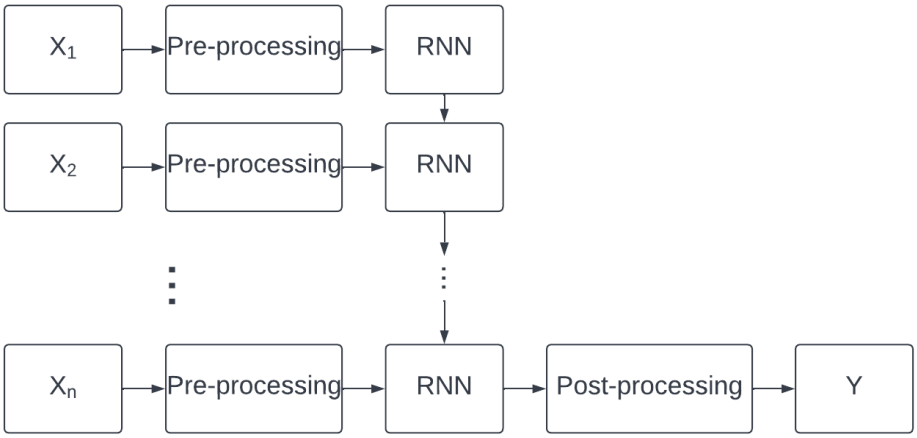


Figure 5.2. Architecture of the RNN encoder

the agent that is observing the environment. So, in the environment, the encoding part will be added as a post-processing overlay applied to the observations, before they are given to the agents.

5.2.2. Training

The encoders will be trained with supervised learning. To explain, we'll take the example of the obstacles encoder, where each element of the input sequence (or observation sequence) is a vector containing two values: the x,y-coordinates of an obstacle. The whole sequence is thus composed of all the obstacles visible by an agent. The only thing that will change for the two other encoders is the information that is contained in every element of the sequence, and thus the length of those elements.

At each learning step, i.e. each step of the optimizer, a batch is generated. Each element of a batch is composed of:

- An observation sequence (a sequence of random coordinates representing the visible obstacles) of random length (between 0 and a maximum length).
- A positive sample: One element of the observation sequence.
- A negative sample: One element that could be in the observation sequence, but is not. This means a coordinate that is visible by the agent, but where there is no obstacle.

The challenge here is to train the encoder to put all the information from the input sequence (or observation sequence) into the fixed-length output. To do this, we'll add a decoder at the output of the encoder. This decoder is fed with the output of the encoder, concatenated with some test coordinates (or test sample). The decoder is a FC NN that only has one output whose value is constrained between zero and one by a sigmoid activation function. An illustration of this training architecture is shown on figure 5.3.

The loss is then calculated with this single output with a BCE loss function. The target value for the loss computation is 1 if the test sample is in the input sequence (that has been fed into the encoder) and 0 otherwise. In each element of a training batch, there is one observation sequence and one sample with a target value of one and another with a target value of zero.

Given that the decoder has no access to the observation, but only to the output vector of the encoder, if it is able to tell if the test coordinates are in the input sequence or not, it means that the information

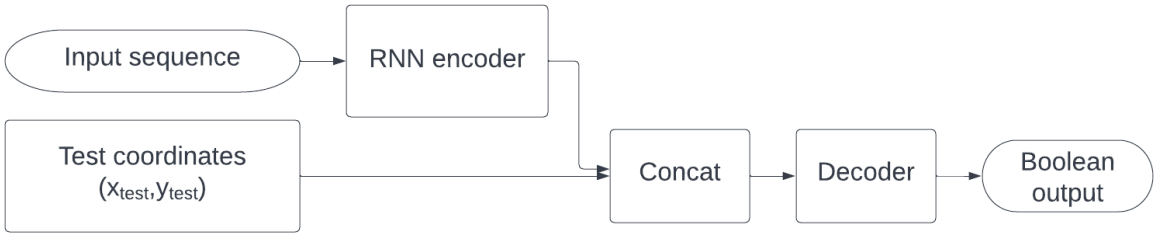


Figure 5.3. Training architecture of an RNN encoder

contained in the observation sequence is still present in the fixed-length output of the encoder.

Finally, there still remains a small issue. In order to give an output, the encoder has to be fed with at least one input, i.e. it cannot be fed with an empty sequence. That is a problem, since the observation sequence could be empty, that is if there are no visible obstacles. As the position of the obstacles are given in relative coordinates, centred on the observing agent and as the agent can't be at the same coordinates as an obstacle, there will never be the (0,0) coordinates in the observation sequence. Thus, when the observation sequence is empty, the input sequence for the encoder is composed of one single element : (0,0).

5.2.3. Encoder of other agents

In previous section, we took the example of the obstacles encoder to explain the learning process. However, we also need to train the two other encoders (for the allied and enemy tanks). The difference between those encoders and that of the obstacles is the information contained in every element of the observation sequence. For the enemy tanks, the observed tanks' IDs are also put in the observation sequence. Hence, this information should also be encoded in the fixed-length output of the encoder.

Nevertheless, the negative samples in the training batches are generated randomly (random position and ID), but with the random relative positions kept below a maximum distance, that is the visibility range of the agent. Besides, for a visibility range of 20[2], there are more than 1200 possible positions where an agent can be observed. Hence, there is a very small probability that the negative sample has the same position as an element of the observation sequence. During the learning phase, the decoder could thus learn to make the difference between positive and negative samples only based on the position and it would work almost every time. Therefore, it could be that the information of the ID of the observed tanks is lost and not hidden in the fixed-length observation vector.

To overcome this potential issue, half of the negative samples are generated with the same position as one of the observed tanks (i.e. one of the elements in the observation sequence), but with another ID. This will force the encoder and the decoder to make the difference between them based on the IDs.

Furthermore, for the encoder of the friendly tanks, there could be even more information added to the observation sequence (for example, the ammunition). However, the principle of forcing the encoder and the decoder to distinguish between positive and negative samples based on only one part of the observation (ex: ID, ammo) by putting the rest of the observation (ex: the position) to the same value as one element of the observation sequence, can still be applied the same way it has been explained for the enemy tanks encoder.

[2]Value used for training (see implementation in chapter 7)

Part II.

Environment

6. *tanksEnv*

This chapter presents *tanksEnv*, a tanks battle ground environment. It is based on the environment developed in [BOE20]. *tanksEnv* is a grid-like environment of finite size. Two teams are playing against each other: team ”red” and team ”blue”.

6.1. Tanks

Each team is composed of one or several players (or agents, or tanks) that are defined by their state, which contains the same information for each tank, that is:

- The position of the tank in (x,y) coordinates
- A positive or zero ID that is different for all tanks (of both teams)
- Its team
- How much ammo it has left
- Which other tank it's aiming at
- Whether it is still in the game or has been hit and can no longer move or shoot

6.2. Actions

The different available actions are:

- *nothing*: Do nothing.
- *north*, *south*, *east*, *west*: Move in one of the four cardinal directions, if there is nothing preventing the tank from doing so (obstacle, other tank, edge of the grid).
- *aim-i*: Aim at tank i (tank which ID is i), if this tank is visible by the tank that tries to execute the action.
- *shoot*: Shoot at the tank previously aimed at, if it is visible and action *aim-i* has already been carried out.

When taking action *shoot*, there is a certain probability to kill the target, that depends on the range to the target. This is shown on figure 6.1. Explanations about how it was determined are to be found in appendix A.1.

6.3. Observations

6.3.1. Four parts observation

When the *use_encoder* parameter is set to *False*, the observing agent gets an observation that is divided into four parts:

1. Information about the observing agent

The first part the agent gets is its own absolute position in the grid, in x,y-coordinates. To this is added the ID of the tank the agent is aiming at. This value is -1 if the agent is not aiming.

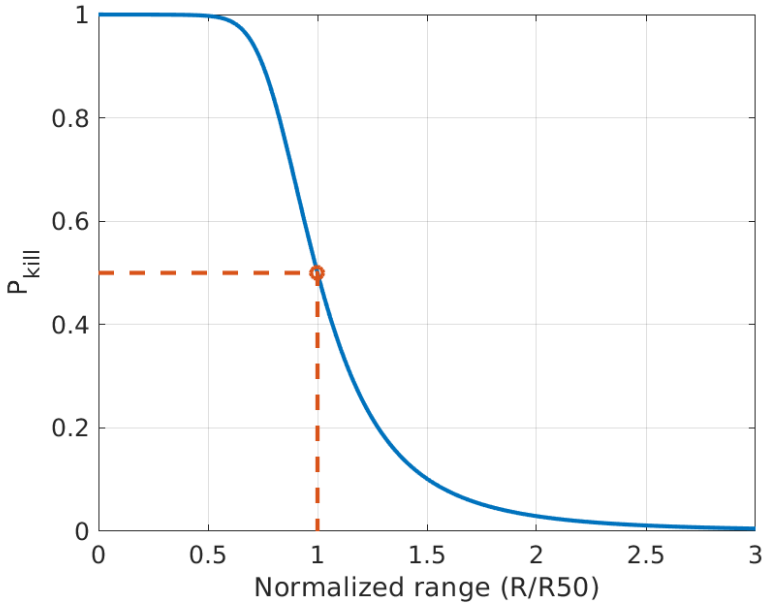


Figure 6.1. P_k as a function of $\frac{R}{R50}$

2. Information about the enemy tanks

The second part is a sequence of vectors of three values. Each vector contains the relative position of an enemy tank, centered on the observing agent, as well as the ID of the enemy tank. There is one element per visible[1] enemy tank.

3. Information about the friendly tanks

The third part is only present in MARL, that is when the agent is not the only one on its team. The information is almost the same as the sequence of the enemy tanks, but with two differences. First, the remaining ammo is added to each element of the observation sequence. Second, it is done with every visible friendly tank, and not with the enemy tanks.

4. Information about the obstacles

The last part of the observation is a sequence composed of all the relative positions of the visible obstacles.

6.3.2. Encoded observations

When the *use_encoder* parameter is set to *True*, three encoders are used: one for each of the three last parts of the observation. Those three observation sequences are then encoded to fixed length vectors. Finally the three encoded sequences, as well as the first part of the observation are concatenated together and returned to the agent as one observation.

6.3.3. Full state observations

tanksEnv also allows to observe the full state of the environment. This observation is then the same as the three last parts of a single agent observation, but for all tanks and obstacles on the grid and in absolute coordinates. In the same way as for the agent observation, it is possible to get the raw observation or the encoded observation, depending on the value of the *use_encoder* parameter.

[1] when not specified, "visible" is used in the sense of "visible by the observing agent"

6.4. Reward

The states where all the agents of one team have been killed are considered final states. In a final state, the reward accorded to the winning team (i.e. the team that has killed all tanks of the opposing team) is 1 and the reward accorded to the losing team is -1. This reward is awarded to every agent in the team, even those who have been killed before the end of the episode. In all other states, the reward for every agent is -0.01. This is done to penalize the agent that win in more time steps. The optimal policy should thus be to kill all the opponents as fast as possible.

6.5. Parameters

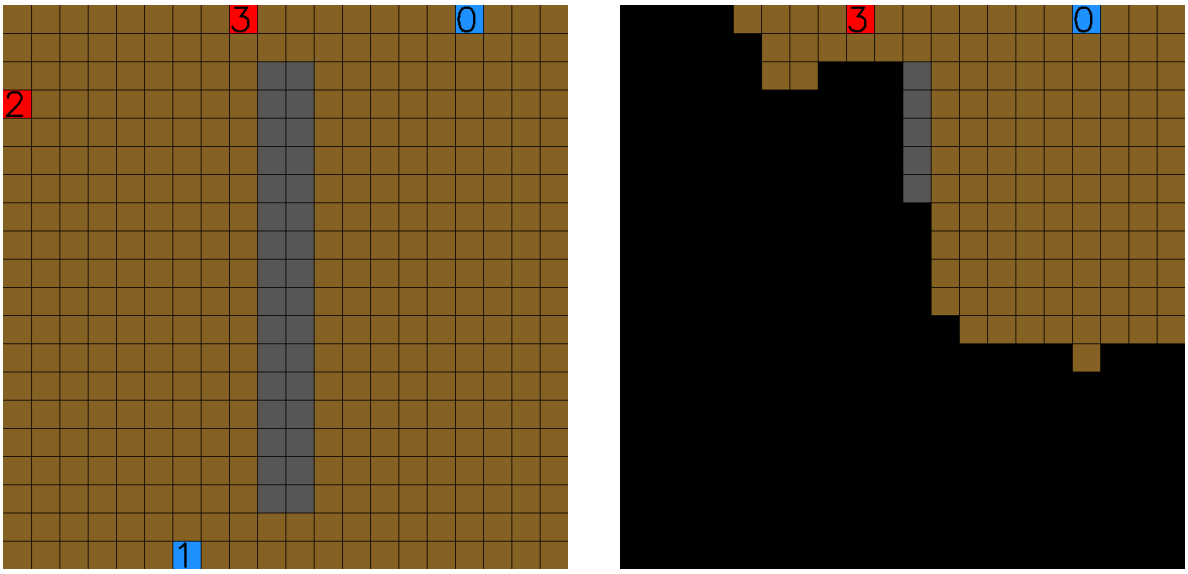
The environment has a lot of parameters that allow to modify its characteristics and working. Those parameters, description and default values are to be found in table 6.1. When specifying the parameters for the environment further on in this thesis, the parameters whose value are not explicitly mentioned are set to their default value.
Note: One cell = the distance between two adjacent cells.

Parameter	Description	Default value
size	Size of the grid	5×5
agents_description	Start position and team of each tank	One tank per team Random start position
visibility	Maximum observable range (euclidean distance)	4 cells
R50	Range at which $P_h = 50\%$	3 cells
obstacles	List of coordinates of all obstacles in the grid	No obstacles
borders_in_obstacles	Boolean: edges of the grid added to obstacles list	False
max_ammo	Maximum number of shots per tank (-1 = no limit)	-1
im_size	Number of pixels in height of the render image	480
max_cycles	Maximum number of transitions before the episode is automatically over (-1 = no limit)	-1
use_encoder	Boolean: Enable encoder processing of observations	True
random_ids	Boolean: Assign new IDs to the tanks at each new episode	False

Table 6.1. Parameters of the *tanksEnv* environment

6.6. Rendering

tanksEnv offers two rendering modes: one rendering of the full grid (figure 6.2a) and one ”Point of view” (POV) rendering, where the image is limited to what one agent can see (figure 6.2b). Those two rendering modes correspond to the single agent and full state observations, described in section 6.3.



(a) Full rendering

(b) POV rendering of agent 0 (visibility = 12 cells)

Figure 6.2. Two rendering modes of a 20x20 grid with two blue agents and two red agents

Part III.

Implementation and results

7. Encoders training

This chapter shows the results of the supervised training of the different encoders, as it is described in chapter 5.

7.1. Parameters

When working with deep learning and reinforcement learning, the performance and training time are influenced a great deal by a lot of parameters: learning rate, batch size, epsilon (decay), type of optimizer, topology of the neural network(s) (type, size and number of layers),... This section describes the main parameters, and the values used for the training of the different encoders and their corresponding decoders.

7.1.1. Enemy tanks encoder

The parameters and their values used in our implementation of the enemy tanks encoder are shown in table 7.1. When generating the training data, the IDs of the observed tanks are random integer between 0 and *max_id*. The sequence lengths of the elements in the batches are generated the same way, between 0 and the maximum sequence length. This also holds for the other encoders.

Parameter	Value
Pre-processing network	Two FC layers of 16 and 64 neurons
RNN	One GRU layer of 64 neurons (many-to-one architecture)
Post-processing network	One FC layer of 12 neurons
Decoder NN	Two FC layers of respectively 40 neurons and 1 neuron
Visible range	20 cells
Max. sequence length	4 enemy tanks
max_id	10
Optimizer	Adam
Learning rate	0.003
Batch size	256
Loss function	BCE loss
Learning steps	500 000

Table 7.1. Enemy tanks encoder parameters

7.1.2. Allied tanks encoder

The parameters and their values used in our implementation of the allied tanks encoder are shown in table 7.2. Just like the IDs of the tanks, the ammo of the observed allies are generated randomly between 0 and *max_ammo*.

Parameter	Value
Pre-processing network	Two FC layers of 16 and 64 neurons
RNN	One GRU layer of 64 neurons (many-to-one architecture)
Post-processing network	One FC layer of 15 neurons
Decoder NN	Two FC layers of respectively 64 neurons and 1 neuron
Visible range	20 cells
Max. sequence length	4 allied tanks
max_id	10
max_ammo	10
Optimizer	Adam
Learning rate	0.003
Batch size	256
Loss function	BCE loss
Learning step	500 000

Table 7.2. Enemy tanks encoder parameters

7.1.3. Obstacles encoder

The parameters and their values used in the implementation of the obstacles encoder are shown in table 7.3.

Parameter	Value
Pre-processing network	Two FC layers of 64 and 256 neurons
RNN	Two GRU layers of 256 neurons (many-to-one architecture)
Post-processing network	One FC layer of 64 neurons
Decoder NN	Three FC layers of respectively 64, 64 and 1 neurons
Visible range	20 cells
Max. sequence length	20 obstacles
Optimizer	Adam
Learning rate	0.003
Batch size	256
Loss function	BCE loss
Learning step	500 000

Table 7.3. Enemy tanks encoder parameters

7.1.4. Full state encoders

As explained in chapter 6, the environment also allows to observe the full state of the environment. In this full-state observation, the observation sequences are the same as for the single-agent observation, but the coordinates are absolute, instead of being centered on the observing agent. This is why three other encoders, similar to the ”single-agent” ones will be trained.

Those three full-state encoders have all the same parameters as their single-agent counterpart, but the data is generated differently. For the single-agent encoders, the positions are limited to a visible range.

For the full-state encoders, the x,y values of the different positions will be generated between 0 and a maximum value, i.e. a maximum size for the environment. This maximum size for the randomly generated training data is set to 40 for all full-state encoders.

7.2. Results

To assess the quality of the encoders, we'll use two different metrics: the accuracy and the recall. A prediction can be classified in one of the four following categories:

- True positive (TP): The target of the prediction (i.e. the correct value, used for the training) is positive and the prediction is positive.
- True negative (TN): The target and the prediction are negative.
- False positive (FP): The target is negative, but the prediction is positive.
- False negative (FN): The target is positive, but the prediction is negative.

The accuracy is the percentage of true predictions over a batch, that is:

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{7.1}$$

where TP, TN, FP and FN refer to the amount of predictions of the four categories in a batch.

The recall is the percentage of true positives over the positive items of a batch, that is:

$$\frac{TP}{TP + FN} \tag{7.2}$$

Table 7.4 shows the accuracy and recall of the six encoders after the training. Except for the obstacles encoders, those results are rather good. Using DQN to train agents in the *tanksEnv* environment in single-agent mode shows that the agents that are provided with encoded observation learn at least as good as the agents that receive the non-encoded observation (see chapter 8). The learning time for all of the encoders is about 4 hours.
Note: The processing times indicated in this thesis must not be considered as precise values, but rather as an indication or an order of magnitude. See appendix B for more explanation.

Encoder	Accuracy	Recall
Foes single-agent	98.8%	99.2%
Foes full-state	98.8%	99.2%
Friends single-agent	98.8%	99.2%
Friends full-state	99.0%	99.4%
Obstacles single-agent	88.2%	85.0%
Obstacles full-state	88.0%	86.0%

Table 7.4. Accuracy and recall of the RNN encoders

7.3. Representation of the results

Since the accuracy and recall values are not 100%, we tried to represent them visually. Since the results are very similar for all the other tanks encoders, we will only do this once, for the single-agent enemy tanks encoder.

To do this, we fed the encoder with a random sequence, generated the same way as during the training. We then gave as input to the decoder network all the possible observations[1], i.e all the possible (x,y), (x,y,ID) or (x,y,ID,ammo) tuples (depending on the encoder), that can be observed, called test tuples. For the single-agent enemy tanks encoder, this means all possible (x,y,ID) tuples with an ID between 0 and *max_id* and for which the norm of the x,y-coordinates is less or equal than *visibility*.

For all test tuples, the decoder outputs a value between 0 and 1. If this value is less than 0.5, we'll consider that the decoder prediction is that the test tuple is not present in the observed sequence that has been fed to the encoder. Otherwise, for a value greater than 0.5, we'll consider that the decoder predicts that the test tuple is in the observation sequence. We represented the result graphically.

The results of this for the single-agent enemy tanks encoder are shown on figures 7.1 to 7.4. On the different graphs, each pixel represents a coordinate, whose color is light if there is effectively a tank or obstacle there, that is if this coordinate is in the observation sequence. Furthermore, the color is green if the prediction is correct and red if it is not. So, TPs are light green, TNs are dark green, FPs are dark red and FNs are light red. In addition, for the enemy and allied tanks encoders, there is one image of this kind per ID or per ID/ammo combination.

Figure 7.1b has been made with the single-agent enemy tanks encoder-decoder network. The observation sequence only contains one tank at position (15, 12) and with ID=8. So, its corresponding tuple is (15, 12, 8). The figure shows the output of the decoder for every tuple where the ID is 8. The other subfigures in figure 7.1 show the same result, but for tuple where the ID is 7 (7.1a), 9 (7.1c) or 10 (7.1d). For this particular example, all the figures with the remaining IDs are not shown because they are exactly like figures 7.1a and 7.1d. There is no enemy tank and the decoder predicted it accurately for every coordinate.

7.4. Analysis of the results

Although it is only one example for one input sequence, the graphs of figure 7.1 give a good idea of the general behavior of the encoder-decoder architecture for an input sequence of one observed tank. We can see that the decoder gives positive predictions for the positions around the observed tank. This is not a problem, because those positions are really near the observed tank. Also, the agents in the *tanksEnv* environment aim by choosing an ID, not coordinates. So, the important information for the agent is the ID of the observed tank(s) and their range (because it affects the hit probability).

However, there are tuples where the ID is 9, and for which the decoder gives a positive prediction. Thus, it is possible that an agent, fed with such a fix-length observation encoded by a encoder, won't be able to determine the IDs of the observed tanks. If the agent can't know the observed tanks ID, it might be less efficient when training on the environment. The results with other sequence of length 1 are very similar: some positive predictions on and around the observed tank, and sometimes some positive predictions for an index 1 higher or lower than that of the observed tank (ex: on figure 7.1, there are positive predictions around the tank which ID is 8, as well as some other positive predictions on the $ID = 9$ map).

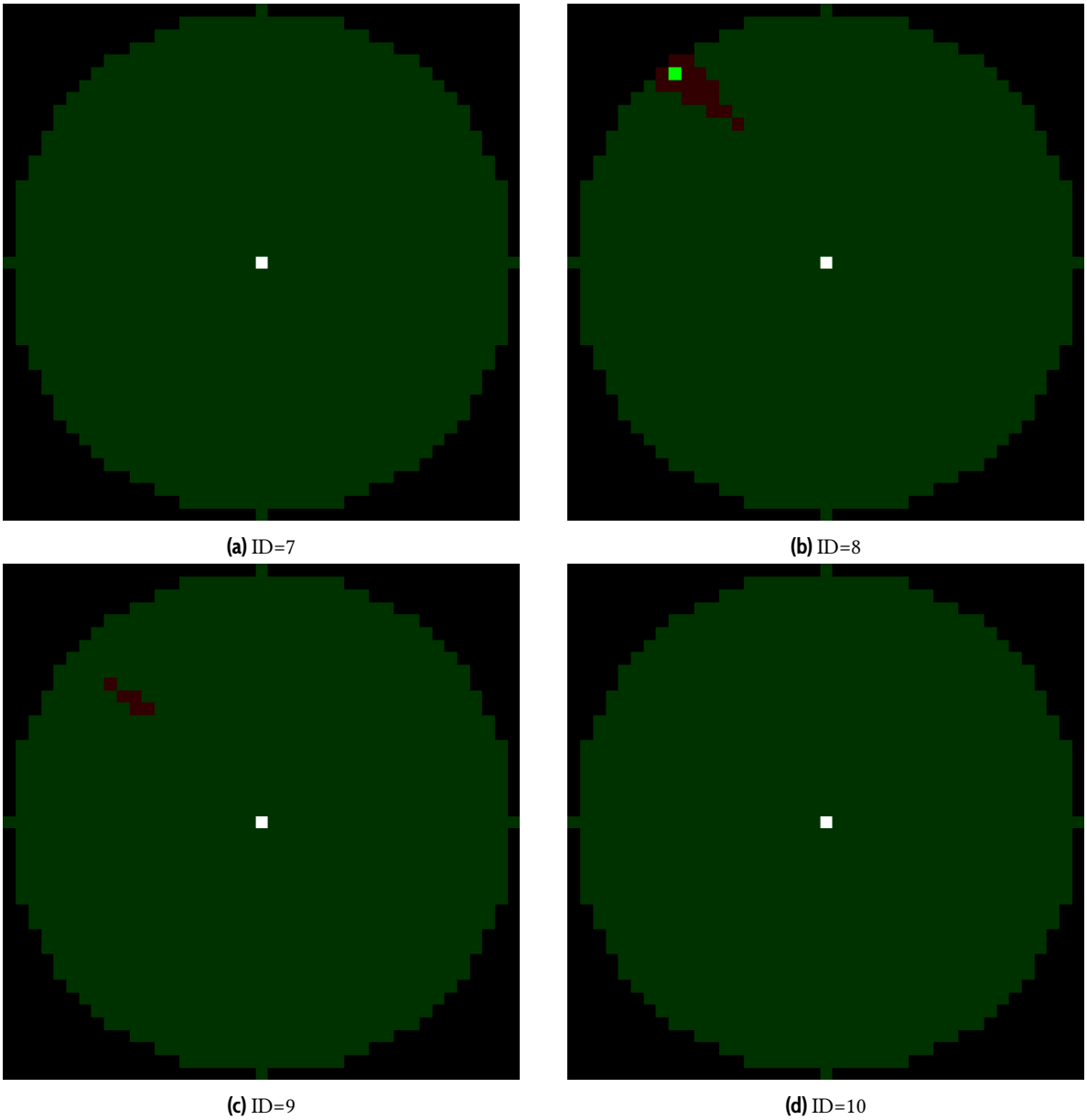
Furthermore, when the observation sequence length increases, the results get worse. For a length of 2, the decoder starts to make positive predictions for several IDs that are not those of the observed tanks (see figure 7.2).

For observation sequences containing three or four tanks, there are positive predictions for almost every ID, and the "spots" of false positives (dark red) become much larger. Some examples are shown on

[1]Of course, the decoder is also fed with the encoded sequence, i.e. the encoder output (see chapter 5)

figures 7.3 and 7.4.

When the sequence is empty, the encoder doesn't make any positive prediction. The accuracy and recall are thus both 100%.



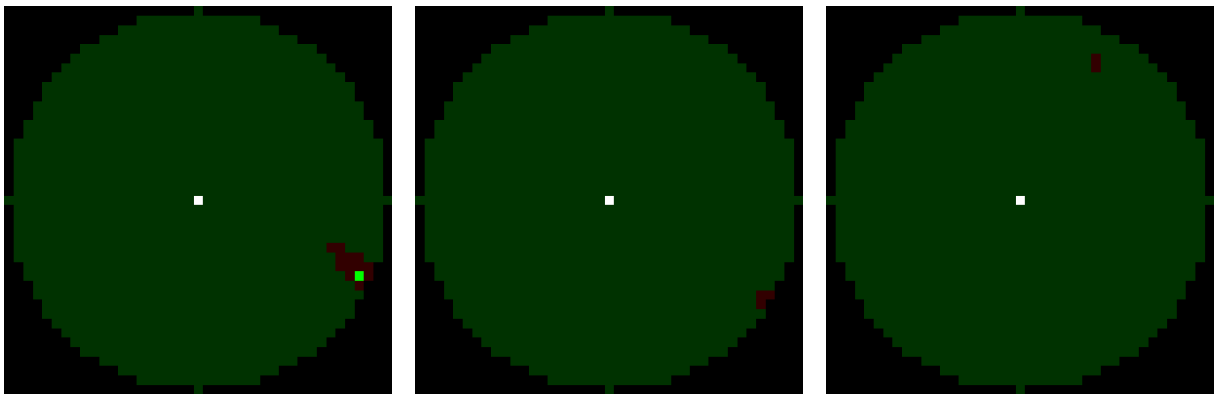
(a) ID=7

(b) ID=8

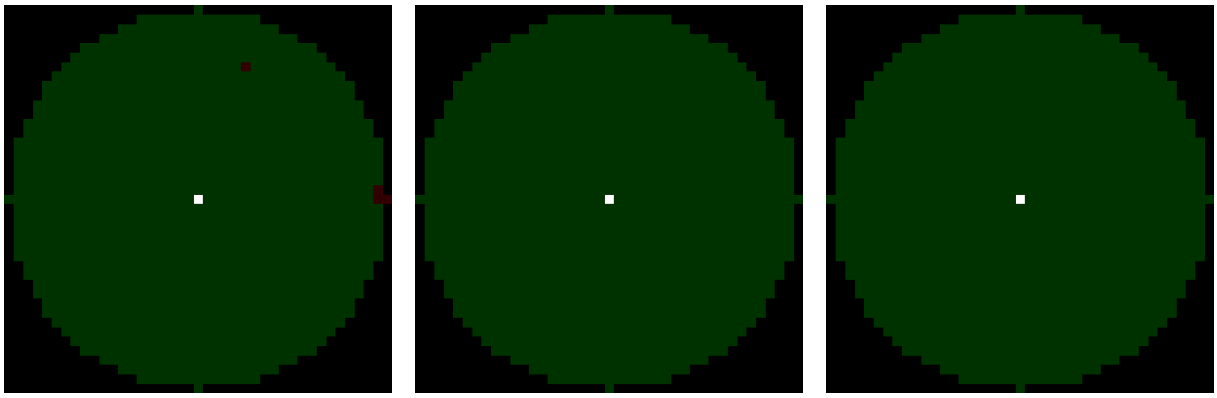
(c) ID=9

(d) ID=10

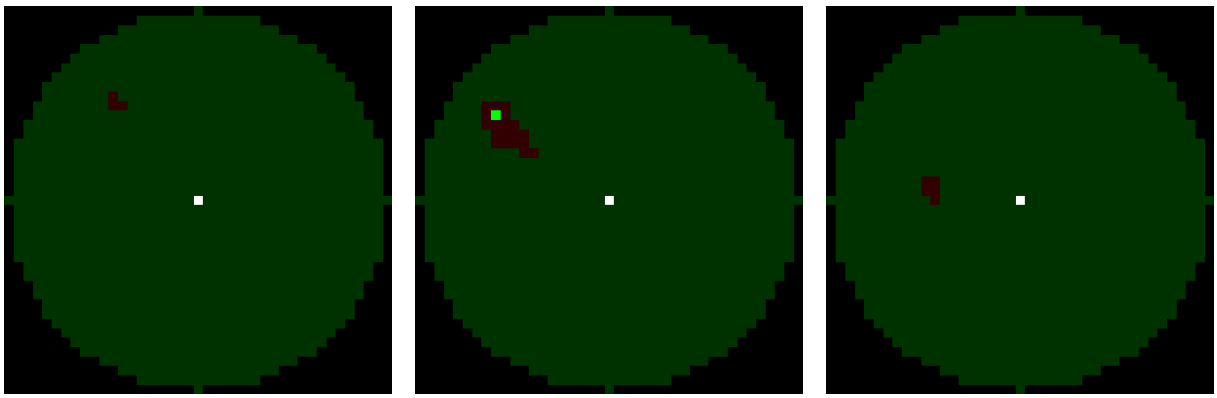
Figure 7.1. Enemy tanks encoder-decoder result for an input sequence of length 1



(a) ID=0 **(b)** ID=1 **(c)** ID=2

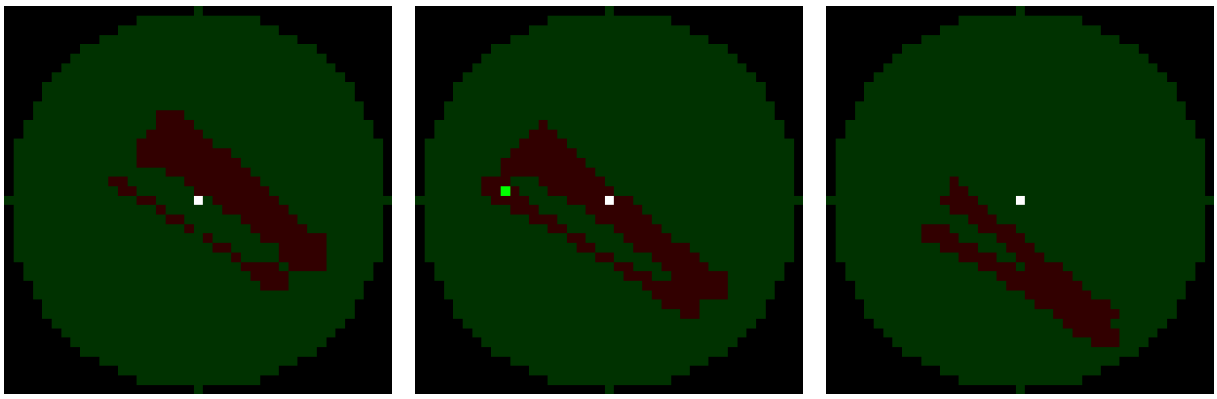


(d) ID=3 **(e)** ID=4 **(f)** ID=5



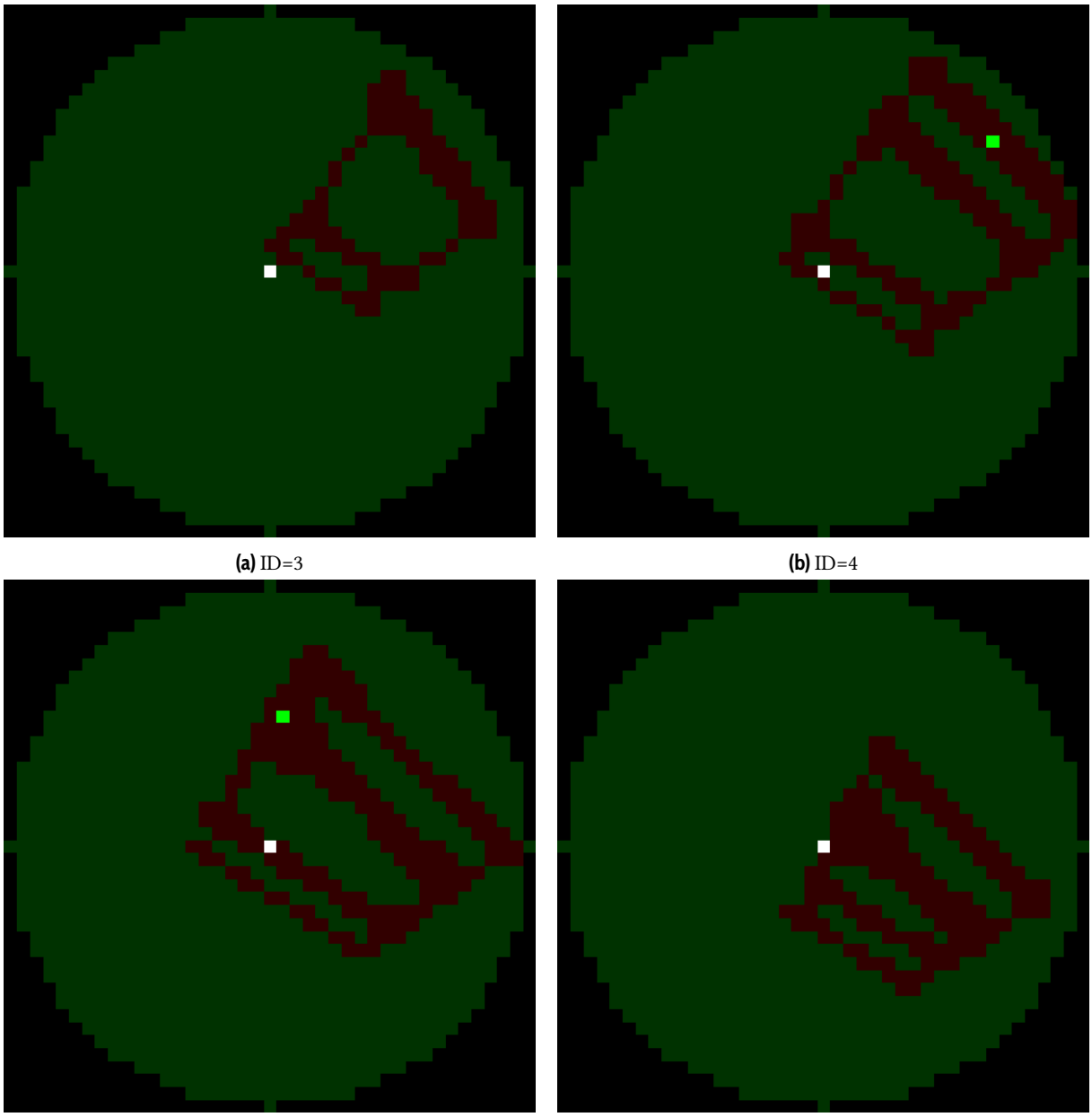
(g) ID=6 **(h)** ID=7 **(i)** ID=8

Figure 7.2. Enemy tanks encoder-decoder result for an input sequence of length 2



(a) ID=8 **(b)** ID=9 **(c)** ID=10

Figure 7.3. Enemy tanks encoder-decoder result for an input sequence of length 3



(a) ID=3 **(b)** ID=4

(c) ID=5 **(d)** ID=6

Figure 7.4. Enemy tanks encoder-decoder result for an input sequence of length 4

7.5. Threshold value

Given that the decoder has a tendency to make more false positives then false negative (i.e. the decoder never fails to detect a tank when there is one), we might try to change the minimum output value for which the prediction is considered positive. First, we removed the sigmoid activation function applied at the output of the decoder. This way the outputs are not constrained between 0 and 1 anymore. Previously, the limit between positive and negative predictions (the threshold) was 0.5 for the outputs values. This means a limit of 0 if we remove the sigmoid activation function (because $\sigma(0) = 0.5$). If we now increase that limit (or threshold value), we should have less positive predictions.

The results for the single-agent enemies encoder with a threshold of 4 are shown on figures 7.5 to 7.8. Only the graphs on which there is at least on tank and/or false positive are shown. The graphs with the remaining IDs are all like figure 7.1d.

Imposing a threshold value of 4 on the enemies encoder yields better results:

- The size of the false positives spots are reduced for all sequence lengths.
- The amount of false positives on tuples where the ID doesn't correspond with any of the observed tanks is reduced. For sequence lengths of 1 and 2, the false positives are almost always only for tuples where the IDs are that of the observed tanks. For sequence lengths of 1 or 2, there is almost never positive predictions for wrong ID values.
- But, we observe false negatives, whereas there weren't hardly any when we didn't used this threshold value. Nevertheless, there are often false positives near the false negatives. This means the decoder thinks that the observed tank is just a couple of cells further than it really is. This is visible on figures 7.5, 7.7c and 7.8d.

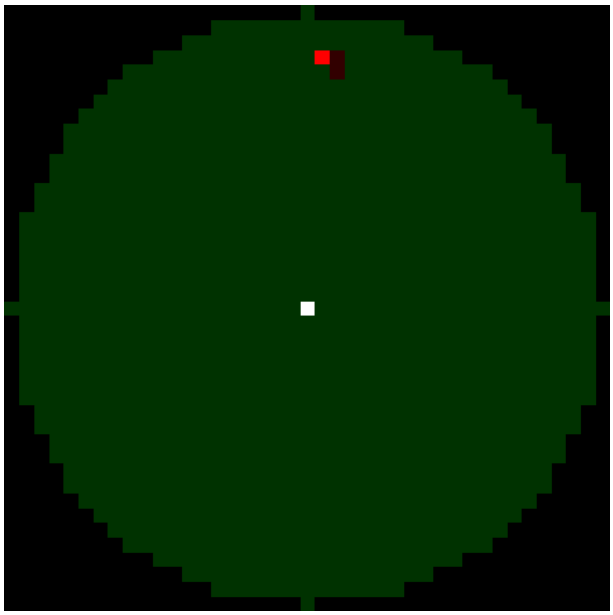
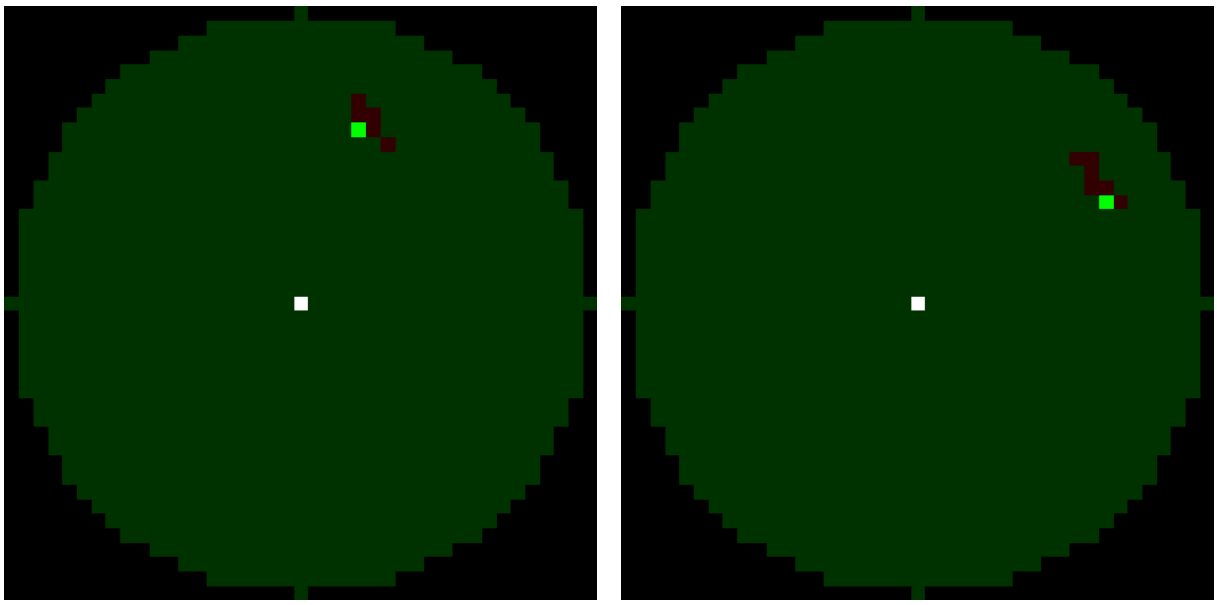
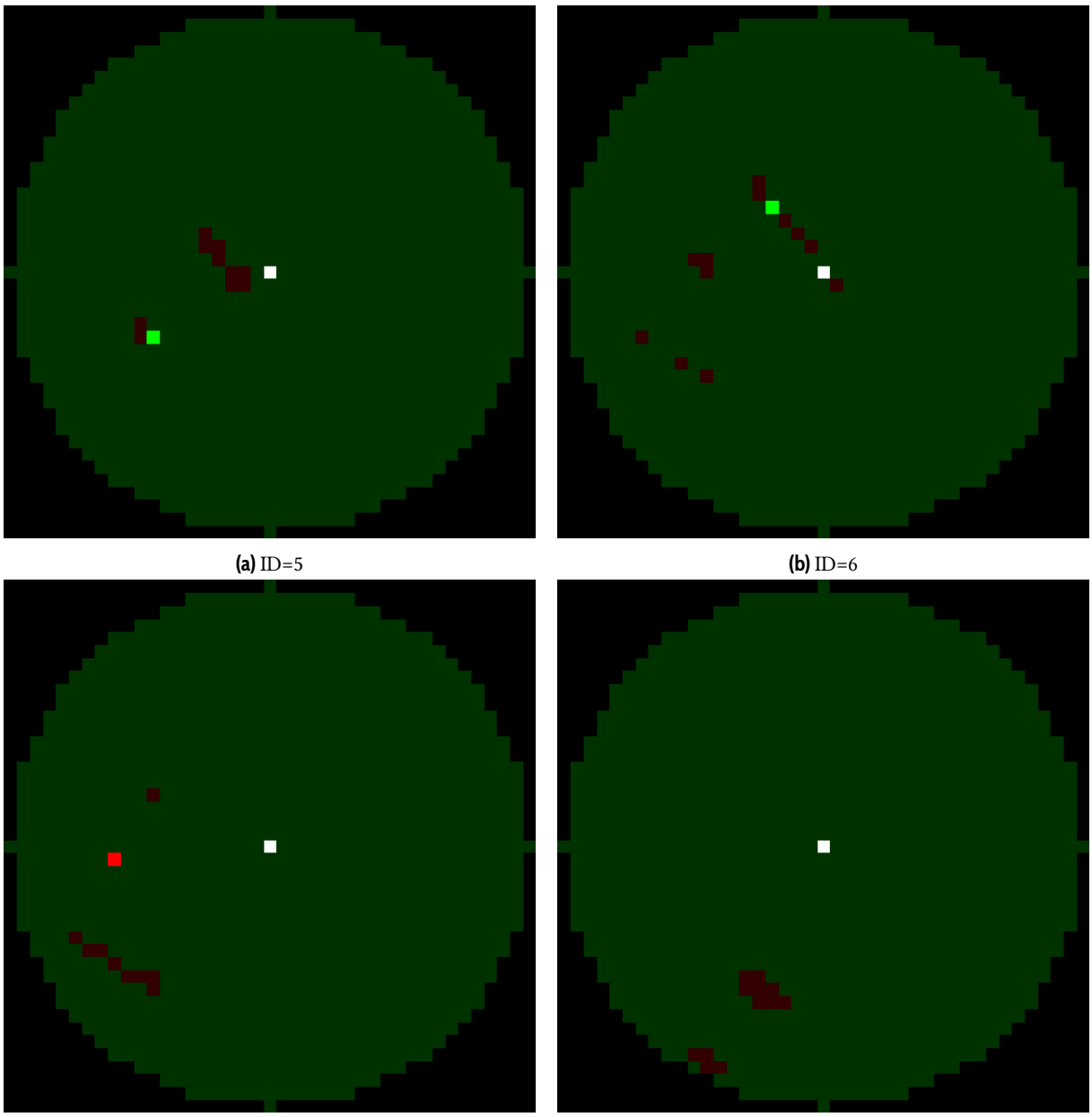


Figure 7.5. Result for an input sequence of length 1 with a threshold value of 4, ID=2



(a) ID=5 **(b)** ID=7

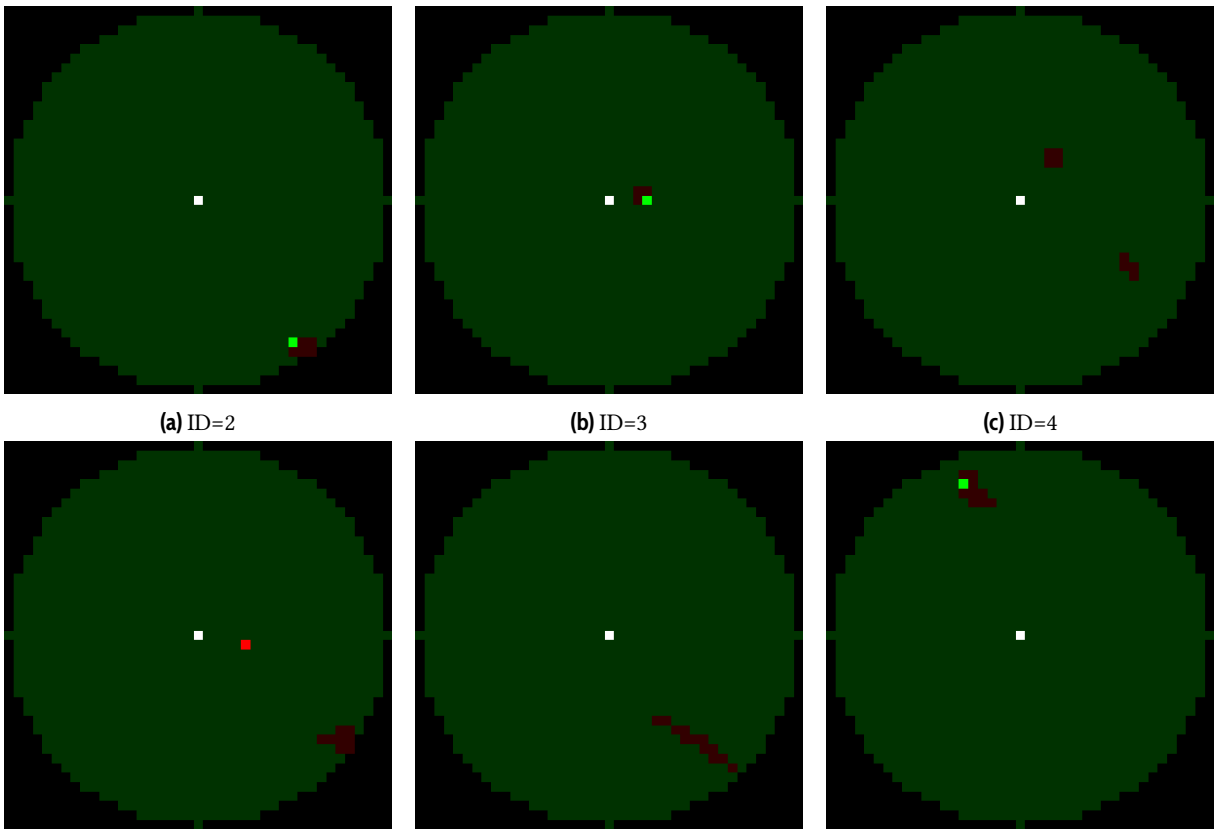
Figure 7.6. Result for an input sequence of length 2 with a threshold value of 4



(a) ID=5 **(b)** ID=6

(c) ID=7 **(d)** ID=9

Figure 7.7. Result for an input sequence of length 3 with a threshold value of 4



(a) ID=2 **(b)** ID=3 **(c)** ID=4

(d) ID=5 **(e)** ID=6 **(f)** ID=10

Figure 7.8. Result for an input sequence of length 4 with a threshold value of 4

7.6. Obstacles encoders

In the previous sections, we only analyzed the results for the single-agent enemy tanks encoder. However, as one could suppose by looking at table 7.4, the results for the foes and friends encoders, be it the full-state or single-agent ones, are very similar. This is why we won't redo this analysis for the three other tanks encoders. However, the results for the two obstacle encoders are less good, hence the need to present and comment them.

Because the obstacle observation sequences are only composed of (x,y) tuples, the full result for an input sequence can be shown on one single image. We did it for several randomly generated observation sequences of various lengths going from 0 to 20 obstacles. Figure 7.9 shows the obtained result.

The results are way less precise than for the other tanks encoders. For each obstacle, there is a cone of (false) positive predictions on the image. And when the obstacles are too close to each other, they blend into the same cone, as shown in figure 7.9c. Furthermore, there are some false negatives. If we try to introduce a threshold value, as we did previously, the results don't improve: the cones don't shrink much and the amount of false negatives increases.

As for the full-state obstacles encoder, the results are very similar. Hence, showing or discussing them individually would add little value to the thesis.

7.7. Discussion

In this chapter, we saw that the different encoder-decoder architectures used to train the encoders for the different variable-length observation sequences didn't reach a 100% accuracy. On one hand, for some encoders, that is the allied and enemy tanks encoders, the results are pretty good. On the other hand, for the obstacle encoders, the obtained accuracy rates after training were significantly lower.

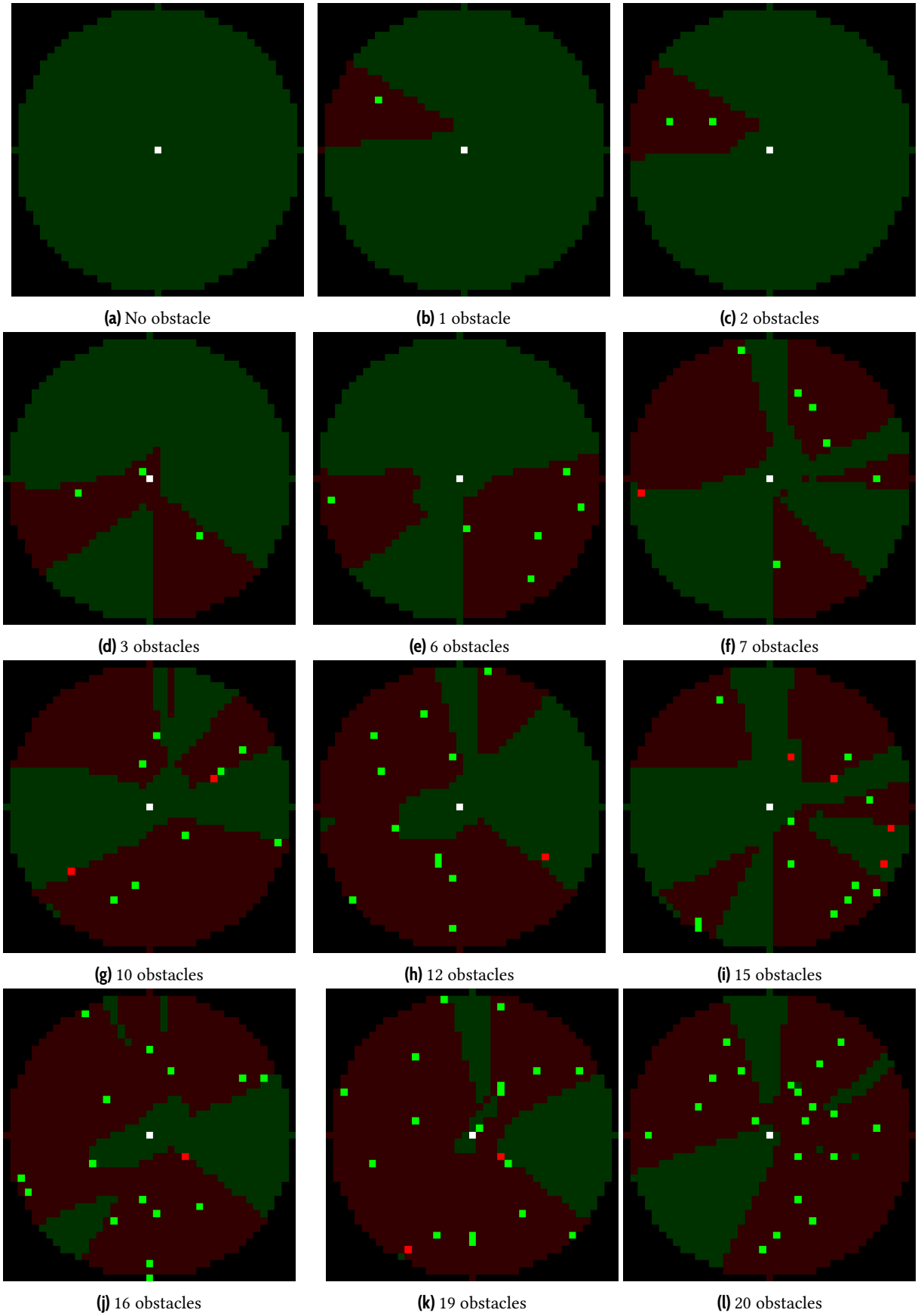
This difference might be explained by the fact that the obstacles observation sequences (up to 20 obstacles) are longer than the other tanks observation sequences (up to 4 other tanks). Indeed, in RNNs, the longer a sequence is, the more difficult it is for the neural network to retain the information from the first elements of the sequence in order to transmit them to the following time steps, via the hidden state. However, this is just an hypothesis. The reason for the less performance of the obstacles encoders could also be bad hyperparameters, such as the size and or amount of layers in the neural networks.

In order to reach better results, we could try to optimize the architecture of both the encoder and the decoder, and maybe change some other parameters, like the learning rate, or the batch size. Another idea cold be to train the encoder during the RL training, in parallel with the agent. However, that would sensibly increase the computing time of the RL algorithms, because the encoder is a complex RNN, and also because it would have to be trained at every RL training.

Nevertheless, the goal of this whole supervised training with a complex encoder-decoder architecture was not per se to obtain perfect prediction results. It was rather to train the different encoders in order to encode variable-length observation sequences into fixed-length encoded observations with a minimal loss of information. Hence, as long as there is enough information in the encoded observations to be able to estimate the Q-values, the encoders are good enough for our application.

Those encoders can now be included in the *tanksEnv* environment to provide fixed-length observations to agents. With this, we can apply the different RL methods described in the previous chapters of this thesis to agents performing in our environment. In the next chapters, we'll show the results of our

implementations of some of those algorithms. We'll then try to draw some conclusions regarding the



(a) No obstacle **(b)** 1 obstacle **(c)** 2 obstacles

(d) 3 obstacles **(e)** 6 obstacles **(f)** 7 obstacles

(g) 10 obstacles **(h)** 12 obstacles **(i)** 15 obstacles

(j) 16 obstacles **(k)** 19 obstacles **(l)** 20 obstacles

Figure 7.9. Obstacles decoder output

results provided by agents trained with encoded observations.

8. Reinforcement learning

In this chapter, we'll implement a version of the DQN algorithm to train single agents in the *tanksEnv* environment. We'll first start with a simple setup (one opponent with random action selection and no obstacles), then we'll increase the complexity: increasing the number of enemy tanks and/or the quality of their policies and/or adding obstacles.

In order to asses the quality of the learning of agents that receive observations processed by the different encoders, we need other agents to compare them with. This is why we'll also train agents that get raw observations (i.e. not processed by the encoders) from the environment. In order to do this, we'll apply a technique described in chapter 5. We'll fix the length of the observation vector to a maximum size, and if the true observation is shorter, we'll fill in the observation vector with dummy values that can never be observed in the environment (ex: a relative position of (0,0)[1], an ID of -1).

In this chapter, we'll train and compare four different agents:

1. *classical*: An agent that gets the raw observations from the environment and whose Q-Network is a FC NN.
2. *encoder*: An agent that gets the encoded observations from the environment and whose Q-Network is also a FC NN.
3. *RNN*: An agent that gets raw observations from the environment and whose Q-Network is an RNN. This means that the Q-Network has access to information from the states observed earlier in the episode to evaluate the Q-values of the actual state.
4. *RNN_encoder*: An agent that gets the encoded observations from the environment and whose Q-Network is an RNN.

8.1. Setup 1

First, we'll implement a very simple setup: one agent[2] is trained against one or several opponent(s) (up to 3). These red agents have a random policy, and there is no obstacle on the battlefield. Because there is only one agent and there are no obstacles, only the single-agent enemy tanks encoder is needed to provide an encoded observation to an agent.

8.1.1. Parameters

Environment parameters

The environment is set up as follows:

- *size*: 10×10
- *agents_description*:
  - One blue agent (the one that is learning) with random start position.
  - One to three red agent(s) with random start position and(s) random actions selection.

[1]The agent will never observe any tank or obstacle at a relative position of (0,0) because two agents and/or obstacles cannot be on the same cell of the environment grid.

[2]When not specified otherwise, "agent" refers to "blue agent", the one that is training

- *visibility*: 7 cells
- *R50*: 5 cells
- *obstacles*: None
- *max_cycles*: 25
- *max_ammo*: 5

Training parameters

The agents are all trained with the following parameters:

- Buffer size: 50 000 transitions
- Batch size: 128 transitions
- Training length:
  - 100 000 episodes for 1 and 2 red agents
  - 200 000 episodes for 3 red agents
- Epsilon decay: exponential with
  - $\epsilon(t = 0) = 1$
  - $\epsilon(t = \text{last episode}) = 0.05$
- Learning rate: 3×10^{-4}
- γ: 0.9
- Target network synchronization period: 25 episodes
- Adam optimizer and MSE loss function

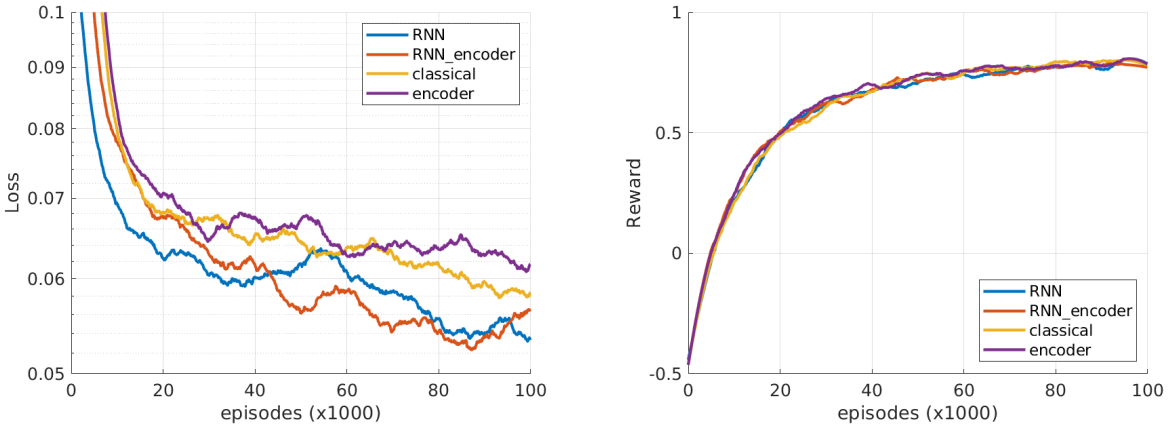
Agents parameters

All the agents Q-Networks are deep neural networks with two layers of 64 neurons. All layers are FC NNs, except for the *RNN* and *RNN_encoder* agents, where the second layer of both their Q-Networks is composed of 64 GRU cells.

8.1.2. Results

For each simulation (one per agent), the result will be two curves: one showing the loss value of the Q-Network, and one showing the cumulative reward, that is the sum of the received rewards (not discounted) throughout the training steps, i.e. one at each episode.
Note: All the reward and loss curves shown in this thesis are smoothed (see appendix C).

The final values of the loss and the sum of the rewards (or total reward), that is the smoothed values at the last episode are shown on table 8.1 to 8.3.

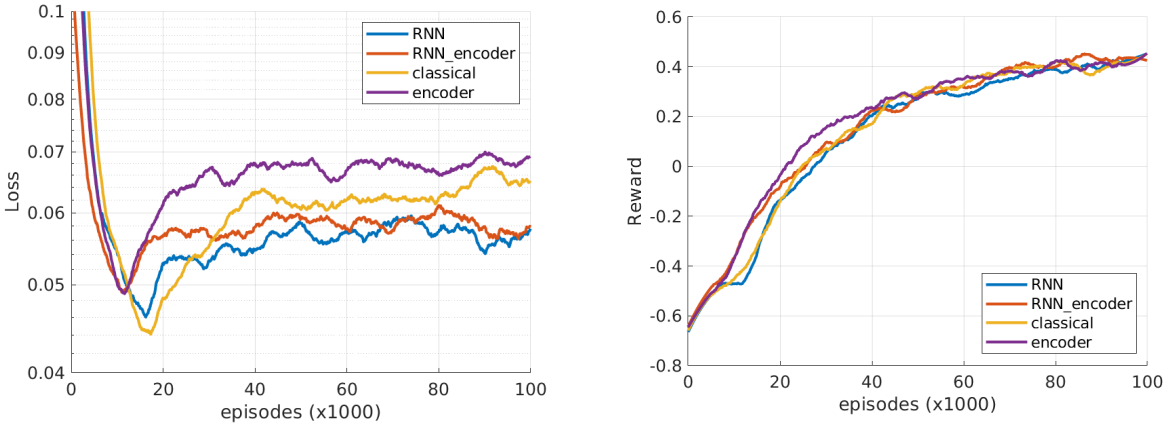


(a) MSE loss **(b)** Sum of rewards

Figure 8.1. Setup 1, one red agent

Agent	Total reward	loss	Computation time [min]
classical	0.80	0.06	10.5
encoder	0.78	0.06	15
RNN	0.80	0.05	27
RNN_encoder	0.79	0.05	33

Table 8.1. Expected sum of rewards for setup 1, one red agent

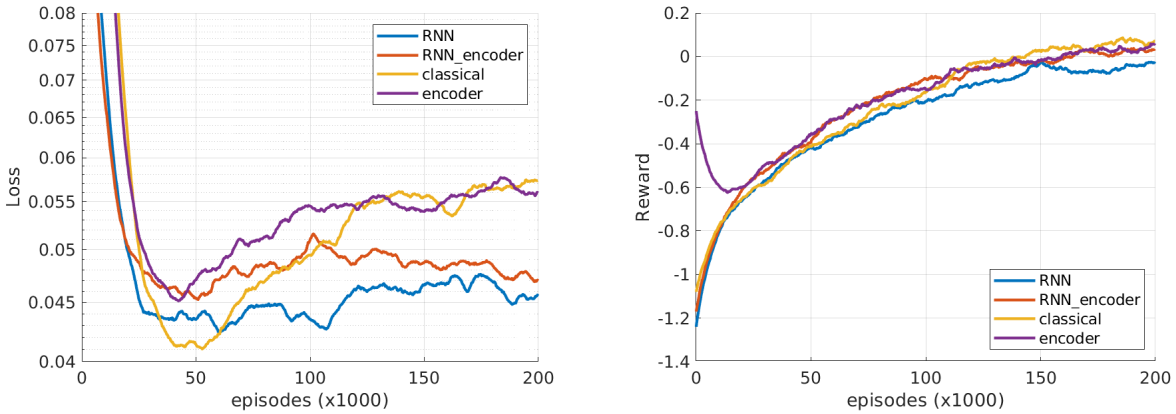


(a) MSE loss **(b)** Sum of rewards

Figure 8.2. Setup 1, two red agents

Agent	Total reward	loss	Computation time [min]
classical	0.40	0.07	16.8
encoder	0.43	0.07	22.3
RNN	0.41	0.06	35.5
RNN_encoder	0.43	0.06	41.3

Table 8.2. Expected sum of rewards for setup 1, two red agents



(a) MSE loss

(b) Sum of rewards

Figure 8.3. Setup 1, three red agents

Agent	Total reward	loss	Computation time [min]
classical	0.05	0.06	44.6
encoder	0.07	0.06	64
RNN	-0.02	0.05	95.4
RNN_encoder	0.05	0.05	105.6

Table 8.3. Expected sum of rewards for setup 1, three red agents

8.1.3. Discussion after the first setup

Because the red agents are playing randomly, the reward values are really noisy. Sometimes the opponents will directly aim then shoot at the blue agent, and sometimes they just won't be dangerous at all and simply move around. In addition to this, there is randomness added by the probabilistic aspect introduced both by the probability to kill in function of the range and the random start positions of all the agents.

Therefore, based on the final reward values obtained by the four agents after their training, we cannot say that some perform better than others. The result values are too close to each other. Still, there seems to be a difference in the loss between the agents with RNNs and the agents with simple FC NNs, both the *RNN* and *RNN_encoder* getting lower values than their FC NN counterpart. So, maybe the RNN helps to better approximate the Q-values, but anyway, it doesn't make any (significant) difference in the obtained rewards.

The only clear difference there appears to be is the computation time. A smaller one between the "encoded observation" agents and the "raw-observation", and a more important one between the RNN and FC NN agents. For the encoders, it seems logical that the training takes a bit more time, because every observation has to pass through a RNN before it is given to the agent. As for the RNN agents, the large difference in computing time can easily be explained. In all the transitions of the batches, the whole history of the observed states is fed to the RNN of the agent that is training, whereas for the non-RNN agents, only the current observation is used to train[3].

In the next sections, we'll add some elements to increase the complexity of the environment. We'll see if the agent learning from the encoded observations still can perform at least as good as the agent learning from the raw observations. Since to focus is more on the difference between raw and encoded

[3]See appendix F for more details about the training of RNN agents.

observations, and as the first setup didn't show any significant difference between the rewards for the RNN and FC NN agents, we will now only train the two agents that have a simple FC Q-Network. This is mainly done to reduce the computation time. Furthermore, and for the same reason, we will also only train the agents against one opponent and not against two or three. Finally, we observed that a training length of 50 000 episodes was enough to have the total rewards value converging to a maximum, so we also reduced the training length in the next setups.

8.2. Setup 2: random IDs

In the environment, when the DQN agent observes the visible red agent(s), it gets information about its (their) x,y position(s) and its (their) ID(s). Based on that, it has to approximate the best way possible the Q-values of all possible actions. There are actually several *aim* actions: one for every tank in the environment, even the tanks that are temporarily not visible by the agent. To make the distinction, every *aim* action always refers to one single ID. Hence, in order to perform efficiently, the agent has to understand that, when it sees agent with ID equals to *i*, it has to take action *aim-i* to aim at this particular tank. So, we might think that, if the agent performs well, then it means that it is able to make the link between getting an observation where the ID equals *i* and choosing action *aim-i*.

However, at each episode, the same tanks always get the same IDs, and the DQN agent always has to aim at the same ID(s). This means the agent might not need to make this link, but just remember which IDs it has to aim at. For instance, if the blue agent has ID 0 and learns to perform against a red agent that has ID 1, it might just learn that when it observes any tank, it automatically has to perform action *aim-1*[4], because it will never have to aim at any other tank. In other words, it is not certain that the agent had to exploit the ID part of the observations in order to determine the correct Q-values. Hence, it is also not certain that the encoded observation still contains this information after the encoding.

For this reason, we started the same training as before, but with one major modification: every time the environment resets and starts a new episode, the agents get new random IDs. We added this feature to the *tanksEnv* environment. It can be enabled by setting the *random_ids* parameter to True.

Given that we now only have two training setups, it is easier to run both of them several times. We ran them six times each and took the mean of the six curves. This result is shown on figure 8.4. Those two curves are the mean of the six series of results for both agents. The mean execution time is about 6.6 minutes for the raw observations and 7.6 minutes for the encoded observations[5].

Also, we observe a higher start of the "raw observations" curve. This is due to the smoothing and the fact that the total reward received at the very first episode was higher.

In addition to this, running the same simulation multiple times allows to make some statistics tests. We ran 100 extra episodes after each training with $\epsilon = 0$ and without training the agents. We then took the average over the 100 values. Thus, we have a total of six values for the expected total reward for both agents. The results are shown in table 8.4. We will perform hypothesis tests on those values[6].

The first test we did is a two sample t-test. The null hypothesis of this test is that the means of the two samples are equal. The result of the test is a p-value of 12.5%. Hence, we cannot reject the null hypothesis with a certitude of 95%[7].

[4]We proved it by letting this agent perform against an opponent that has a different ID than the ID of the tank it trained against. The agent never aimed at this tank and performed really poorly.

[5]Here, the terms raw observations and encoded observations actually refer to the agents that get the raw or encoded observations.

[6]Some key concepts of hypothesis tests are summed up in appendix E.

[7]Typical value used in the scientific literature.

[!ht]

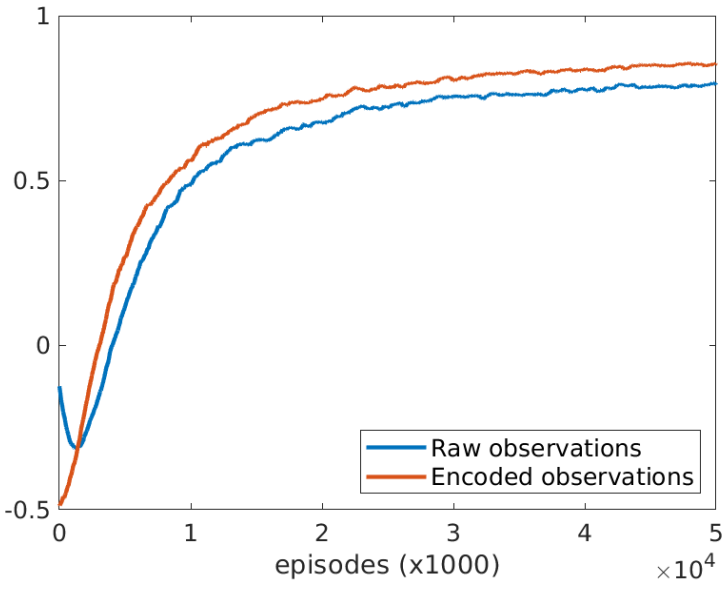


Figure 8.4. Sum of rewards per episode for setup 2

Training n°	Raw observations	Encoded observations
1	0.7954	0.7936
2	0.8267	0.8709
3	0.7486	0.8559
4	0.7591	0.8369
5	0.6997	0.8929
6	0.8837	0.7826
mean	0.7855	0.8388

Table 8.4. Expected sum of rewards for setup 2

We also did a one-way ANOVA test, which has the same null hypothesis. We obtained a p-value of 7.1%. With this result, we still cannot reject the null hypothesis with a certitude of 95%, but we can with a certitude of almost 93%.

Those two tests give slightly different results. However, the two sample t-test can only be used for samples that have the same variance. In order to verify this, we also did a hypothesis test on the variance of the two samples. This yielded a p-value of 62.4%. This time, the higher the p-value, the higher we thrust that the variances of the two samples are equal. Thus, we cannot assume with certainty that the two samples have equal variances and therefore it is safer to use the results of the ANOVA test instead of the two sample t-test.

8.3. Setup 3: obstacles

In this section, we will analyse the results that we get when using the obstacle encoder. For this, we will add some obstacles at fixed positions on the simulated battlefield. The observations that the agents will get will thus also contain the coordinates of all the visible obstacles. Hence, the single-agent obstacles encoder will be used to provide the encoded observations.

Again, the parameters are almost the same as in the previous section. The only difference being the addition of some obstacles in the center of the battlefield. Figure 8.5 shows two examples of the start state. The layout of the obstacles is visible there. We can see that the map is divided into a western and an eastern part, and that there are two passages to the north and south of the obstacles barrier that allow to move between the two parts.

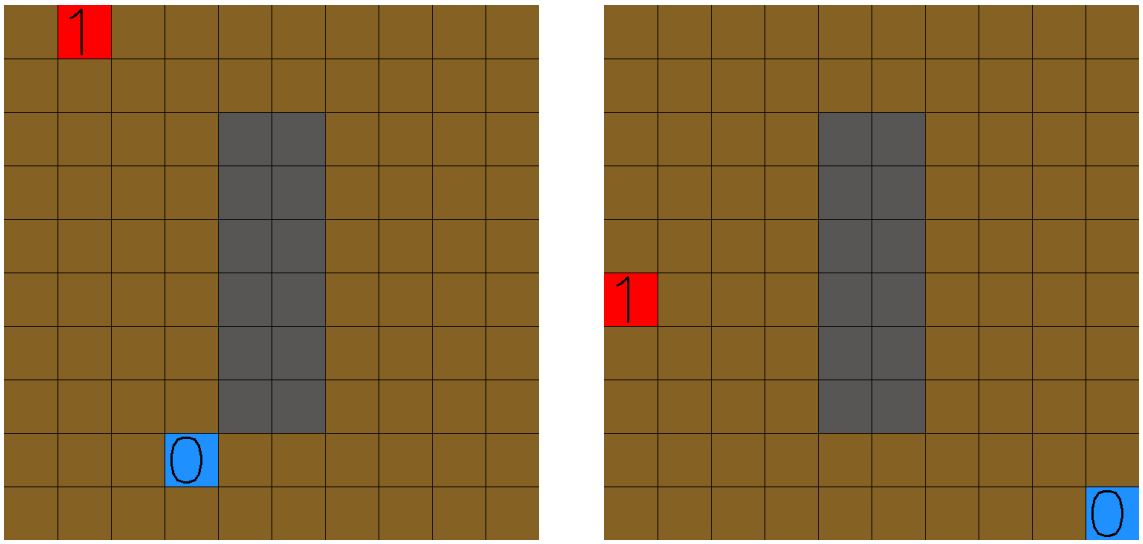


Figure 8.5. Examples of start states for setup 3

As in previous section, we ran the training for the two agents six times. Figure 8.6 shows the mean of the six curves of the reward values for both agents. We also ran the extra 100 episodes with no learning and $\epsilon = 0$ at the end of each training. The mean values over the 100 episodes are shown in table 8.5. The training time for the raw observations agent is around 8 minutes and increases to 24 minutes for the encoded observations agent.

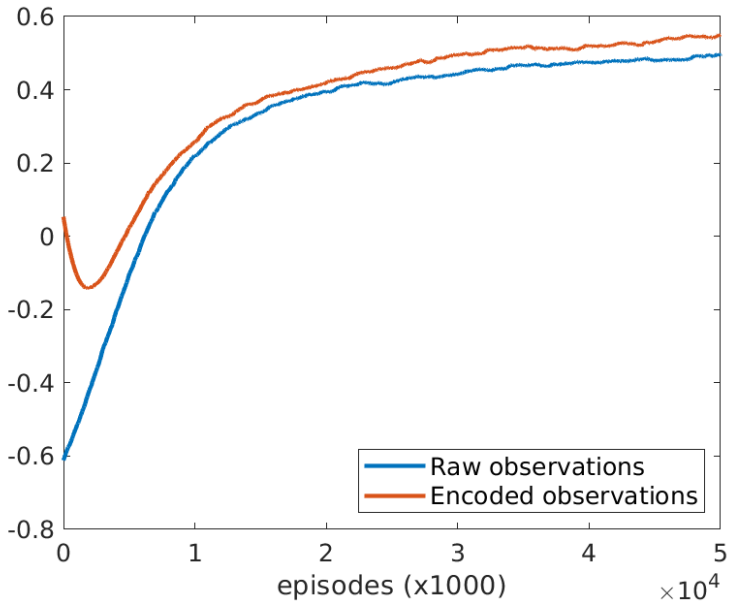


Figure 8.6. Sum of rewards per episode for setup 3

We also did the statistics tests that we did previously. The hypothesis test for the variances of the sample yielded a p-value of 46%. Hence, we only did the ANOVA test, because we cannot be sure that the variances are equal in both the samples. The p-value of the ANOVA test was 0.27%. This means

Training n°	Raw observations	Encoded observations
1	0.4774	0.6531
2	0.4859	0.5531
3	0.4406	0.6521
4	0.5412	0.3571
5	0.3926	0.5074
6	0.3671	0.6471
mean	0.4508	0.5617

Table 8.5. Expected sum of rewards for setup 3

we can conclude with a very high level of certainty ($> 99.7\%$) that the agents that train with encoded observations are able to reach higher rewards than the agents that train on raw observations.

8.4. Setup 4: random obstacles

By training on an environment where the obstacles are always at the same positions, we might encounter a phenomenon similar to what we explained in section 8.2. That is, the agents won't learn to exploit the information from the obstacles observation. In fact, when a part of the environment state is constant through all the episodes and transitions, it is not necessary for the agent to get this part of the state in its observations in order to perform well. In this case, since the obstacles are always at the same positions, the agent doesn't need to get those positions in its observations. The obstacles are just part of the environment, in the same way as the size of the grid or the $P_{kill}(range)$ are parts of the environment.

For this reason, in order to force the agent to take the obstacles part of the observations into account, we will again train the agents with a similar setup as the previous one, but this time the obstacles will have random positions.

As for the previous setups, we did 6 simulations for the two training agents (raw and encoded observations). The progression of the reward throughout the episodes is shown on figure 8.7, and the mean results of the 100 test episodes after the training are shown on table 8.6. The training time for the raw observations agent is around 8 minutes. For the encoded observations agent, it increases to 25 minutes.

Training n°	Raw observations	Encoded observations
1	0.6351	0.7881
2	0.5563	0.6879
3	0.616	0.6937
4	0.5803	0.745
5	0.6727	0.6376
6	0.6744	0.7017
mean	0.6225	0.7090

Table 8.6. Expected sum of rewards for setup 4

We also did the same statistics tests. The two sample t-test and the ANOVA test yielded exactly the same result: a p-value of 1.3%. So, once again, we can state that the agent that gets encoded observations learns a better policy, that allows it to get higher rewards.

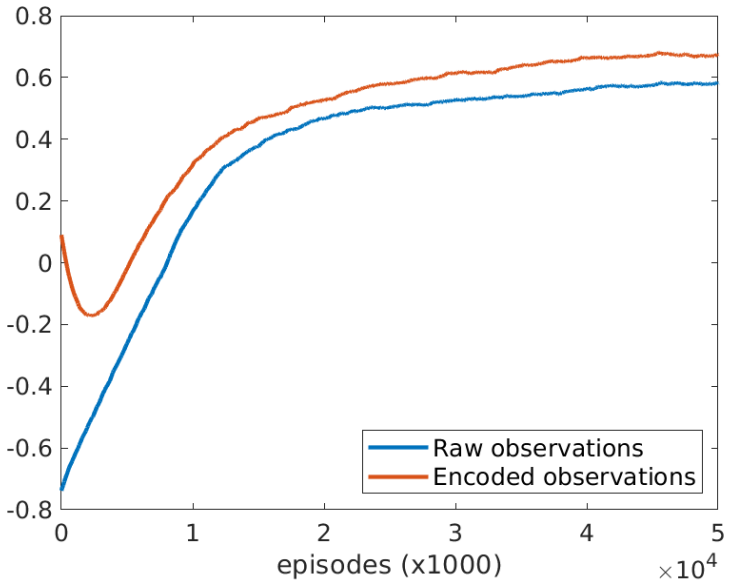


Figure 8.7. Sum of rewards per episode for setup 4

8.5. Multi-Agent algorithms

Up until now, we only implemented DQN, which is a single-agent RL algorithm, we could only test the single-agent obstacles and enemy tanks encoders. In this section, we will try to determine whether our encoding technique is still able to work when combined with Multi-Agent Reinforcement learning algorithms. Also, this will allow to test the allied tanks encoder, as well as the full-state encoders, when using QMix.

8.5.1. Parameters

We'll first describe the setup for the training. It is really similar to the setup for single-agent training.

Environment parameters

- *size*: 10 × 10
- *agents_description*:
  - Two blue agents (the one that are learning) with random start position.
  - Two red agents with random start position and random actions selection.
- *visibility*: 7 cells
- *R50*: 5 cells
- *obstacles*: None
- *max_cycles*: 25
- *max_ammo*: 5
- *random_ids*: True

Training parameters

The agents are all trained with the following parameters:

- Buffer size: 100 000 transitions

- Batch size: 128 transitions
- Training length: 100 000 episodes
- Epsilon decay: exponential with
  - $\epsilon(t = 0) = 1$
  - $\epsilon(t = \text{last episode}) = 0.05$
- Learning rate: 1×10^{-3}
- γ: 0.9
- Target network synchronization period: 25 episodes
- Adam optimizer and MSE loss function

Agents parameters

We will again train one agent that gets encoded observations and one that gets raw observations. The Q-Networks of both the agents are fully connected deep neural networks with two layers of 64 neurons. The hidden layer of the mixer network in QMix is composed of 32 neurons.

The simulations take much more time for the MARL algorithms. This is why, as in the first DQN setup, we only ran one training per algorithm and per agent.

8.5.2. VDN

We will first implement the simplest of the CTDE MARL algorithm that we presented in chapter 4: VDN. Since there are now several agents in both teams, the observation will contain information about the allied tanks and we will thus use the allied tanks encoder to compute the encoded observation.

Figure 8.8 show the total reward per episode for the VDN training. The smoothed value of the total reward at the last episode are: 0.52 for the encoded observations agent and 0.48 for the raw observations agent. Again, we get a slightly higher result for the agent that trains on encoded observations. However, for this setup we are not able to prove that there is a significant difference. Besides, the training time are about 3h35.

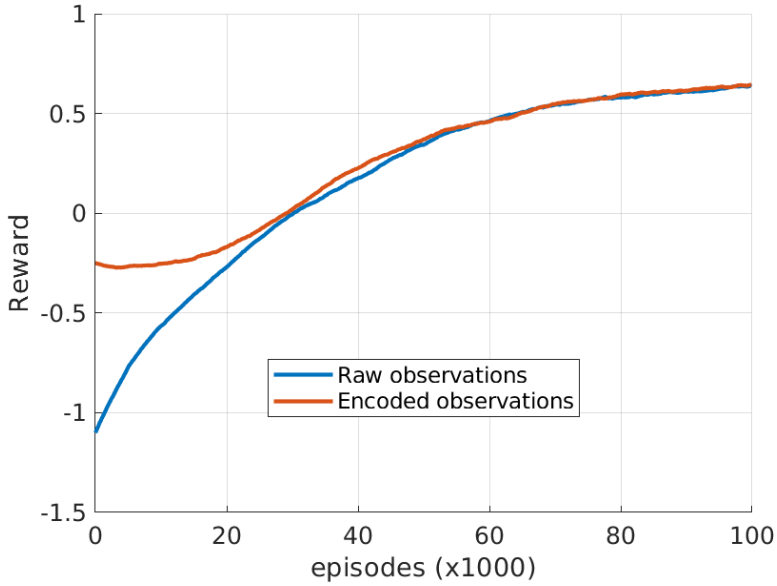


Figure 8.8. Results for VDN training

8.5.3. QMix

We will now implement the QMix algorithm with the same setup as for VDN. Because QMix needs full-state observations, the full-state encoders will also be used during this training.

Figure 8.9 show the total reward per episode for the QMix training. The smoothed value of the total reward at the last episode are respectively 0.78 and 0.79 for the agents training with encoded and raw observations. The computation time is about the same as for VDN.

Once more, the encoded observations agent performs about as well as the raw observations agent. We can also observe that QMix reaches way higher rewards after the training. However, it is possible that VDN could reach higher rewards than it did with the 100 000 training episodes, as in the paper about QMix [RSS+18], the authors mention that VDN needs more learning episodes than QMix to provide good results.

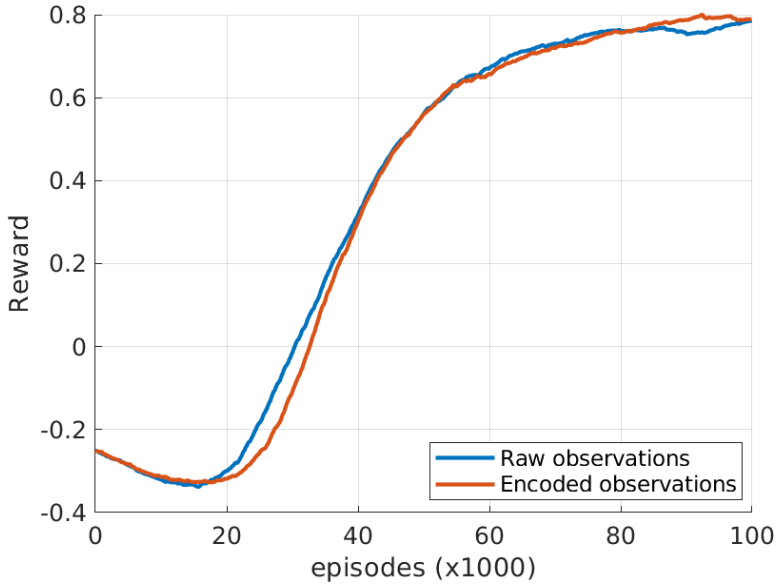


Figure 8.9. Results for QMix training

8.6. Discussion

With the different learning setups, we have seen that the agents that learned to perform in the *tanksEnv* environment with encoded observations performed at least as well as the agents that learned from raw observations. For some setups, the encoded observations agent even outperformed its counterpart. This result is rather surprising, as both agents actually receive the same information. The only explanation that we can provide is that the encoder acts as (a) supplementary layer(s) in the Q-Network of the agent, in spite of the fact that the weights of the encoder network are frozen during the learning.

In machine learning, up to a certain extent, expanding the network (putting more layers and/or more neurons in the layers) often offers better performance, hence the better results of the encoded observations agent. The drawback of this is the longer computation time. We could look at it differently, and say that it is simply easier to learn from information that has already been pre-processed by the encoder than from raw information.

Nevertheless, the intent of this thesis is to develop variable-length observations encoders and determine whether RL algorithms can learn from encoded observations. So, it appears that, in our study

case, the DQN agent is able to learn from the encoded observations.

Another point that we noticed is the reward that the agent get in the beginning of their training. They are rather low. In the early episodes, all agents have random policies, we might thus expect the reward to be around zero, or a bit lower due to the reward of -0.01 for each step taken. However, the values we get are more around -0.5 or -0.6, and sometimes even below -1. Our explanation is that when a red agent's policy is set to random, the environment will select the agent's action only amongst the action that he is authorized to take. For instance, it will never move against a wall, or aim at a tank it cannot see, or shoot if it has not aimed at another tank previously. On the other hand, the blue agent acts according to its own policy, that happens to be random at the start of the training. In this case, it chooses its actions amongst all the actions that the environment proposes. The environment will simply cancel the action if it is not possible. Thus, in the beginning, the red agents have a higher probability to randomly select the actions that will lead them to win the episode.

9. Conclusion and discussion

9.1. Conclusion

In this thesis, we first developed a new environment called *tanksEnv*, which was deigned to be a framework in which we can train agent with both single-agent and multi-agent reinforcement algorithms. Besides, it also allows to train those agents with raw fixed-length observations, or with variable-length observations that first have to be processed by (an) encoder(s) before being passed on to the agent.

We then developed a method to train encoders with a complex architecture that integrates a decoder. This method allows to train encoders with supervised learning only once, and also not in parallel, but before the RL training. This significantly reduced the time of the latter. The test results of the training were mitigated. The accuracy of our encoder-decoder architecture seems to decrease as the observation sequence becomes longer. As already said in chapter 7, a deeper study of the encoder and decoder architectures and/or of the training parameters could be done to improve the obtained results.

However, the goal of the encoders was not to reach perfect results during the supervised learning, but to encode observation sequence in a way that would allow RL algorithms to be able to train agents getting those encoded sequences as observation. Hence, we implemented our own version of DQN, as well as two MARL algorithms: VDN and QMix. Given that the *tanksEnv* environment allows to get observations in a fixed-length mode or in variable-length mode, we could train some agents that learned with encoders and others that learned without encoders. We did this for different training setups.

The results were surprising. For some setups, the agents learning from encoded observations even outperformed the agents learning from unencoded raw observations. Our explanation to this is that it is easier to learn from already pre-processed data, than from the raw data. In other words, the encoder makes it like there are some extra layers in the neural network of the agent, making it wider and more complex, which, in machine learning, often allows to reach better results. However, we could not prove this, and it is just a hypothesis.

9.2. Belgian Defense

As we explained in the introduction, this thesis is a follow-up to [BOE20], which is a thesis that has been written within the framework of the IRIS project, which final aim is to develop a software that we hope will be able to assist crews of armored vehicles during missions on real battlefields and allow to increase the efficiency of such vehicles when they are deployed. Hence, the interest for Belgian Defense is obvious.

Furthermore, with the growing importance of artificial intelligence and the increasing progress in this field of research, it is highly probable that such technologies will start to develop and will soon be a powerful tool used by some nations to increase their efficiency on the battlefields. Belgian Defence should invest in such projects if they don't want to be left behind in this area of development.

9.3. Ideas for further improvements

While working on this thesis, we had some ideas that are worth looking into and could be the subject of future research projects in the continuity of this thesis, or in the framework of the IRIS project.

A first idea would be to try different ways to train the encoders. With the supervised learning method that we used in this thesis, the distribution of the samples in the training sequences is not the same as the distribution of the sequences that the agents may encounter in the environment. Besides, it is also possible that not all of the information from the observation sequence is needed.

So, we could for instance put one or several untrained encoder(s) (depending on the need) before the Q-Network and train the Q-Network and the encoders together during the reinforcement learning process. Otherwise, instead of generating the training batches randomly, we could store "real" observations that the agents get when performing in the environment and then use the same supervised learning method as in this thesis, but with those samples. Then, we could compare the obtained results with the method we proposed in this thesis.

Another idea would be to make the environment more realistic by adding some features. Then, see if the agents can develop more subtle strategies. For example, strategies take advantage of more complex terrains. At the same time, since the bottleneck for the computing time is the environment, we could use the opportunity to optimize the code and make it faster.

Different ideas of new features for the environment are:

- Allow for different types of terrains that have their own characteristics, for example: camouflage, speed of movement, ...
- More parameters for the tanks: fuel, fuel consumption, different types of ammunition, different types of armor, different types of armor protection depending on the orientation, ...
- Allow for different types of tanks on the battlefield
- Aiming system based on the azimuth and elevation of the barrel, instead of the ID of the tanks.
- Effects of the weather: wind, rain, ...

We could also work in collaboration with people that have experience about real battlefields and might know more about what are important features of the real environments that should be added in the simulated one.

A. Supplementary Information

A.1. Probability of kill as a function of range

Table 9 from [CRG19] reports the results of an experiment where the ballistic dispersion is measured as a function of range. This is done for one particular laboratory weapon. Table A.1 and figure A.1 show this experimental data as well as a fourth-degree polynomial approximation. This approximation is:

$$\hat{\sigma}(R) = 9.53 \times 10^{-13} R^4 - 9.28 \times 10^{-10} R^3 + 3.92 \times 10^{-7} R^2 + 2.42 \times 10^{-5} R + 0.243 \tag{A.1}$$

Let's make some assumptions:

- The shape of the function that links ballistics dispersion and range is the same for this experiment and for the tanks in the *tanksEnv* environment.
- The aim point (i.e. the mean point of the impacts) of a tank is exactly positioned at the center of the target.
- The target is perfectly round.
- The error E (or ballistic deviation) of one shot is measured as the distance between the center of the target and the actual impact point.
- The error E follows a half normal probability distribution. The CFD (cumulative density function) of such a distribution is $F_c(e) = P(E < e) = erf\left(\frac{e}{\sqrt{2}\sigma}\right)$
- The target is always killed when hit.

Range [m]	std [mils]
50	0.25
100	0.25
150	0.25
200	0.26
250	0.26
300	0.27
350	0.27
400	0.28
450	0.29
500	0.3
550	0.31
600	0.32
700	0.36
800	0.43

Table A.1. Standard deviation of the ballistic dispersion as a function of range

According to the previously stated assumptions, the probability to hit the target is:

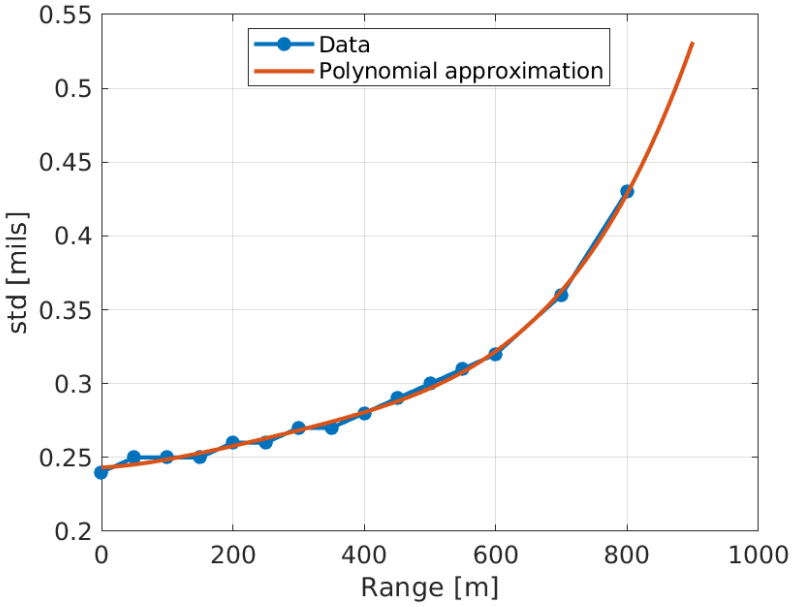


Figure A.1. Standard deviation of the ballistic dispersion as a function of range

$$P_h(R) = P(E < d/2)\Big|_{\sigma=\hat{\sigma}(R)} = erf\left(\frac{d/2}{\sqrt{2}\hat{\sigma}(R)}\right) \tag{A.2}$$

where d is the diameter of the target. This parameter doesn't influence the final result since it will be removed by the normalization and replaced by the R50 parameter.

Let's now try to express the function $P_k(R)$ (probability of kill in function of the range) that will be used in the *tanksEnv* environment. One of the parameters of the environment is $R50$, the range at which the probability to kill is 50%. Moreover, for $d/2 = 1$, P_h(R=1269m) $= 0.5$. Hence (given that the target is always killed when hit):

$$P_k(R; R50) = P(k|h)\, P_h\left(\frac{1269R}{R50}\right) = erf\left(\frac{1}{\sqrt{2}\hat{\sigma}(\frac{1269R}{R50})}\right) \tag{A.3}$$

Figure 6.1 shows P_k as a function of the normalized range. This final result resembles a lot to the "VISUAL RE" curve of single shot hit probability in [Var14] (slide 13), which is based on experimental data from APDS (armor piercing discarding sabot) shots on 2.3m by 2.3m NATO standardised targets.

B. Computing time and hardware

B.1. Comment on the computing times

In this thesis, computation times will sometimes be mentioned. They are not to be considered as precise values, but rather to compare the eventual differences of order of magnitudes between values obtained during different simulations. The main external factor that influences the computation time (if simulating the same code on the same piece of hardware) is the other tasks running on the computer. In this case, it was a personal laptop that ran the simulations. The computer was also used for other tasks during simulations: working on other parts of this thesis (ex: generating plots or implementing algorithms and testing them), daily use, ... This could have had a significant influence on the computation times.

B.2. Hardware

Moreover, the computer used to make all the computation tasks needed for this thesis was composed by the following major pieces of hardware:

- MSI Prestige 15 A10SC (16S3.1) laptop
- Micro-Star MS-16S3 Motherboard
- Intel Core i7-10710U CPU, 6 cores @3.2GHz (boost 4.7GHz)
- 2*8GB DDR4 RAM memory @2667MHz
- Nvidia GeForce GTX 1650 Mobile / Max-Q GPU with
  - 1024 cores @1020MHz (boost 1225MHz)
  - 4GB GDDR5 128bit memory

B.3. Profiling of the learning programs

Profiling a program consists of analyzing the time and memory resources it consumes, and more precisely, what parts of the program consume the most. In this case, we will focus this profiling on the computation time.

B.3.1. Simple DQN

We did an analysis of the computation time with the cProfile Python library. This tool tracks all function calls of our program, as well as the processing time an number of repetitions of those calls. We did this for the the fastest DQN setup from this thesis:

- One blue DQN agent, FC NN Q-Network with two layers of 64 neurons
- One red random agent
- Raw observations (not encoded)
- 50 000 episodes

- 10×10 grid
- No obstacles
- *max_cycles* = 25
- Visibility: 12 cells
- R50: 7 cells
- Adam optimizer and MSE loss
- All PyTorch calculations (NN call, loss and gradient computation, optimizer steps, ...) are done on the GPU

This code takes about 4 minutes and 30 seconds to run, of which about $1/3 \sim 1/4$ is used by the PyTorch operations. The most important ones being (in order of importance):

1. Converting Python lists to PyTorch tensors data types.
2. Calling the Q-Network (for action selection and learning).
3. Computing loss from the batch.
4. Running the backpropagation algorithm to compute the gradient of the loss.
5. Taking a step forward with the Adam optimizer.

Almost all of the remaining $2/3 \sim 3/4$ of the computation time are used by the environment. The most time consuming functions are (in order of importance):

1. The *get_observation(agent)* method that computes a local observation of the environment's state for the specified agent. This is not used by agents with a random policy.
2. Select the random action for the red agent amongst the action that are available for it. This means, verify whether every action is possible or not in the current state.
3. The *step(action)* method with which the environment updates its inner state every time an agent takes an action.

B.3.2. Encoder and RNN

Now, if we take the same setup as before and change only one parameter at the time, this is the change in computing time that we get:

- Encoded agent observation instead of raw observation (only the single-agent enemy agents encoder is used): +25% computing time.
- RNN Q-Network instead of FC NN: $+200 \sim 250\%$ computing time

When switching to encoded observation, the increase of time is mainly due to the post-processing of the observation by the environment with its encoder(s). This means the environment uses an even bigger proportion of the computing time. On the other hand, including recurrence in the agent's Q-Network will rather increase the PyTorch operation, as the agent has to learn from whole sequences of states.

So, it appears clearly from this analysis that the most time consuming parts of the learning programs are the environment calculations. That is, if we don't use RNNs, because in that case the computing of long sequences of observed states that pass multiple times through the RNN, as well as the computation of the gradient will take even more time than the environment's computations. This is why we put more efforts into implementing the environment and the handling of RNNs in a efficient manner.

B.4. Improvements

This section presents some methods used in the code that are particularly time consuming, as well as some new approaches that allowed to reduce significantly the processing time of those methods.

B.4.1. The *is_visible* method

Every time the environment has to provide an observation of the state to an agent, it has to compute whether each observable element (other tanks and obstacles) is visible by the observing agent. This means: verify whether the observable element is nearer than the visible range, as well as verify that there is no obstacle between the observing agent and the observable element. The Python code of the *is_visible* method is shown in listing B.1. *pos1* is the position in (x,y) coordinates of the observing agent and *pos2* the position of the observable element.

```
def is_visible(self,pos1,pos2):
        if pos1 == pos2:
            return True
        dist = norm(pos1,pos2)
        if dist > self.visibility:
            return False
        LOS = self.los(pos1,pos2)
        for pos in LOS:
            if pos in self.obstacles:
                return False
        return True
```

Listing B.1 *is_visible* function from *tanksEnv*

This second verification is not so straightforward. The environment first has to find all the cells that are on the line of sight (LOS) between *pos1* and *pos2*. Then, it verifies if any of those LOS cells matches with one of the coordinates of the obstacles.

The most time consuming part of this method is the computation of the LOS. Initially, it was computed every time *is_visible* was called, using the function shown in listing B.2.

```
def los(vect1,vect2):
    [x1,y1] = vect1
    [x2,y2] = vect2
    N = 100*round(norm(vect1,vect2))
    dy = (y2-y1)/N
    dx = (x2-x1)/N

    x_iter = [round(x1+i*dx) for i in range(N)]
    y_iter = [round(y1+i*dy) for i in range(N)]

    LOS = list(set([(xi,yi) for (xi,yi) in zip(x_iter,y_iter)]))

    LOS = [[x,y] for (x,y) in LOS if [x,y] != vect1 and [x,y] != vect2]
    return LOS
```

Listing B.2 Computation of the LOS between two positions

This took really too much time. Hence, the second method: we calculated the LOS between the $(0, 0)$ coordinates and every possible (x, y) coordinates where x and y are both positive. Then, we stored this in a dictionary *los_dict* whose keys are (x, y) coordinates, and the values are the corresponding LOS vectors. We finally saved this dictionary in a separate file.

Now, when the *tanksEnv* is started, it loads the *los_dict* dictionary. Then, the LOS is computed using a much faster function that uses the loaded dictionary. Listing B.3 shows this function.

```
def get_los(vect1,vect2,los_dict):

    if vect2[0] < vect1[0]:
        vect1,vect2 = vect2,vect1

    diff = [vect2[0]-vect1[0],vect2[1]-vect1[1]]
    mirrored = False
    if diff[1] < 0:
        mirrored = True
        diff[1] = -diff[1]

    los = [[i+vect1[0],j*(-1)**mirrored+vect1[1]] for [i,j] in los_dict[tuple(diff)]]
    return los
```

Listing B.3 Computation of the LOS using the *los_dict* dictionary

B.4.2. Supervised learning of the single-agent obstacles encoder

As said in chapter 7, the computing time for the supervised training of the encoders is about 4 hours. However, it was previously higher by a factor of 3, 4 or even 5 for the single-agent obstacles encoder.

This was due to the fact that it is not possible to see through obstacles. Hence, when generating the training batches, we had to ensure that no obstacle from the input sequence was actually hidden behind another one. The code that did this was even heavier than the first version of the LOS calculation. We first tried to approach this problem as we did for the LOS computation, by doing all the calculations before hand and storing them in a dictionary. But still, even if the computation time was reduced a great deal with this technique, the final result was still too slow.

At the end, we just gave up the idea of ensuring that no obstacle was hidden behind another one from the same learning sequence. The learning time was then the same as for the other encoders. Moreover, not implementing this feature shouldn't really prevent the encoder from reaching a high accuracy. It will only unnecessarily make the training data more diversified. These extra observations that we didn't prevent from being in the training data will just never be encountered in the real application of the encoder. Of course, this is just a supposition and we weren't able to prove it.

C. Time series data smoothing

In this thesis, the graphs showing the loss and reward data during training must be smoothed. The computed loss highly depends on the batch sampled from the experience buffer, and the obtained reward also depends on the actions chosen by the opposing team (the red agents). These are random processes, that introduce a lot of variability in the data measured during the different episodes, even between two consecutive episodes. Hence, the loss and reward values are very noisy. As a result of this, it would be almost impossible to analyze unsmoothed curves.

The smoothing method we choose is the exponential moving average (EMA) algorithm. It is an infinite impulse response (IIR) filter that has a parameter α allowing to control how much the curve will be smoothed. The higher α, the smoother the curve.

The code of our implementation of the EMA algorithm is shown in listing C.1. Figure C.1 shows three curves based on the same time series: one with $\alpha = 0$ (unsmoothed), one with $\alpha = 0.9$ and one with $\alpha = 0.995$. The unsmoothed data is a sample from a standard normal distribution.
Note: In this thesis, all the plots, and thus also the code related to them, are generated with Matlab®.

As we can see on the figure, for higher values of α the curve takes time to converge to the mean value. This is due to the fact that the first value of the smoothed time series is set to the first value of the unsmoothed series. And, if that first value in the unsmoothed time series is far away from the expected value at that time step (i.e. the noise value in the first sample is high), the first value of the smooth data will also be far from the mean value, in this case 0.

On the other hand, using a higher value for α yields much smoother curves. The dilemma is thus to find the right compromise that fits the best the data (not too high α), but that isn't too noisy (not too low α).

```
function smooth_values = smooth(values,alpha)

    last = values(1);
    smooth_values = zeros(size(values));
    for i = 1:length(values)
        last = last*alpha + values(i)*(1-alpha);
        smooth_values(i)  = last;
    end
end
```

Listing C.1 Exponential moving average algorithm

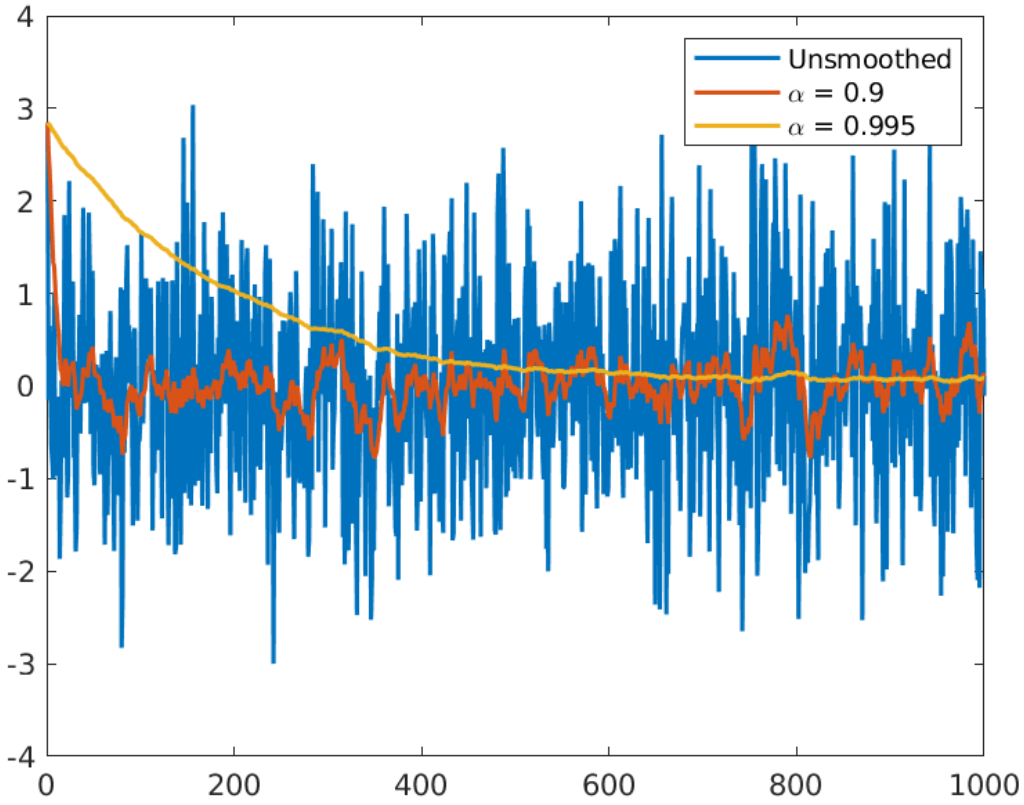


Figure C.1. Smoothing applied on a reward curve

D. Deep Q-Network algorithm

D.1. Implementation

This section describes our implementation of the DQN algorithm. It has all been written in Python, using the PyTorch library for the deep learning part. The algorithm is decomposed in three main parts, or *class* in python:

- An environment, which we discussed in chapters 2 and 6. It provides the observations of the states and the rewards, depending on the selected actions.
- An agent, which holds a Q-Network, can select actions from it applying an epsilon-greedy policy, and also can update its network's parameters by learning from batches containing (S_t, A_t, R_t, S_{t+1}) tuples, as we explained in chapters 2, 3 and 4.
- A runner, that transfers the information between the environment and the agent, and stores the transitions in its buffer. The runner also "translates" the data between the agent and the environment. The environment often gives a simple list of values, but the agent requires a specific type of data that can be fed to the Q-Network, in our case a PyTorch tensor. Also, if the training is done on a GPU, the tensors have to be transferred between the CPU for the environment and the GPU for the machine leanring computation and training.

Figure D.1 shows a diagram of the different actions that the runner executes. Basically, it is the DQN algorithm:

1. Get the state observation from the environment and create an new empty transition.
2. Store observation S_t in the current transition and feed it to the Q-network of the agent (green arrows on the figure).
3. Get the action chosen by the agent based on its internal policy.
4. Store the action A_t in the current transition and forward it to the environment (red arrows on the figure).
5. Store the reward R_t and the new state observation S_{t+1} that the environment provides for the chosen action in the current transition (blue arrow on the figure).
6. Append the current transition to the buffer. If the buffer is limited in size and is full, remove the oldest element.
7. Start again from step 1 with the current S_{t+1} as S_t.

The runner also executes some other actions during the training:

- Reset the environment when the episode is done
- At each episode, sample a batch from the experience buffer and give it to the agent so it can learn from it.
- Synchronize the Q-Network with the target network.
- Store some interesting values such as the loss or the cumulative reward at each episode, for further analysis.

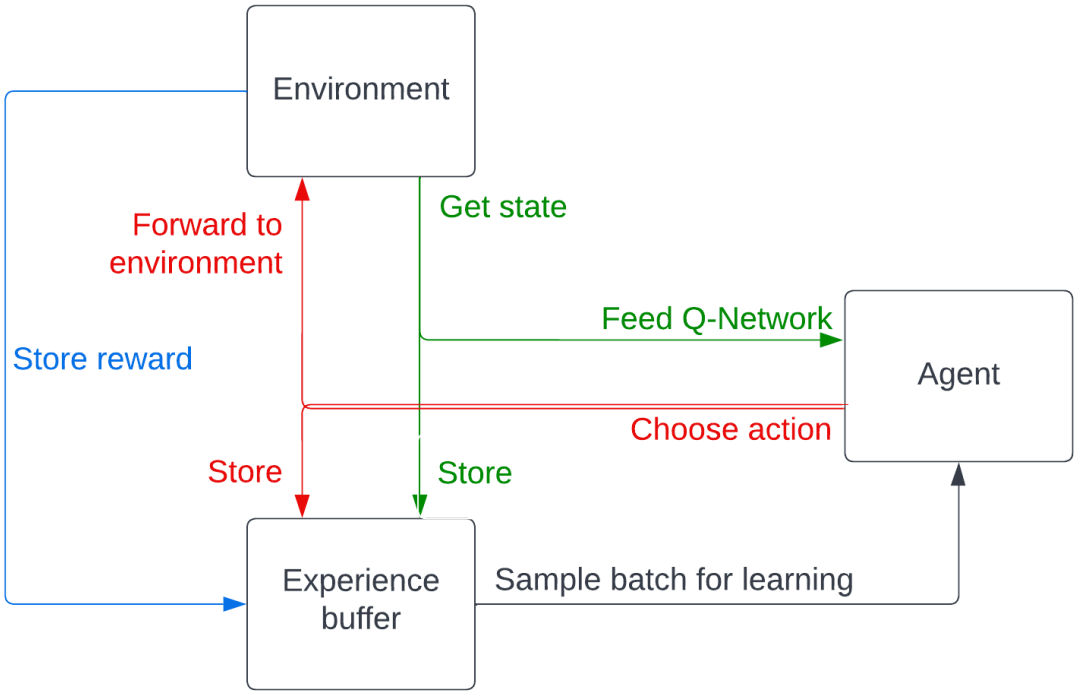


Figure D.1. DQN Runner diagram

E. Hypothesis tests

This appendix sums up some key concepts useful to understand hypothesis testing.

E.1. Null hypothesis

In hypothesis tests, there is always a null hypothesis H_0. With the test, we will try to reject the null hypothesis. If the test rejects H_0 with a sufficiently high level of significance, then we will be able to assume that the opposite of H_0 is true.

For example, if we have two data samples of which we want to know if they have been sampled from distributions that have equal means, (i.e we want to know if there is no significant difference between the mean values of the two samples), then we will be interested in testing

$$H_0 : \mu_1 = \mu_2 \tag{E.1}$$

where μ_1 and μ_2 are the means of the two samples. If we can reject H_0 with a level of significance of α (typically 5%) then the certainty that the two samples have different means is $1 - \alpha$.

E.2. p-value

In statistics, the p-value is the probability that, if H_0 is true, we obtain a result that is less probable than our actual result. Taking again our previous example: we had two samples with means μ_1 and μ_2. The p-value of this test is the probability that, if $\mu_1 = \mu_2$, we would get samples that contradicts H_0 even more than our two samples do. If a hypothesis test yields a p-value of p, it means we can reject H_0 with a certainty of $1 - p$.

F. RNN agents

F.1. DQN with RNN

F.1.1. Exploitation

As explained in chapter 3, the added value of RNNs is that (a part of) the hidden state is forwarded to the next time step. This creates a kind of memory throughout the time steps of a sequence of inputs.

At each time step in the environment, the agent selects its action using its Q-Network. Now, if the Q-Network is an RNN, the agent must keep in memory the hidden state of its Q-Network, so that it can be fed to the Q-Network (along with the observation of the environment) at the next time step, when the agent will select the next action. This deep recurrent Q-Network exploitation architecture is shown on figure F.1.

F.1.2. Learning

When it comes to the learning, things get a bit more complicated. In addition to what a classical DQN agent has to learn, a DRQN (Deep Recurrent Q-Network) agent also has to learn how to interpret the hidden state from the previous time step, and what information to put in the hidden state for the next time step. In order to do that, we cannot learn from simple (S_t, A_t, R_t, S_{t+1}) tuples anymore. We have to add some more data in the transition tuples.

The method we used is to feed the entire sequence $S_1, S_2, ... S_t$ (we will denote it $S_{1 \to t}$) to the Q-Network. The Q-values that we keep are only the output of the last element in the sequence. This is a many-to-one RNN architecture, which is the same thing we used in the encoders. So, for the learning, the $Q(S_t)$ is the last output of the DRQN when fed with $S_{1 \to t}$, and the $Q(S_{t+1})$ is still the output of the Q-Network when fed with S_{t+1}, but computed using the last hidden state of the Q-Network after the processing of $S_{1 \to t}$. This training architecture is shown on figure F.2.

In summary, for the exploitation, the Q-Network is used in a one-to-one architecture, and the hidden state is stored for the next time step. For the learning, the Q-Network is used in a many-to-one architecture to determine $Q(S_t)$ and in a one-to-one architecture to determine the $Q(S_{t+1})$, but with the hidden state of the $Q(S_t)$ processing.

F.1.3. Storing the transitions

During the training, the agent performs in the environment, and the transitions are stored in the experience buffer. As already explained, for DRQN we need to store the whole $S_{1 \to t}$ sequence of states in each transition. However, this could take a lot of memory. Our solution to that is to store the whole sequence until the terminal state only once per episode. Then, each transition refers to the correct sequence, that is the sequence corresponding to the episode that the transition is a part of. Then, in each transition, we only save the index t that refers to the position of the transition in the episode. From that we can rebuild the $S_{1 \to t}$ sequence for each transition without saving it explicitly in each transition tuple.

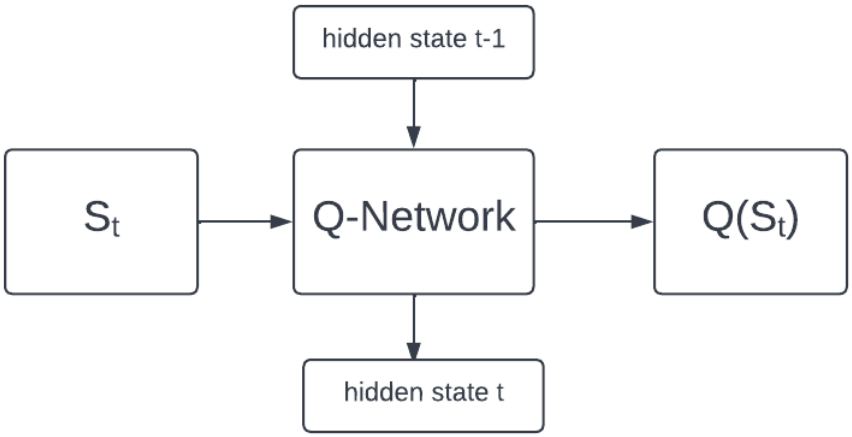


Figure F.1. DRQN exploitation architecture

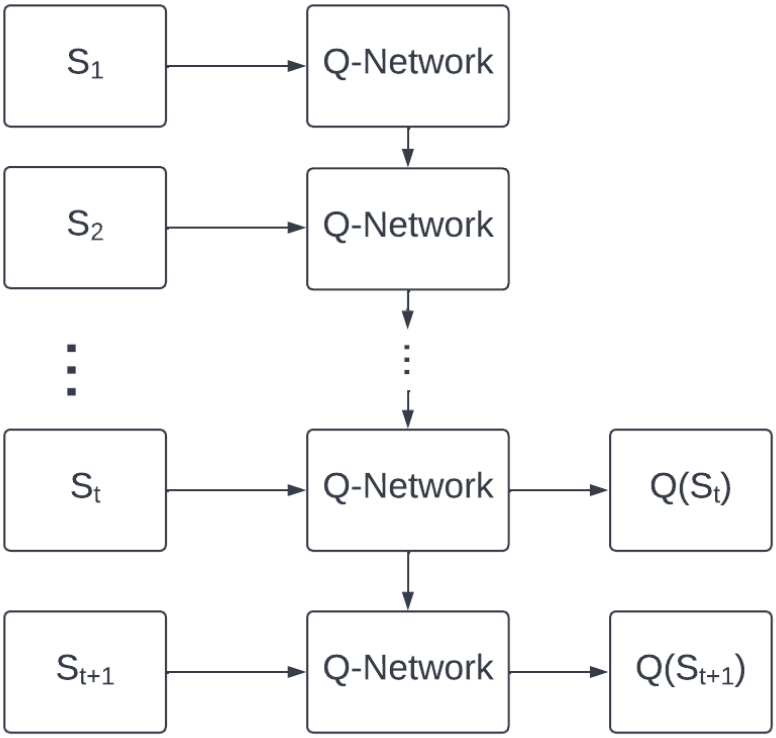


Figure F.2. DRQN training architecture

Bibliography

[AA22a] Ava Soleimany Alexander Amini. 6s191 - introduction to deep learning. http://introtodeeplearning.com, 2022.

[AA22b] Afshine Amidi and Shervine Amidi. Cs 230 - recurrent neural networks cheatsheet. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks, Apr 2022. [Online; accessed 6. Apr. 2022].

[ale22] alexlenail. Nn-svg. https://github.com/alexlenail/NN-SVG, Mar 2022. [Online; accessed 19. Mar. 2022].

[Bae21] Baeldung. Epsilon-Greedy Q-learning. https://www.baeldung.com/cs/epsilon-greedy-q-learning, Jan 2021. [Online; accessed 16. Mar. 2022].

[BKG20] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. https://arxiv.org/abs/2003.05991, 2020.

[BOE20] Cdt Ir Koen BOECKX. *Developing Strategies with Reinforcement Learning*. PhD thesis, Royal Military Academy, 2020. [GitHub repository: https://github.com/koenboeckx/VKHO].

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv*, December 2014. https://arxiv.org/abs/1412.3555.

[Che18] Guillaume Chevalier. Larnn: Linear attention recurrent neural network. *arXiv*, Aug 2018. https://arxiv.org/abs/1808.05578v1.

[CRG19] D. Corriveau, C.A. Rabbath, and A. Goudreau. Effect of the firing position on aiming error and probability of hit. *Defence Technology*, 15(5):713–720, 2019. https://www.sciencedirect.com/science/article/pii/S2214914719301965.

[Dis20] Dishashree26. Activation functions | fundamentals of deep learning. https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them, Jul 2020. [Online; accessed 16. Mar. 2022].

[HDL16] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016. https://arxiv.org/abs/1609.09106.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HS15a] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. https://arxiv.org/abs/1507.06527, 2015.

[HS15b] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015. https://www.aaai.org/ocs/index.php/FSS/FSS15/paper/viewPaper/11673.

[Kar22] Andrej Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness, March 2022. [Online; accessed 9. May 2022].

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv*, Dec 2014. https://arxiv.org/abs/1412.6980.

[KW52] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952. https://www.jstor.org/stable/2236690.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv*, December 2013. https://arxiv.org/abs/1312.5602.

[Muc21] Eric Muccino. LSTM vs GRU: Experimental Comparison - Mindboard - Medium. *Medium*, December 2021.

[Mus22] Musstafa. Optimizers in deep learning - mlearning.ai - medium. *Medium*, Feb 2022. https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0.

[Phi20] Michael Phi. Illustrated Guide to LSTMs and GRUs: A step by step explanation. *Medium*, June 2020. https://towardsdatascience.com/illustrated-guide-to-lstms-and-grus-a-step-by-step-explanation-44e9eb85bf21.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999. https://www.sciencedirect.com/science/article/abs/pii/S0893608098001166.

[Rem22] Sp0utn1k (Maximilien Remacle). GitHub repository, thesis. https://github.com/Sp0utn1k/thesis/tree/main, May 2022.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation, parallel distributed processing, vol. 1. *Foundations. MIT Press, Cambridge*, 1986. https://apps.dtic.mil/sti/citations/ADA164453.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951. https://www.jstor.org/stable/2236626.

[RSS+18] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4295–4304. PMLR, 10–15 Jul 2018. https://proceedings.mlr.press/v80/rashid18a.html.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016. https://arxiv.org/abs/1609.04747.

[SLG+17] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-Decomposition Networks For Cooperative Multi-Agent Learning. *arXiv*, June 2017. https://arxiv.org/abs/1706.05296.

[Tan93] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *In Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337. Morgan Kaufmann, 1993.

[Var14] Prof James K Varkey. Chance of hit. Dec 2014. https://cupdf.com/document/chance-of-hitpptx.html.

[vHGS15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. https://arxiv.org/abs/1509.06461, 2015.

[Wat89] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.