

École polytechnique de Louvain

Reinforcement learning-based AI to play first-person video games

Authors: **Damien VANECKER, Quentin DE GRAEVE**
Supervisor: **Pierre SCHAUS**
Readers: **Siegfried NIJSSEN, Vianney COPPÉ**
Academic year 2020–2021
Master [120] in Computer Science,
Master [120] in Computer Science and Engineering

Abstract

In our current world, artificial intelligence and machine learning is constantly getting more and more importance in a variety of domains, in order to get autonomous vehicles or smarter programs where the human intelligence is not sufficient.

One domain of interest is reinforcement learning, which aims to get a program to discover and learn his environment, in order to accomplish a given task without direct supervision from a human.

This master thesis explores reinforcement learning and more specifically, Q-learning. The goal will be to create an agent which is going to learn to play video games autonomously. We believe that using games can be a good playground to train programs safely before taking them to the real world to perform more useful tasks, such as exploring difficult terrain or piloting self-driving cars. The paper will present in detail how the algorithms work and which techniques can make it perform better.

Our code is accessible online at the github repository:

https://github.com/dVaneberck/Master_Thesis

Contents

Nomenclature	iii
1 Introduction	1
2 Theory	3
2.1 Reinforcement Learning	3
2.1.1 Markov Decision Process	4
2.1.2 Q-learning	7
2.1.3 Reinforcement learning	11
2.2 Neural Network and Deep Learning	12
2.2.1 Different approaches for the inputs	12
2.2.2 Type of layers	14
2.2.3 Learning using backpropagation	19
2.3 Additions for stability	25
2.3.1 Decaying Epsilon-Greedy Policy	25
2.3.2 Double Deep Q Networks (DDQN)	26
2.3.3 Replay Memory	28
2.3.4 Prioritized Experience Replay	29
3 Architecture	32
3.0.1 Architecture of the networks	32
3.0.2 Transcribing the theory in pseudo-code	33
3.0.3 Architecture of the code	34
4 Applications and results	40
4.1 Cartpole	40
4.1.1 Environment	40
4.1.2 Adaptations to the architecture	41
4.1.3 Results	42
4.2 Super Mario Bros	48

4.2.1	Environment	48
4.2.2	Adaptations to the architecture	48
4.2.3	Results	50
4.3	MineRL (Minecraft)	53
4.3.1	Environment	54
4.3.2	Solution using images	54
4.3.3	Solution using numerical metadata	58
4.3.4	Approaches using a combination of different inputs	59
5	To go further	63
6	Conclusion	65
	Bibliography	67

Nomenclature

AI: Artificial Intelligence

ML: Machine Learning

RL: Reinforcement learning

MDP: Markov Decision Process

DQN : Deep Q-Network

DDQN : Double deep Q-Network

PER : Prioritized Experience Replay

ANN: Artificial Neural Network

MLP: Multi-layer perceptron

TD: temporal difference

Chapter 1

Introduction

Machine learning has been constantly getting more and more attention in our world, in order to extract and analyze interesting data, build models for prediction or assist humans in biology. Those are the most scientific applications of it. However, one subject that the mainstream media seems to be concerned with, is machines actually learning by themselves about their world. In this thesis we will study one aspect of this field, using an approach called reinforcement learning.

Reinforcement learning is one of the 3 main areas of machine learning (the other two being supervised and unsupervised learning). It aims to make an agent learn from his environment, without any previous knowledge. In that sense, it can indeed seem like it is a program that gets smarter with time and adapts to its surroundings.

In comparison, the other fields of ML rely on having a dataset over which a model can be fitted. Being free of the need to collect such data beforehand can be profitable in a variety of situations. It also seems more alike to how humans or animals would learn a new skill, by performing actions first randomly, then with time, only keep on doing the actions that worked. But this doesn't mean that this approach will give us a smart program that can solve any problem. Actually, the agent only tries to achieve a predetermined goal, and will get there by replaying the best actions it has discovered so far.

Nevertheless, it is a field of interest that deserves to be studied and could have many applications in the real world. It can be used in autonomous movement, to train self-driving cars before they hit the road, or to help the navigation in complex environment. Robots could learn to perform some tasks in the same way as humans do. It can also have applications in optimization. As a proof of its potential, we can cite AlphaGo that relies partly on RL in order to achieve the best results. In fact, the algorithm was able to beat all top players of the game. [1]

This is why this thesis will have as final aim to develop an algorithm that is able to play a first-person view video game. Indeed, a simulated environment or a real one are not necessarily that different, and it is a nice playground to train an agent without any harm done to the real world.

During the year where we worked on this thesis, we focused on environments of increasing difficulty: Balancing a pole on a cart, playing the classical Super Mario Bros. game, and then solving small tasks on the Minecraft game. We will detail later in this report how we solved each one, which approaches and variations we considered and how we implemented them. However before explaining the practical part of our work, we will give a theoretical explanation about Reinforcement learning and the techniques we can use in complement to increase its efficiency. After this theoretical part, we will present how this theory was translated into code and finally presenting the different environments.

Chapter 2

Theory

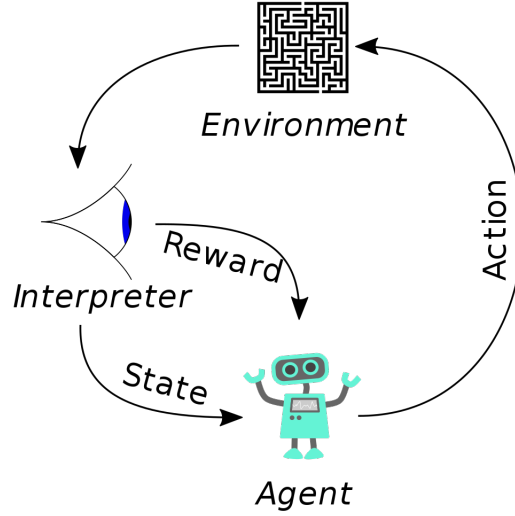
To start this explanation, here is a general explanation of how RL works. In the next sections, we will explain different variations we use compared to the simpler versions of RL, and how we modified the algorithms to perform better.

2.1 Reinforcement Learning

Reinforcement Learning is a technique that makes an agent learn an optimal policy to perform a task. In this approach, we represent the agent evolving in a discrete-timed environment. At each time step, he must choose an action to do. Each one will give it a feedback, in the form of a positive or negative reward.

Basically, the agent observes his environment as a state, and needs to choose an action to perform on the basis of his observation to transition to a new state. The transition being done generates a reward, that it needs to consider to modify its policy over time. The goal of the agent is to maximize the cumulative reward over its lifetime, as it should make him closer to realizing the wanted task.

This formulation using states, actions and rewards takes its basis on Markov Decision Process (MDP). In the next sections, we will explain the mathematical aspects on which reinforcement learning takes its origin. MDP, while slightly different, can show some similarity with dynamic programming too.



2.1.1 Markov Decision Process

Markov Decision Processes are based on Markov chains. Those are a stochastic model used for representing states and events that makes the transition between them. Each event is associated with a probability, which depends only on the current state.

A MDP is an extension of the Markov chains, which adds the concept of actions that an agent can take, and rewards for them. The MDP is a control process used in decision making. They are stochastic, so they are useful in situations where randomness is involved. Its goal is to find a policy function that returns the action a to take when it is in state s . Finding the optimal policy means to maximize the cumulative reward. [2]

Some notations:

- S : the set of states.
- S_t : the state of the environment at a specific time step t .
- s : a specific state.
- A : the set of actions.
- A_t : the action taken at a specific time step t .
- a : a specific action.
- R_t : immediate reward at a specific time step t .

- $r(s'|s, a)$: a specific reward (immediate) received during the transition from a state s to the next state s' using action a .
- G_t : cumulative reward over the life time.
- $p(s'|s, a)$: the probability to transition from a state s to a next state s' using action a . Indeed, here we make the assumption that choosing the same action in a specific state does not always lead to the same next state, due to randomness and unknown events. This is called the dynamics function of the environment. If choosing an action a in state s always lead to the same next state, then the process is deterministic and the probability is always 1.
- γ : discounting rate, with $0 \leq \gamma \leq 1$.
- π : a policy function that dictates which action to take for every state. It maps from state space to action space.
- $v_\pi(s)$: value of state s following policy π . This is the expected sum of rewards, when following the policy, starting at state s until the destination.
- $v_*(s)$: the value of state s under the optimal policy.

In a MDP, the Markov property needs to be always valid: the probability to get to a state or to receive a reward depends on the preceding state and the preceding action only.

We want to maximize the return, or cumulative reward G_t :

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.1}$$

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

γ , the discounting rate modifies the importance of future rewards in the cumulative return. The higher this value is, the more importance will be given to the future compared to immediate rewards. When it is close to 1, this can lead to more unstable results, as a lot of future states needs to be considered. Setting a γ smaller than 1 is also useful for continuous scenarios, as there is no particular destination state. In some simpler cases [3], the return can be a simple sum of immediate rewards (case of $\gamma = 1$). This weighted sum is what gives MDP's an ability to perform well in the long term, as it can take into account that a future reward will be more important than the immediate reward.

The value function v_π gives the expected cumulative reward of its input state s , received by following the policy starting from that state, until reaching the

destination. The symbol $\mathbb{E}_\pi[\dots]$ means that the expectation of the random variable is considered when following a policy π . t refers to a known time step. For every destination/goal state (denoted as d), we have: $v_\pi(d) = 0$, because if the process is already in the goal state, it does not lead to any more rewards. For all other states:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi [G_t | S_t = s] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] && \text{(using (2.1))} \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma G_{t+1} | S_t = s] && \text{(using recursion)} \end{aligned}$$

Given that the policy returns only a single action to take in each state:

$$\begin{aligned} v_\pi(s) &= \sum_{s'} p(s'|s, a) \left[r(s'|s, a) + \gamma \mathbb{E}_\pi [G_{t+1} | S_{t+1} = s'] \right] \\ &= \sum_{s'} p(s'|s, a) \left[r(s'|s, a) + \gamma v_\pi(s') \right] \end{aligned} \tag{2.3}$$

This last result is called the Bellman equation and is very important for reinforcement learning, it will often be used for the rest of this thesis. It shows how the value of the current state is influenced by the probability to get to a new state, the reward associated by this transition and the (discounted) value of the new state.

This equation leads to the value-iteration algorithm, where for a specific iteration, we compute the value v_π for all states using the values v_π obtained during the previous iteration.

The optimal policy π_* tells us which is the best action to take in state s . It is also associated with the optimal value function for the state, that is, the one that maximize the value of that state, where using (2.3) gives:

$$\begin{aligned} v_*(s) &= \max_{\pi} v_\pi(s) \\ &= \max_{a \in A(s)} \sum_{s'} p(s'|s, a) \left[r(s'|s, a) + \gamma v_*(s') \right] \end{aligned} \tag{2.4}$$

This is known as the Bellman optimality equation. It provides the expected return for the best action in that state.

The equation to find the optimal policy in state s is:

$$\pi_*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ \sum_{s'} p(s'|s, a) (r(s'|s, a) + \gamma v_\pi(s')) \right\}$$

2.1.2 Q-learning

In many cases in a Markov decision process the dynamics of the environment are not known, neither are the rewards and the following states, and they can't be predicted. This will prevent us from solving the process using the Bellman optimality equation.

This is where Q-learning gets useful. Since rewards and dynamics of the environment are assumed to be previously unknown, agents using this algorithm are able to learn by experiencing the consequence of their actions. It is an algorithm of reinforcement learning that is said to be model-free, as it does not need to have an internal representation of the environment's model. The agent only need to experience the information given to him to be able to learn. [4]

Here we introduce a new value q_π called the Q-value, hence the name of the algorithm. We also call it the action/value function. It represent the value of taking an action a in state s , then relying on policy π . Again, the Q-value of a terminal state (a goal) is 0. It is defined such that the value of a state is always the best Q-value possible in this state:

$$v_\pi(s) = \underset{a \in A(s)}{\max} q_{\pi_*}(s, a) \quad (2.5)$$

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi [G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_{s'} p(s'|s, a) [r(s'|s, a) + \gamma v_\pi(s')] \end{aligned} \quad (2.6)$$

The optimal Q-value for this process is computed as the maximal Q-value possible, obtained using a specific policy π . It gives the expected immediate return when taking this action a , and then the value of relying on the best policy in the future :

$$\begin{aligned}
q_*(s, a) &= \max_{\pi} q_{\pi}(s, a) \\
&= \mathbb{E} [R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a \right] \quad (\text{using (2.5)}) \\
&= \sum_{s'} p(s' | s, a) \left[r(s' | s, a) + \gamma \max_{a'} q_*(s', a') \right] \quad (2.7)
\end{aligned}$$

These last equations are the Bellman optimality equation for the Q-values. It gives the long-term expected return for every state-action pair. By comparing the values known for the next state s' , it is easy to compute a value for this state/action pair (s, a) , and then select the optimal action to be taken now.

As the Q-values for possible successor states s' are known, there is no need to have more information about s' , the rewards associated with it or the environment's dynamics in order to choose an action. But if it is not possible to access these informations for computing the Q-values, it should be possible to obtain them in another way. This is done iteratively with a value-iteration update, where the Bellman equations are applied multiple times. It is here that Q-learning learns-by-doing.

$$Q^{new}(S_t, A_t) = Q(S_t, A_t) + \alpha(t) \left[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right] \quad (2.8)$$

This equation describes a fixed point iteration, where Q tries to approximate q_* . Initially the Q-values can be fixed arbitrarily, then they will become better approximations in time. The original Q-Learning paper [4] proves that if enough exploration time is given, and all actions/pairs keep on being updated, the Q-values will converge to their true values. This update equation can be used at every time step.

At each iteration, the new Q-values are computed as a weighted average (considering the learning rate $\alpha(t)$) between the old Q-value and a new factor. This new factor is called the *temporal difference*, as it is the difference between a target: $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$, and the old estimate: $Q(S_t, A_t)$. We can observe that the temporal difference can be understood like some sort of error, as it is the difference between an old estimate of $Q(S_t, A_t)$ and a new estimate of the same quantity, computed using the Bellman equations. It is an important concept, also called TD error, and will be used again in Section 2.2.3 to compute the loss function.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

(a) A maze with every tile/state numbered

	North	South	West	East
1	0	0	0	0
2	0	0	0	0
...				
41	0	0	0	0
42	0	0	0	0
43	0	0	0	0
...				
64	0	0	0	0

(b) Table before learning

Figure 2.1: The maze problem

The learning rate α ranges from 0 to 1, and should decrease gradually with time, in order to fix the values later in time, while still being able to learn quickly at first.

Q-Learning uses a table Q , with as indexes the states and actions, it used to store Q-values. Values are updated in the table when an action is experienced on a state.

Example of Q-Learning

To illustrate this, we can imagine the scenario where a robot should navigate and reach the end of a maze (Figure 2.1). In this maze, each tile can be numbered, and will be considered as a state. The set of actions A for most states is: $\{north, south, west, east\}$, except when a border prevents one of the directions. It starts in state 5, and receive a reward of +10 for exiting the maze at the bottom (choosing *south* in state 58). On the other side, it receives a penalty of -5 when it walks on a red tile. We will also use a discounting factor γ of 0.9.

This environment is deterministic, when the agent chooses *west*, it will always end up on the left tile. This means that $p(s'|s, a)$ will always be 1. The optimality equation (2.7) for a deterministic case is one that is often seen, and simplifies as:

$$q_*(s, a) = r(s'|s, a) + \gamma \max_{a' \in A(s')} q_*(s', a') \quad (2.9)$$

At first, the table is filled with zeroes for every action/state pair, because the agent

	North	South	West	East
3	-	0	0	0
4	-	-5	0	0
5	-	-5	0	0
6	-	-5	0	0
7	-	-5	0	0
11	0	0	0	-5
12	0	0	0	-5
13	0	0	0	-5

(a) Table early in the learning

	North	South	West	East
5	-	-1.515	2.82	1.85
41	3.871	2.29	-	4.78
42	-1.515	3.1	4.302	5.31
43	-0.21	2.29	4.77	5.90
52	5.90	7.29	2.29	5.90
57	2.29	-	-	9
58	3.1	10	8.1	8.1
59	2.29	-	9	7.29

(b) Table after learning

Figure 2.2: Q-table for the maze problem

has not yet discovered the values associated with its actions. During learning, it will apply the Q-learning update equation (2.8) to iteratively update its approximations of the Q-value.

In Figure 2.2, on the left, we show the estimations for Q-values early in the learning, with a learning rate α simplified for this example to 1. The table directly reflects the negative impact of the red tiles, but there is no indication of which direction is most profitable to reach the goal. This is due to the rewards being relatively sparse: most states do not generate any reward. Before having a path to the exit, the agent will need to walk randomly until there.

With this example, we can already show why having some dedicated exploration time is necessary: the first path reaching the goal gets positive values, and if the agent only follows the best action discovered yet, it will always follow that path. To learn properly, it should try to experience all actions in all states too. This will be explained more in Section 2.3.1.

On the right of Figure 2.2, we show the end of learning, when the values should converge towards the Bellman optimality equations (2.9). At this point we can see that it actively avoid the red tiles and follow the shortest path. Again, this supposes that the agent has got a complete experience of the maze.

With this example, we can see that Q-Learning can be compared to dynamic programming in the sense that a table is used to compute values, and they hold recursive relationships between them. The difference is that in Q-Learning, a fixed-point algorithm is used to solve the process. Indeed, without knowing probabilities

to transition from one state to another, and without knowledge about the rewards, it would not be possible to fill in the table, so we need to compute successive approximations until convergence. At the opposite, in a MDP, the environment's dynamics are known, and a more simple dynamic programming algorithm can be used to get all entries of the table (the values V of states this time).

2.1.3 Reinforcement learning

For the rest of this thesis, we won't be using pure Q-Learning either. Computing the Q-values for all state-action pairs is not always possible in large problems. In the last example it was still easy to do since the state space was pretty small, but it still needs a row for each state. In many applications, there are often too many states. In a 3-dimensional world, the states also change based on the angle of vision, so the state space can quickly grow towards infinity. In this case, it is not feasible to store and compute every value, it would require too many resources. Also, most states/actions pair are not visited often, which decrease the interest of storing all values. The table will be replaced by a function approximator: a neural network. It will be trained to predict the Q-values. This approach is referred to as Deep Q-Network (DQN), more details on those are explained in section 2.3.2.

One other advantage of the neural network is that similar states will gives similar outputs, which can speed up the process as all the similar states do not need to be visited before having a approximated idea of their results. This brings disadvantages too, as neural networks need to be trained, which can be a problematic task in itself.

Unrelated to neural nets, some other problems can happen. The Markov property does not completely applies for some use cases, as the algorithm could need some sort of memory to understand and remember its environment correctly, so the current state and reward would not be sufficient to make the best guess concerning the action to take. But even if the states do not satisfy the Markov property, it is still useful to see them as having an approximation of it. The closer we can get to have a Markov state, the better the result will be. In some cases, it is possible that having a non-Markov state representation could be beneficial, for instance representing the environment with less details could speed up learning, without losing too much information.

The next sections will differentiate themselves more from the pure mathematical theory seen previously, as we will explain how we created our model of reinforcement learning in practice, with the different additions to help solve the problems.

2.2 Neural Network and Deep Learning

An artificial neural network is used as a non-linear function approximator. As the name suggests, it is a network of connected nodes, organised in multiple layers. Each one of the nodes can take multiple inputs from a previous layer and it computes an output that is forwarded to a following layer. An ANN is trained to match the response expected when some data are presented at its input.

We use the network to approximate the Q-value of taking a specific action on one particular state, in order to choose which action is the better one. Using an ANN to compute the Q-values remove the fix-point algorithm used otherwise when doing simple Q-Learning. Using this technique to do reinforcement learning is called Deep Q-Learning/Deep Q network (DQN). More details about how to use neural networks with Q-Learning are presented in section 2.3.2.

A simple case of ANN are feedforward networks. In this case there is no loop of data, no layer will feed its output to a previous layer. In the opposite, when loops are present, the network is said to be recurrent, but those are more difficult to train.

2.2.1 Different approaches for the inputs

In classical Machine Learning, models can count on a set of data to build themselves. These sets, named training sets are data that have been collected before the creation of the model. This is one of the points where Reinforcement Learning is different from other more classical machine learning approaches.

In RL, it is the program that will create the input or experience that the model will use to build itself. In general, these inputs will be screens but if more classical data like numerical inputs are available, they can be also used. In this section, we will discuss the two approaches.

Screen based input

Like any other machine learning problem, dealing with input is a critical part. In the case of reinforcement learning, our common input is a screen or a batch of multiple screens. The program has to search and recognize the relevant data inside the screen without direct help from the programmer.

One issue is that simply feeding the screen to the program will not always yield good results. Searching pertinent data inside an image containing a lot of information could be related to one of the main problematic in machine learning, the curse of dimensionality.

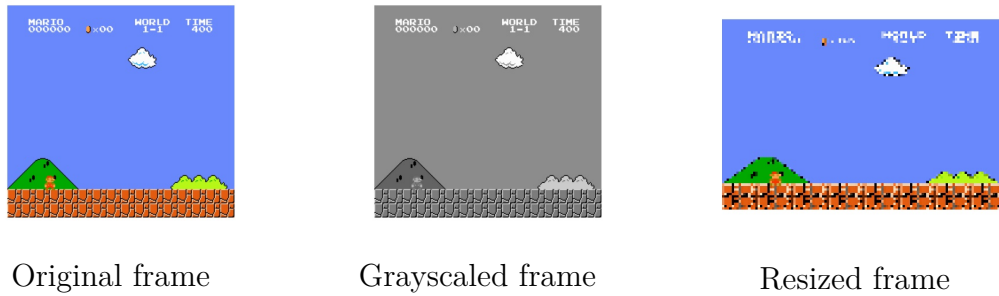


Figure 2.3: A example of different processing methods

The curse of dimensionality is a well know problem in ML where a large amount of data compared to a low amount of observation is detrimental to the model by making a generalization difficult. Since we could consider the pixels screen as the equivalent of features in a more classical machine learning problem, the large amount of pixels, and thus information, makes the creation of a model difficult. Unlike other ML problems, it's not possible to strictly perform feature selection since a specific pixel could be useless at one time but crucial at another.

We need to reduce the amount of noisy information in order to achieve better results. To do that, we can preprocess the image at hand before using it in the neural network. This task is heavily dependant of the problem nature and requires the programmer to analyse the screens that the program will receive. There is no real typical rule of thumb to preprocess an image, but there are some common methods, like reducing the resolution of the screen.

It can be counter-intuitive since our eyes prefer images with larger resolution to see more details but in the RL case, this additional information is mostly noisy and distracts the program with an abundance of information. Another method is to change the screen color by greyscaling them. Colors could also be noisy in some cases so reducing the amount of information by using levels of grey instead of colors helps. Cropping part of a screen can also be a way to have better results by removing useless parts.

To improve the amount of useful information, we can use a sequence of screens instead of just one. The principle behind this idea is that we can deduce information by using a batch of consecutive screens like the speed, acceleration or direction of an object in the screen.

Data based input

Even if screens are the most typical type of input used in RL, it is also possible to use simpler data like numerical data types. The main advantages of using those is that the program doesn't have to detect features with a risk of error, and their size is usually smaller. Those data are ready to be put inside the neural network and they give better results in less time. However, this approach is not possible everywhere. For some problems, it's not possible to have access to them. We will demonstrate the efficiency difference of these both approaches in the Applications and results section, chapter 4.

2.2.2 Type of layers

To build an effective neural network, it's interesting to use several type of layers in order to retrieve different parts of the information. In this section, we will discuss important layers in reinforcement learning, and other necessary layers we will need to use in our own implementation.

Layer of non-linear units / Fully connected

It is a simple form of layer that can perform a non-linear regression over the data points presented. It is better than using a simpler linear model, as they would limit the range of functions that can be approximated.

A non-linear unit can be done in 2 steps: first taking a linear combination of the inputs, then applying a non-linear function on the result. We will use matrix form to represent the equations:

$$y = \sigma(w^T x)$$

with

- x : a input vector
- w : a weight vector (of the same size as the input)
- σ : a non-linear function
- y : the output vector

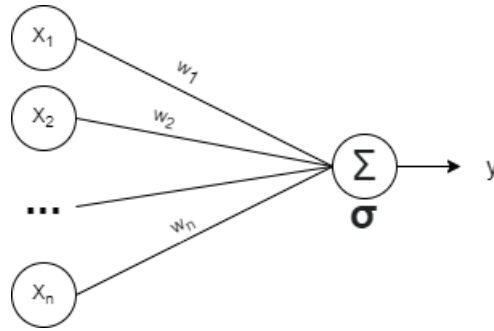


Figure 2.4: A single node performing a non-linear function

The non-linear function is called the activation function and needs some important characteristic to be used inside a neural network. Since neural networks use the gradient descent algorithm to perform its training, it is necessary to be attentive to the gradient vanishing problem.

This problem is caused during the backpropagation phase (more details about this will be given in section 2.2.3) where each weights of the networks are updated to reduce the loss, or error of the network. These updates can lead in some circumstances to reduce the gradients of some nodes so much that they will vanish to 0. So the activation function should not shift the gradient to zero.

Other important concepts to pay attention to are the computational cost and the differentiability. The gradient descent requires each of its layers to be differentiable to be able to perform the backpropagation. Thus the activation function must be differentiable too. The computational cost is also an important point to take into account. Since each activation function will be calculated a large number of times which can be in millions of iterations, the function must be cheap to compute.

Here some popular activation function used :

- sigmoids/ hyperbolic tangent: smooth and continuous, limits the output range to $[-1; 1]$, or logistic function between $[0, 1]$. However this function is subject to the vanishing gradient problem and thus is not usually used in reinforcement learning.
- threshold: discontinuous in 0, binary output values -1 or 1.
- RELU: has been proven to ease the training of deep networks. Its function is $y = \max(0, x)$. This function doesn't suffer from the vanishing gradient problem but since all the negative value are put to zero, some node can die with their gradient not updating anymore, and thus learn nothing. This

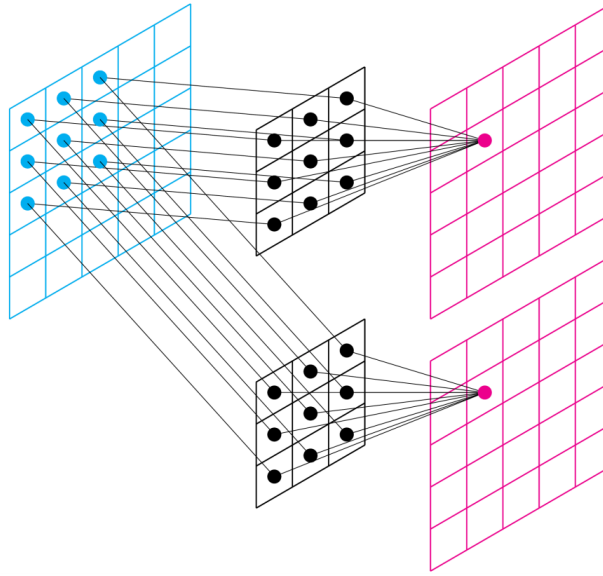


Figure 2.5: A convolution layer with 2 channels. There is one image and 2 kernel filters, resulting in 2 feature maps [5]

problem is named "dying Relu" but it has been corrected with more advanced Relu function like Leaky Relu.

In PyTorch, a non-linear unit can be done using a Linear layer, then any of the activation layers.

When multiple layers are used, it is also called a multilayer perceptron. In this case, at least one intermediate (hidden) layer is used before the output layer. It has multiples nodes in the same layer, and it can also have multiple inputs or outputs. It is said to be dense if the node takes an input from all nodes all the previous layer.

We use it to process numerical data.

Convolutional layer

A more advanced form of layer used over high dimensional data such as matrices of pixels. As the name suggests, here each neuron performs a convolution between an area of the image - called the receptive field - with a kernel filter.

The convolution (represented in Figure 2.5) is a local operation: only the receptive field is considered during the operation, and this produces one output pixel. By sliding the kernel filter over the input image, a new image is created at the output:

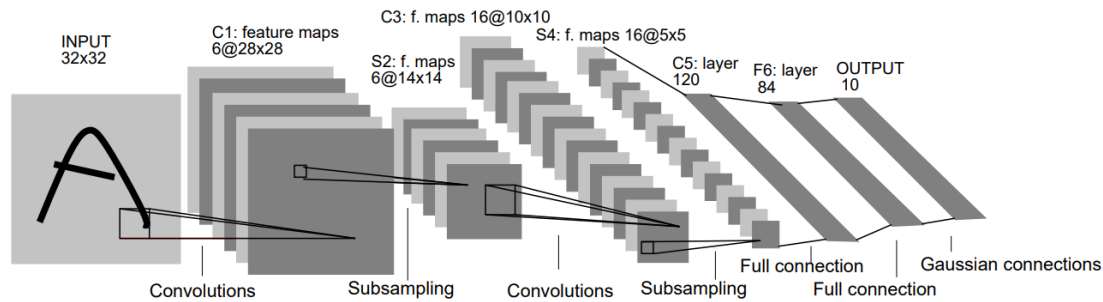


Figure 2.6: Example of a convolutional network: LeNet [6]. It takes a 32x32 pixels images representing handwritten digits, and outputs which number is most likely to be represented.

the feature map. The meta-parameters in this layer are: the kernel size (the size of the receptive field), the stride (how much translation is done on the input image between convolutions), and how many channels (kernel filters) are used.

Here, it is the kernel filter that is trained. Each of its pixel is a trainable weight. Overall, this layer uses a lot less training parameters (weights) than if we were to use a fully connected layer on each pixel of the image. Indeed, a single filter is used over the complete image, and it has a small size compared to the original image (typically a square of 3x3 to 7x7). Also, the weights of the kernel are shared between the neurons of the filter, but the set of neurons in another channel have different weights. In the end, a convolutional layer will use only $kernel_size^2 \cdot \#channels$ weights. This uses less memory and makes the training easier.

Convolutions are used as patterns detector. Each filter is trained to detect particular shapes. The layers can represent an increasing level of abstractions. The first layer can recognise basic shapes such as lines and edges. The second will represent collections of simple shapes, more complex shapes. The third layer gives more abstract features, and this goes on until very complex features such as faces, objects etc can be recognized. [7] [8].

Following the convolutional layer is traditionally some fully connected layers, as they can select or perform a function between all the features recognised through the image, and combine them in the output format that is needed. An example is given in Figure 2.6.

Flatten

Commonly use as a transition between a convolutional layer and a fully connected layer. It's role is to transform the multidimensional input (multiple squared feature maps) into a large unidimensional output, ready for a fully connected layer taking a 1D array as input.

Softmax

The Softmax layer is a layer which will turn the input into a vector where the sum of the elements is equal to 1. This is useful since it's possible to interpret the value of each element as a probability. A softmax layer can be interesting to put as last layer to have a easily understandable representation of the neural network output. For instance, LeNet architecture use this layer at the end to attribute a probability for each category of possible character.

Sub-Sampling

Sub-Sampling is a kind of layer used inside a convolutional neural network. These layers are used to decrease the size of an image by different techniques. The goal is to reduce the possible noisy information inside the image and avoiding overfitting. There are several techniques possible to perform the sub-sampling but the most popular ones are the maximum pooling and the average pooling.

The idea of each pooling is to take a sub-sample of the original image and perform an operation on it. In the case of the maximum pooling, the pixel with maximal value will be kept, and the other ones will be discarded. Average pooling will average the value of the present pixels in the sub-sampled image.

The maximum pooling technique is the most used by the reinforcement learning community [9]. However, it doesn't mean it's the best technique. Like many others parts in reinforcement learning, the best techniques to use is a matter of design choice depending on the input and it's up to the programmer to find it.

Other types of layers

There is a lot of different kind of layers which can be used inside a neural network. Since there is no point in explaining them all in detail, here is a list of some interesting ones that could be used in RL:

- Dropout: Used to avoid overfitting. It "drops out" a small proportion of links between nodes of two layers.

- LSTM: A recurrent type of layer. Remember parts of the history, that will be kept in 'memory'. It could be useful for RL, but it is more difficult to train.

2.2.3 Learning using backpropagation

Once the data has been processed by the neural network, this one needs to adapt itself in order to produce better results in the future. In this section, we will explain how this is achieved by using backpropagation, where the neural network will be updated from the back to the front.[10]

Loss function

First it is needed to introduce what is the loss, as it is essential to the backpropagation algorithm. To put it simply, a loss function is a way to compute the error between values. That is why it is key in neural networks, in order to quantify how good the network is at predicting its target.

In a traditional neural network, the loss is computed as a function between the target values t_i (which are given in a training set) and predicted values y_i .

There are multiple ways to compute that loss. Each loss function can have its own advantages and disadvantages. Even if there is some popular and common choices among these function, it's up to the programmer to choose the best function depending on the application.

Mean squared error	$E = \frac{(t_i - y_i)^2}{n}$
Mean absolute error (L1 loss)	$E = \frac{ t_i - y_i }{n}$
Hubert loss	$E = \begin{cases} 0.5(t_i - y_i)^2, & \text{if } t_i - y_i < \delta \\ \delta (t_i - y_i - 0.5 \delta), & \text{otherwise} \end{cases}$

MSE and MAE are the most popular loss function used in Machine Learning when problem are solved by using regression and they can as well be used in Reinforcement Learning. We can note that MSE is subject to bias due to the presence of outliers. Since the difference is squared, an outlier (which in RL could be an overoptimistic estimation) will be given more importance. But it is often used in practice since it has an easy to obtain a derivative.

MAE on the other side doesn't have a continuous derivative in zero, but has the same slope over R^+ , such that any learning step will be the same size, regardless of how big is the loss. This can be a problem in late-stage learning, as a small loss leads to the same big learning step. Also, the MAE loss may be too forgiving with regards to outliers, focusing too much on the average of values.

The disadvantages of the two previous losses lead us to introduce the following one. The Huber loss is a more advanced loss which tends to combine the positive aspect of MSE and MAE. The smoother MAE part makes those losses less sensitive to outliers than the MSE, and it is continuous in zero thanks to the squared function. The downside is that the parameter δ needs to be trained which can take more time. This approach requires some design decisions. The more δ tends to zero, the more the Huber function will act like MAE and the more δ tends to ∞ , the more it will act like MSE. For instance, Pytorch uses the default value 1 for the parameter, and it seems to be a good compromise but tuning the parameter could still give better results.

In a traditional DQN, the loss function is based on the TD-error introduced in Equation 2.8, we reproduce the equation for the temporal difference error alone here:

$$TD - error = R_{t+1} + \underbrace{\gamma \max_{a'} Q(S_{t+1}, a')}_{t_i} - \underbrace{Q(S_t, A_t)}_{y_i} \quad (2.10)$$

This case is a little bit less straightforward than what we have seen, because we don't have actual, fixed training points t_i . Here the target points t_i are the Q-values of a state/action pair, computed by using the Bellman equation, as the sum of the immediate reward and the discounted value of the future actions. And the predicted values y_i are also the Q-values of the same state/action pair, but this time predicted directly by the network, without using Bellman's equation. When the 2 values of the temporal difference are known, they can be fed in any of the loss function.

This may seem confusing, but this loss function can be understood as the error of predicting the values y_i directly by the network, this error being only obtainable at the next time step, by experiencing a reward and the value of the next state (giving t_i).

It is also confusing for the algorithm itself, because it has been shown that the values t_i and y_i are obviously strongly linked together, as the 2 of them relies on the same network to predict the value of the current state, and the value of the

next state. We will see in section 2.3.2 that this leads to some problems that needs to be fixed.

Gradient descent

The gradient descent is a way to find the local minimum of a function f . The principle is to iteratively update the position of a point x , following the opposite direction of the biggest gradient at this point.

$$x(t+1) = x(t) - \alpha(t)\nabla f(x(t))$$

In that way, it is guaranteed that the position of the point will be lower on the function at each iteration. The process will reach a local minimum, but there is no guarantee that this minimum is global. The learning rate $\alpha(t)$ can be decreased over time to take progressively smaller steps, and stabilize the result.

Improving the network

Once a neural network is first initialized, the weights of the neurons are random and the predicted outputs do not fit the data. This is why it needs a phase of learning. To achieve this, the backpropagation algorithm is used. It is a supervised learning technique that aims at minimizing the loss of the network, using gradient descent. It gets its name because the errors computed at the end of the network are propagated back to its input. Along the way, the weights are updated accordingly to their contribution to the error. In this section, mathematical notations and terminology can vary depending on the reference used, we will focus on the one used in [5].

First we should compute activation and output values for each neuron. Those values are obtained during the *forward pass*. This phase happens when inputs are presented at the front of the network. At that time the calculation of the state of every unit can be done, layer by layer, until the output values are obtained. The activation of a neuron is the input of a unit, before the activation function. The output of a neuron is received after the non-linear function. We can note that in vector form a bias b is present after the weighted sum. In matrix form, we do not represent it, because it is included as the 0^{th} element of the weight vector, with a 1 added as the 0^{th} element at the start of the input vector.

$$\begin{aligned}
a_i^{(l)} &= \sum_j w_{ij}^{(l)} z_j^{(l-1)} + b_i^{(l)} && \text{(activation at layer l)} \\
\mathbf{a}^{(l)} &= \mathbf{W}^{(l)T} \mathbf{z}^{(l-1)} && \text{(in matrix form)} \\
\mathbf{z}^{(l)} &= \sigma(\mathbf{a}^{(l)}) && \text{(output of layer l)} \\
\mathbf{y} = \mathbf{z}^{(L)} &= \sigma_L(\mathbf{W}^{(L)} \sigma_{L-1}(\mathbf{W}^{(L-1)} \dots \sigma_1(\mathbf{x}))) && \text{(output of last layer)}
\end{aligned}$$

The activation of the first layer is simply the product of the weights of the first layer by the inputs of the network.

Then, to perform the backpropagation algorithm, it is needed to compute the error, or loss between the output values y_i predicted by the network (so the z_i at the last layer) and the target values t_i , which is the sum of the current reward and the best predicted value for the next state.

We want to minimize the loss to get the best performing network, in the sense that the target values are the closest to the values predicted. To reach that objective, the weights of every layer need to be updated. For this many rules exist, but we will focus on the stochastic gradient descent, where the weights are updated in the direction of the biggest gradient. The update rule is as follows:

$$w(t+1) = w(t) - \alpha(t) \left. \frac{\partial E}{\partial w} \right|_{w(t)} \quad (2.11)$$

It is an iterative process that gradually improves the values of w at time $t+1$, using its value of time t , updating it proportionally to how much those weights influence the error (and proportionally to a learning rate α). Now the algorithm only needs to have a way to compute the partial derivative of the loss, with respect to every weights.

In order to find those quantities, we need to use the chain rule and introduce new notations, with l being the current layer, i being the position of a neuron in this

layer and j that of a neuron in the next layer:

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^{(l)}} &= \frac{\partial E}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial w_{ij}^{(l)}} \\ &= \delta_i^{(l)} z^{(l-1)}\end{aligned}\tag{2.12}$$

with $\delta_i^{(l)} = \frac{\partial E}{\partial a_i^{(l)}}$

As z is already computed for each layer, it is still needed to compute δ_i at each layer. For the output units this is done easily:

$$\begin{aligned}\delta_i^{(l)} &= \frac{\partial E}{\partial a_i^{(l)}} = \frac{\partial E}{\partial y_i^{(l)}} \frac{\partial y_i^{(l)}}{\partial a_i^{(l)}} \\ &= \frac{\partial E}{\partial y_i^{(l)}} \sigma'(a_i^{(l)})\end{aligned}\tag{2.13}$$

The derivative of the loss function with regards to the output values is known, in the case of the MSE error it is $-2(t_i - y_i)$. And the derivative of the non-linear function can be known too.

Computing $\delta_i^{(l)}$ for hidden units inside the network is more difficult, but it has been shown that it relies in part over $\delta_k^{(l+1)}$ of the next layer, and $w_{ki}^{(l+1)}$, the weight between these two layers. This is why this phase is also called the *backward pass*, as the error terms are first computed at the output of the network and brought step by step back to the input layer:

$$\begin{aligned}\delta_i^{(l)} &= \frac{\partial E}{\partial a_i^{(l)}} = \sum_k \frac{\partial E}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_i^{(l)}} \\ &= \sum_k \left(\delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(a_i^{(l)})\end{aligned}\tag{2.14}$$

Once every derivative is computed, it is possible to apply the entire algorithm.

As a summary, we can synthesise the steps needed to train the network with this algorithm:

Algorithm 1 Backpropagation

- 1: Forward pass: apply an input x^k , remember activations a^k and neuron outputs z^k .
 - 2: Loss computation: between expected output t^k and network output y^k , to get δ_i^L in last layer.
 - 3: Backward pass: propagate error terms $\delta_i^{(l)}$ to get $\delta_i^{(l-1)}$
 - 4: Evaluate $\frac{\partial E}{\partial w_{ij}}$ using Equations 2.12, 2.13, 2.14
 - 5: Update weights using gradient descent in Equation 2.11
-

By applying this algorithm multiple times, with different inputs, the network will converge to predict correctly the value of data points presented to it.

However ensuring a convergence is not sufficient to have a good model. There is two points which still need to be considered in order to have an usable AI :

- Overfitting
- Slow learning

Overfitting is a well know problem in Machine Learning and unfortunately Reinforcement learning doesn't escape this. If the model overfits the data, it will not produce good results in real cases. Slow learning is the second problematic. Because of the large amount of computation, RL can be slow even for simple cases. This is visible when using images as input since the model can be lost in a image with a lot of information inside. Receiving data slowing in the course of time doesn't help. These two problematic cannot be directly addressed inside the neural network. However there are solutions, they are explained in the Additions for stability section.

2.3 Additions for stability

As explained before, Deep Q-Learning faces multiple problems that impact the learning performance. To correct this, this section will present several solutions which can be added to help the learning phase.

2.3.1 Decaying Epsilon-Greedy Policy

This is a small addition to help Q-learning explore more possibilities. Without it, the algorithm is more prone to always go for actions it already knows since it is uncertain of the results of other ones. Then the cumulative reward could stay stuck in a local maxima, while other better options exist.

The idea of this policy is to rely more on exploration in the earlier stage of training, and exploit the knowledge we have later in the training, as we can hope most actions have already been tried.

This is done by having an exploration probability, represented by an exponential decay function (Figure 2.7), with a parameter *epsilon_max* starting at a fixed value and going down towards its ending value *epsilon_min*. When an action needs to be chosen, we generate a random number between 0 and 1, if it lies below the curve, we will explore a random action.

$$\text{exploration_probability} = \text{eps_min} + (\text{eps_max} - \text{eps_min})e^{(-\text{eps_decay_time_step})}$$

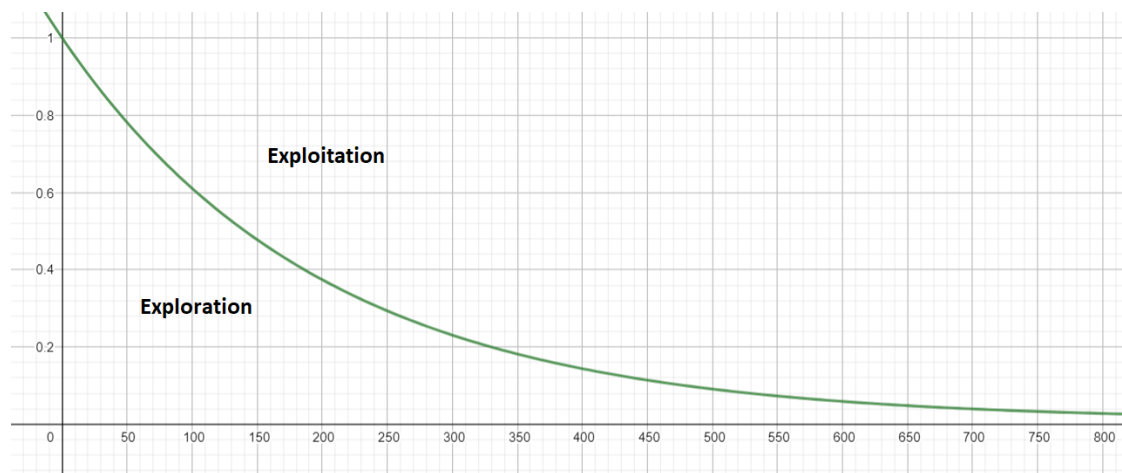


Figure 2.7: The exploration probability of the epsilon-greedy policy on the vertical axis, while the horizontal one shows the time steps. Parameters are: *epsilon_min* = 0.05, *epsilon_max* = 1.0, *epsilon_decay* = 0.05

We have always used it with the *epsilon_min* higher than 0, so that even very late in the training, if the policy is stuck and can't progress anymore, it has a chance to try new things, experience different rewards for different actions, and maybe escape a local maxima. If the policy learned was already good, as the probability is pretty low, it should not affect the result too much. If the exploration action was worse than what the agent already knows, the policy won't change.

2.3.2 Double Deep Q Networks (DDQN)

The Double Deep Q Networks or DDQN is an advanced technique largely used in the reinforcement learning community. This technique is based on several other algorithms. To have a better understanding of this concept, we will give a brief explanation of the techniques on which the DDQN was built.

Q-learning

At first, reinforcement learning was based on Q-learning, we already discussed it in 2.1.2. However, there are two important equations to keep in mind. The first one is the Q-learning update after taken an action a :

$$Q_{t+1}(S_t, a) = Q_t(S_t, a) + \sigma(Y_t^Q - Q_t(S_t, a)) \quad (2.15)$$

where S_t is the state at time t . σ is a scalar step size which is used as learning factor, whose range is between zero and one. This factor helps to decide how much the new information will impact the current network, equal to zero there will be no impact and equal to one the impact will be the most important. Q_t represent the network at time t . Y_t^Q is the target value and the second important equation:

$$Y_t^Q = R_{t+1} + \gamma \max_{a \in A} Q_t(S_{t+1}, a)$$

where R_{t+1} is the immediate reward at time $t+1$, γ is the discount factor that trades off the importance of immediate and later rewards and a is an action.

Deep Q-Network

Deep Q-Network takes the place of the Q-learning since this one can't deal with big numbers of continuous states and discrete actions [11]. Even by replacing the q-table by a neural network, DQN keep the same parameters update rule as in Equation 2.15. At the beginning, DQN used only one network to perform both the parameter update and the target value computation. However, as said in [12], the

use of two networks and memory replay greatly improves the capabilities of DQN. The idea of using two distinct networks for DQN was already present, but it's not yet similar to the Double Deep Q Learning.

Thus, the DQN algorithm used two different neural networks. The first one, the online network Q_{online} is used in the parameter updates and the second one, the target network Q_{target} is used in the target equations. The target network has the same shape than the online network but it is updated only after a certain number of steps, this number is arbitrary and is modified according to the task. The equations are thus slightly modified:

$$Q_{online_{t+1}}(S_t, a) = Q_{online_t}(S_t, a) + \sigma(Y_t^Q - Q_{online_t}(S_t, a)) \quad (2.16)$$

$$Y_t^Q = R_{t+1} + \gamma \max_{a \in A} Q_{target_t}(S_{t+1}, a)$$

DQN is used in reinforcement learning to solve task like image recognition but since DQN is based on the Q-learning, it kept a major flaw. Q-learning (and thus DQN) have tendencies to non-uniformly overestimate the values. This can lead to picking non-optimal actions and thus making the learning phase less stable and longer. The problem comes from the maximum operation. Where the noises coming from the function approximator can be amplified and thus creating the overestimation [13].

Double Q-learning

Since this problem was already present in Q-learning, a solution was proposed to fix it, the Double Q-learning. Since it's the maximum function using the same value to select and evaluate an action which causes the overestimation, the solution found is to separate the selection and the evaluation. To realize that, the original Double Q-learning algorithm use two different network to select and evaluate. The target function can be rewritten as:

$$Y_t^Q = R_{t+1} + \gamma Q_{A_t}(S_{t+1}, \underset{a \in A}{argmax} Q_{B_t}(S_{t+1}, a))$$

where Q_{A_t} is the network randomly choose and Q_{B_t} is the other one. To give a better overview, we will show the Double Q-learning algorithm present in the paper [14]:

Algorithm 2 Double Q-learning

```
1: Initialize  $Q_A, Q_B, S_t$ 
2: repeat
3:   Choose  $a$ , based on  $Q_{A_t}(S_t, \cdot)$  and  $Q_{B_t}(S_t, \cdot)$ , observe  $R, S_{t+1}$ 
4:   Choose (eg. random ) either UPDATE( $Q_A$ ) or UPDATE( $Q_B$ )
5:   if UPDATE( $Q_A$ ) then
6:     Define  $a^* = \operatorname{argmax}_a Q_{A_t}(S_{t+1}, a)$ 
7:      $Q_{A_{t+1}}(S_t, a) \leftarrow Q_{A_t}(S_t, a) + \alpha(S_t, a)(R + \gamma Q_{B_t}(S_{t+1}, a^*) - Q_{A_t}(S_t, a))$ 
8:   else if UPDATE( $Q_B$ ) then
9:     Define  $b^* = \operatorname{argmax}_a Q_{B_t}(S_{t+1}, a)$ 
10:     $Q_{B_{t+1}}(S_t, a) \leftarrow Q_{B_t}(S_t, a) + \alpha(S_t, a)(r + \gamma Q_{A_t}(S_{t+1}, b^*) - Q_{B_t}(S_t, a))$ 
11:   end if
12:    $S_t \leftarrow S_{t+1}$ 
13: until end
```

We can see here that the network used for the selection is randomly chosen and the other one is used for the evaluation. The one choose for the selection is also updated. The paper shows that this solution corrects the overestimation bias and thus give better result.

Double Deep Q Networks

Finally, the Double Deep Q Networks is the sum of all precedents approaches. It keeps the advantages of DQN where the Q-learning have bad performance and correcting the problems of overestimation. The idea of separating selection and evaluation is kept but instead of using two distinct network randomly choose, the Double DQN use the online and target network. The selection part will be handle by the online network and the evaluation part, by the target network. The target evaluation becomes:

$$Y_t^Q = R_{t+1} + \gamma Q_{\text{target}_t}(S_{t+1}, \operatorname{argmax}_{a \in A} Q_{\text{online}_t}(S_{t+1}, a))$$

The target network parameter update doesn't change. This modification allows to use DQN with a more stable learning since the overestimation of action selection is gone.

2.3.3 Replay Memory

The basic reinforcement learning algorithm uses sequential experiences as input to create its model. This leads to a problem of correlations between the experiences

and thus creating bias in the model. To correct this, we can break the correlation between them by storing the experiences and randomly sampling them each time when we update the model. This method is called memory replay and has shown good results by stabilizing the result. [15]

In more details, we define an experience, e , by a tuple containing the actual state at time t : s_t , the action who was chosen: a , the next state at time $t + 1$ who has been achieved by choosing the action: s_{t+1} , and the reward for achieving the next state, r . In our implementation, we also use one more field, $done$, to indicate if the game is over.

$$e = (s_t, a, s_{t+1}, r)$$

Each time the agent performs an action, we store those data into a new experience and this experience is stored into a simple list. When the agent wants to update its model, it takes a batch of uniformed random experiences sampled from the list. By taking a random sample of experiences, this breaks the correlation between the experiences and thus allows for a more stable learning of the agent. The correlation is a problem because feeding correlated experiences to a neural network leads to overfitting and thus reduces the performance in more general cases.

This improvement comes with a cost, by using an uniformed random sampling, every experience is considered as of equal importance which intuitively doesn't make sense. This issue is also addressed in a more advanced approach which we will discuss later.

The size of the memory is an important consideration. Intuitively, the larger the size of the memory, the more efficient it would be but it's not the case. It was shown [16] that the memory efficiency is not a monotonic function of its length. The majority of RL developers use a size of 10^6 which seems to be decided by a rule of thumb [17]. Nevertheless, it remains better to tune the size to achieve better performance.

2.3.4 Prioritized Experience Replay

As discussed before, Replay Memory has a flaw. By uniformly sampling, it considers that every experience is of equal importance to another which, intuitively, is not correct in real-world problems. For instance during the creation of an autonomous car, at the beginning of learning, actions which lead to accidents should be more important than regular actions.

Prioritized Experience Replay, or PER, corrects that situation by prioritizing some

experiences over others. In order to realize that, a measure is needed to compare an experience to another. This measure should reflect the potential learning gain of an experience. This is a difficult concept to translate into a concrete measure. To do that, the PER authors [18] decided to use the magnitude of the absolute TD-error, as explained in equation 2.10.

As the TD-error shows the difference between 2 estimates of a state, the idea is thus, the more surprising the transition experience is, the more there is to learn from this experience. These surprising experiences receive a higher score and are taken in priority compared to the others with lower scores.

For performance sake, the algorithm doesn't sweep over the entire memory, thus only the experience scores taken for the learning are updated. This leads to a problem where experiences with lower scores are not considered by the algorithm and making it prone to overfitting. To avoid this problem, stochatization is used. The algorithm will not anymore only take the experiences with higher scores. Instead, those experiences have a better chance to be chosen. The probability of an experience i to be picked up is given by this equation:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

where p_i is the TD score of the experience i , the exponent a define how much the prioritization is used and k being any one of the experiences inside the memory.

Prioritizing experiences with higher TD scores lead to another problem. In fact, the estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation [18]. By prioritizing those scores, the algorithm creates a bias when the weights inside the neural network are updated. This bias is a problem because the distribution is changed and thus the solution that the estimates will converge to is modified. In other words, the bias will give more weight to corresponding high priority experiences, these amplified weights will still have a big impact on the model even if the agent has learned about these experiences. To avoid this, the algorithm uses a importance-sampling on the model's weights, this result in lowering the gap between the weights:

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

β is a hyperparameter which allows to control the unbiasing weights updates and the value is lesser than one and greater than zero. According to the paper [18],

unbiasing is more efficient at the end of the learning. β is thus set at a small value at the beginning and increase toward the value 1 at the end.

With all these modifications, the PER algorithm allows to have a more efficient learning with less data. The paper previously cited shows cases where the use of PER gives better results than without. Having a better learning with less data is particularly interesting since RL requires a lot of data to work. Reducing this number is a blessing when collecting data is costly.

Chapter 3

Architecture

In the previous sections, we described the theoretical knowledge concerning techniques for doing reinforcement learning with Q-learning. This chapter will be dedicated to describing our implementation choices and summarizing how to program all the algorithms.

3.0.1 Architecture of the networks

In the code whereas we wanted to process numerical values, we used a MLP with as little neurons as possible, to ease the training. We used most often:

- **Input layer** →

Hidden layers:

- Linear 512 neurons → ReLU →
- Linear 256 neurons → ReLU →
- Linear 64 neurons → ReLU →

Output layer:

- Linear *action space* neurons

When we needed to process images, we used a convolutional network:

- **Input layer** →

Hidden layers:

- Convolution, out: 32 channels, kernel 8x8, stride 4 → ReLU →

- Convolution, out: 64 channels, kernel 4x4, stride 2 $\rightarrow$ ReLU $\rightarrow$
- Convolution, out: 64 channels, kernel 3x3, stride 1 $\rightarrow$ ReLU $\rightarrow$
- Flatten layer (1024 nodes) $\rightarrow$
- Linear 512 neurons $\rightarrow$ ReLU $\rightarrow$

Output layer:

- Linear *action space* neurons

3.0.2 Transcribing the theory in pseudo-code

Now that the theoretical part is well described, we can show how we translated that into a low level code. We can start with the main loop (Algorithm 3) of our program, which is structured as follows. It represents the continuous loops that train our AI. In the actual Python code, it is named *train*. The first loop iterates over what is called episodes, which are one version of the simulated environment, between its creation and the end of the simulation (fail, win, or time limit). The inner loop plays over time steps, that is between one state and the next one.

Algorithm 3 Training the agent

```

1: procedure TRAINING LOOP
2:   for every episode do
3:     Create new environment
4:     state  $\leftarrow$  Get initial state
5:     for every time step do
6:       action  $\leftarrow$  Select action(state, step)
7:       Apply action on environment
8:       Observe next_state, reward from environment
9:       transition  $\leftarrow$  (state, next_state, action, reward)
10:      Store transition in memory
11:      state  $\leftarrow$  next_state
12:      Update neural net
13:      break if episode ended
14:    end for
15:    if DQN then
16:      Periodically update target net on online net
17:    end if
18:  end for
19: end procedure

```

In order to select the appropriate action based on the current state, we use the function **Select action** described in Algorithm 4, called *act* in the code. It uses the epsilon greedy policy to choose if at this time step, it should try an exploration action (at random), or exploitation by choosing the most appropriate action from its knowledge. If it chooses the best action, the neural net computes which action is associated with the highest Q-value.

Algorithm 4 Selecting an action

```

1: function SELECT ACTION
2:   compute probability following epsilon greedy policy
3:   if probability > rand then
4:     // exploration
5:     action  $\leftarrow$  random action
6:   else
7:     // exploitation
8:     Q-values  $\leftarrow$  predicted Q-values on online net
9:     action  $\leftarrow \operatorname{argmax}_a(Q\text{-values})$ 
10:  end if
11:  return action
12: end function

```

When a new transition has been observed from the environment, it needs to be stored in the replay memory or PER. The simple function **Store** is responsible for pushing that transition in the memory

With this new transition created, the neural net should be trained over it. Then it will be more appropriate for predicting transitions of this kind. This is the job of the function **Update** (Algorithm 5) or *update_net* in the code. Actually it is more efficient to train the network over a batch of transitions, sampled from the replay memory, so we start by selecting those.

Here we apply the Bellman's equation by computing Q-values: the current Q-values, next Q-values, get the temporal difference over them, and compute the loss, in order to finally do the backpropagation algorithm.

3.0.3 Architecture of the code

As we already translated the theory in pseudocode, we will now provide some explanation about the code itself. As a reminder, all the material we used can be accessed online at the github repository:

https://github.com/dVaneberck/Master_Thesis

Algorithm 5 Training the neural network

```
1: procedure UPDATE
2:   Select batch of transitions from replay memory
3:   Separate transitions in batch into: states, next_states, actions, rewards

4:   # get  $y_i$ :
5:    $online \leftarrow$  predicted Q-values on online net, for all actions of states
6:    $online\_Q \leftarrow$  Q-values of states, only for selected actions

7:   # get  $t_i$ :
8:    $target\_next\_Q \leftarrow$  predicted Q-values on online net, for all actions of
   next_states
9:    $best\_action \leftarrow$  action with best Q-value for next_states, according to
    $target\_next\_Q$ 
10:   $next\_Q \leftarrow$  predicted next Q-value on target net, for  $best\_action$  of
   next_states
11:   $td\_target \leftarrow reward +$  discounted  $next\_Q$  (value of Bellman's equation)

12:   $loss \leftarrow \mathbf{Loss}(online\_Q, td\_target)$ 
13:  Backpropagation on online net with  $loss$ 
14: end procedure
```

In order to implement the artificial neural net, we decided to use the Python library PyTorch. It provides easy ways to do machine learning. It is based on Tensors, that are NumPy-like multidimensional arrays.

PyTorch internally uses the CUDA library, so the tensors and the computations over them can be done directly using the GPU. This is a desirable property, because the GPU's differ of the CPU's with their huge number of cores that enables to compute similar operations in parallel. It is particularly useful on the heavy computations of convolutional networks with their big size. Indeed, their shared filters apply the same convolution everywhere on the image, so they are done in parallel, everywhere at the same time.

PyTorch also has a built-in engine for automatic differentiation, called *torch.autograd*. So when we have a computation over some variables, the engine has a way to compute their derivative as it knows which operation has been done over which object.

When we want to access the derivative of the loss of our network with regards to the parameter, we can just call *loss.backward* to do the computation, then PyTorch can already show the value of the derivative for each parameter, this does the backpropagation of the errors, then we call *optimizer.step* to update the parameters.

The code we developed was started using the PyTorch tutorial https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html. We used some of its ways to develop an algorithm of reinforcement learning, but we modified it, notably to make it more inter-operable with other games, as long as they use a Gym environment. Also we added the additions discussed in the theory part.

We decided to use the Bridge design pattern as the idea of how to structure the program. It is usually used to separate an abstraction of its implementation. For us, it allows to separate the agent abstraction with the different implementations of them, and separates the different implementations of neural nets.

Since we have multiples games that need the agent to behave differently, we used a Agent abstract class that already defines most methods, this class can then be extended to get the different version of agent needed for each game. Each agent flavour can then adapt to the specificity of each environment. With some of the methods already defined in the abstract class, we avoid most code duplication.

In this way, the main code found in **reinforcement_learning.py** stays relatively close to the pseudocode 3. The differences are hidden inside the methods of the specialized agent class.

Each of agent must choose a specific implementation of a neural network to use,

there is where the bridge occurs.

The files are structured as follows:

reinforcement_learning.py

It is the start of the program, this is the file that must be run, with one argument, that is the path to the configuration file to be used for providing all meta-parameters.

- **main()**: This method is responsible to decrypt the arguments, create the corresponding specialized agent of the needed game, then launch the training loop.
- **train()**: As we already stated, this method is basically the algorithm of reinforcement learning described in pseudocode 3. The main difference is around the specialized function *act*, that is now responsible to do 3 main things at the same time, more details on its own section.

agent.py

This is an abstract class, but with many methods already implemented. Those methods are the ones that should stay common between all specialized implementations of agents. In this way, we avoid duplicating lines of code when it is not needed. Its instance object possesses many meta-parameters passed as arguments to the program, and also the memory and neural network we asked for.

- **update_target()**: This methods copies the online network to the target network, in case of DDQN.
- **store()**: It simply stores a transition in memory.
- **update_net()**: It basically does the algorithm 5, responsible to train the online network with backpropagation. So it samples a minibatch from memory, computes the Q-values and TD-error over them, and updates the network.

xxxxx_agent.py

This would represent the specialized agent class for game "xxxxx", extending abstract class **agent.py**. In its instance object, there are a few more things added: most variables that are needed only for this game, or variables that change for each agent. Typically, those can be the environment Wrapper that transform the raw Gym environment to its pre-processed one that we would like to use. And of course, it must implement the abstrat method of **agent.py**.

- **reset()** This is simply the method to call to reset the environment and start a new episode. This gives us the initial state. It needs to change for each game, because all the Gym environment do not return the same type of object as a state, and we would like to make it cleaner before sending it to the main code.
- **act()** This method corresponds to `Select_action` in algorithm 4, where we apply epsilon-greedy policy to choose an action. But now it is also responsible for the 2 next steps: applying the action on the environment (**env.step()**), and observing the reward and new state (which is the result of the **env.step()** function). It needs to change between games since the actions do not have same form in each one. This method return the new observation to the main code, alongside the action it has chosen to apply.

neural_net.py

This file contains the abstraction of a neural network, in the form of a simple interface. Then we added our implementation of them. Each one must define what kind of layer is used, their parameters, and the **forward.py** method, necessary for telling how the stream of data must flow during the forward pass.

preprocessing.py

This file is dedicated to the preprocessing part of our algorithm. It contains several classes that implements a Wrapper of the gym environment. Each one of them transform how the environment behaves or transforms just the observation of the state that is returned. Each wrapper performs a particular processing operation like grayscaling or resizing, etc... In that way, when the **act()** function calls the **env.step()** function, it receives directly the modified observation of the state, without seeing any of the difficult transformations done before.

prioretized_experience_replay.py

This file contains two classes. The first one is the implementation of the PER and the second class is a generic implementation of a sum tree. This sum tree is needed to store the value used by the PER, this allows to access the data in less steps than using a list. Since the sum tree implementation is generic, we will not cover it.

- **push()**: This method is used to push an experience in the sum tree and giving to this experience an high priority.
- **sample()**: This method is used to return a batch of experiences from the sum tree with their indexes inside the tree. It also computes and returns the

importance sampling of the weights.

- **update_priorities()**: This method is used to update the priorities of the elements inside the tree linked to the indexes.

Interoperability

If one wanted to reuse this program to learn another game, the only thing that should be done is to create a new specialized agent class that implements methods **reset** and **act**, and if needed providing new implementations of neural network and preprocessing environment wrapper.

Chapter 4

Applications and results

For all of our applications, we decided to use the python toolkit **Gym**. It is specialized in providing environments suitable for reinforcement learning. There is a lot of environments already available, and some third parties have also developed custom ones.

4.1 Cartpole

Cartpole was our first case to resolve, and it's a well know problems in reinforcement learning and thus a good entry point into this world. Cartpole is a game defined by the following principle: a pole is linked to a cart moving along a track. The player can only have an impact on the pole by pushing it to the right or to the left. The cart will move according to the pole (if the pole goes to the right, the cart will go the to left and reciprocally). The goal of this game is to balance the pole as long as possible. In order to lose, the pole must be 15 degrees away from the vertical or the cart must move 2.4 units away from the center.

4.1.1 Environment

To resolve this task, we used the cartpole Gym environment, which allows us in this case to go in two directions to solve the problem. By using the images as input or by using the numerical values given by the environment. The numerical values correspond to: cart position, cart velocity, pole angle, pole velocity at tip. But the programmer should not be concerned with what they represent, reinforcement learning should discover their influence by itself.

As discussed in 2.2.1, the use of the latter approach should be more efficient.

However, the numerical value approach is difficult to transpose to other kinds of problems. To illustrate this difference in efficiency, we decided to do both approach and compare them.

4.1.2 Adaptations to the architecture

Since Cartpole is the first environment, it serves us as the basic architecture model. Even though it's a simple environment, some design decision have to be made in order to have a trainable agent.

Pre-processing and preparing the input

When we used screens to feed the agent, we had to do some preprocessing to avoid images with too much information. The gym environment gives us a 600x400 rgb image, which is a lot of information. In order to have simpler images, we decided to use greyscale images instead of an rgb. Colors doesn't have any impact to keep the pole and the cart on a good position. To go further, since the image only contains three elements: the pole, the attached cart and the background, we decided to only use black and white colors. With this approach, the three elements are still perfectly distinguishable while maintaining a low amount of noisy information.



Figure 4.1: On the left, the frame produced by the gym environment. On the right, the image pre-processed which is given to the agent to learn.

The second processing we performed was to reduce the image resolution. Again, since the image is simple, this reduction allowed use to discard noisy information and keep the important ones. Finally, since movement is an important part of Cartpole, feeding the agent a single screen is not sufficient. Thus, we gave the

agent a batch of four successive screens. With these successive screens, the agent could deduce information like speed or acceleration.

Adapting to the specific environment

The neural networks used for the numerical and screen approach are the same with the exception of the convolutional part for the screen approach. The neural network is a succession of linear layers of size 512 $\rightarrow$ 256 $\rightarrow$ 64 until a final linear layer which give the score of the two actions (left or right). The convolutional part is a succession of three layers, each one decompose the image with 64 channels, the difference between each layer is the size of the kernel and the stride. The kernel size are 5,4,3 and the stride size are 3,2,1 for the first, second and third layer.

The reward function is pretty simple. As long as the pole is upward, the agent receives a reward of one which is added to the cumulative reward of the episode. When the pole fall, the episode is over and the cumulative reward is reset for the next episode. Since we want to play as long as possible, the aim is to have the highest possible cumulative reward.

4.1.3 Results

We used Cartpole as our benchmark to test several potential improvements and to compare them. We chose Cartpole as our experiment environment since it's a relatively simple game and the learning time is short. For the following subsection, we will show the efficiency differences by introducing the more advanced techniques.

Note that for the following graphs, the x axis represents the learning time in minutes. The y axis represents the cumulative reward of the episode at the time x. We chose these two values as the base of our comparison as they are the two main factors of learning time.

Numerical and screen input comparison

As we stated before, the use of numerical input instead of the classical screen input should shorten the learning time of the agent. To show this, we performed two runs of 20 minutes. One for the numerical input and the other one for the screen input.

We tried to make everything else equal by using the same techniques. So we used epsilon greedy policy, DDQN and PER for both. The only difference present was in the neural network which cannot be avoided. Since we need to process an image when using screens, we should use a convolutional network for that. However, the rest of the network is the same neural network used as for the numerical input.

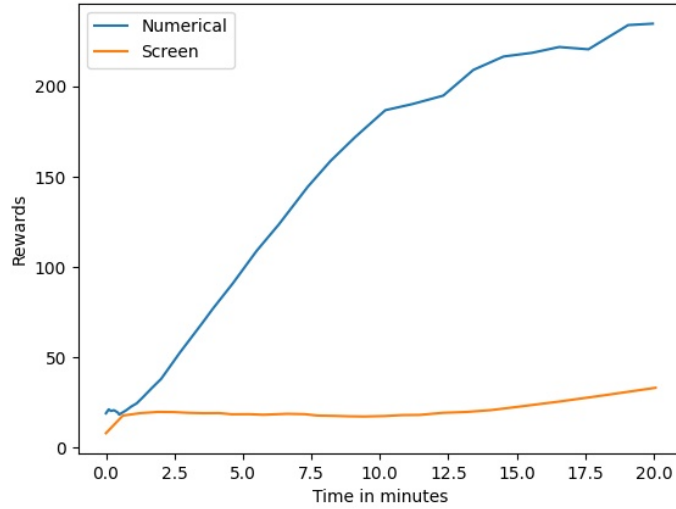


Figure 4.2: Comparison between numerical input and screen input in Cartpole environment

The result shown on the graph in Figure 4.2 are unequivocal, using numerical input give far better results in lesser time than using screen input. A specific neural network for the screen input with a better design for the convolutional part could give better results, but nothing which could approach the efficiency of numerical input.

DQN and DDQN comparison

For this comparison and the following ones, we decided to use numerical input because of its more stable learning. In this way, we can limit variations due to randomness which could impact the comparison.

As stated in the Deep Double Q Networks section (2.3.2), DDQN is a technique which aim to make the learning phase more stable and efficient. To perform this comparison, we decided to disable all techniques so no epsilon greedy policy and replay memory is used in place of PER. In order to only catch the effect of DDQN on the learning.

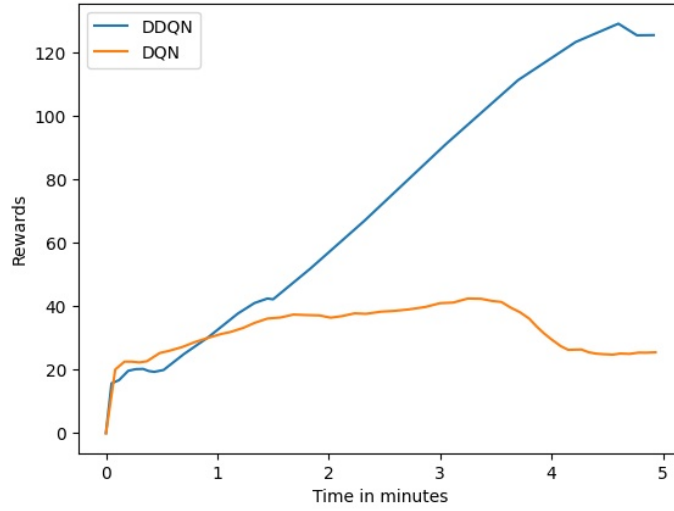


Figure 4.3: Comparison between DQN input and DDQN input on Cartpole environment

The result of the graph in Figure 4.3 are clear. Even if the run time is shorter than the previous comparison, we can see a massive difference between using DDQN or DQN. The learning is more stable and more efficient. These results comfort us in the usage of this technique.

Epsilon greedy policy comparison

Epsilon greedy policy promise us a better learning, by using a compromise of exploration and exploitation of actions. For this comparison, we keep the same approach than in the DDQN comparison, by disabling all techniques with the exception of the one we want to test. This time we used a training time of 30 minutes.

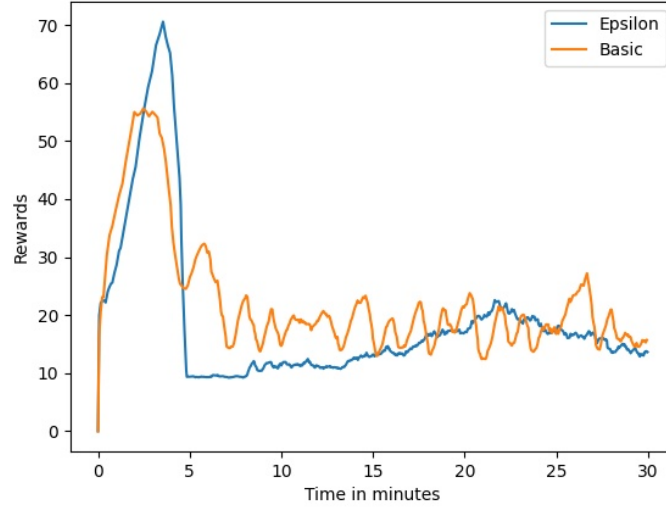


Figure 4.4: Comparison between using epsilon greedy policy and basic policy in Cartpole environment

This time the results are not as good than the previous. The graph Figure 4.4 show us that Epsilon policy give better performance at the beginning of learning around 4 minutes. However, we can clearly see a case of catastrophic forgetting which is a common problem in reinforcement learning [19]. Even after the forgetting, we can see that the learning when using Epsilon is smoother than with the basic policy.

Prioritized experience replay and memory replay comparison

Prioritized experience replay is the most complex technique we have implemented to help the learning of our agent. According to the PER paper, it should allow our agent to have a more efficient learning. To show this, we decide to compare the PER with its most basic version, the Memory Replay. For the first experiment, we decided to disable all others techniques (DDQN and Epsilon-greedy policy) to only see the effect of PER.

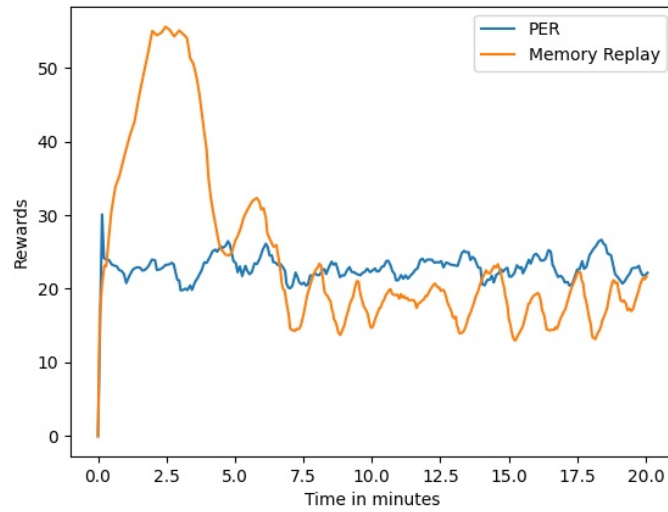


Figure 4.5: Comparison between PER and Memory replay, with all other techniques disabled, in Cartpole environment

As shown in Figure 4.5 the results are pretty bad. Even if Memory replay also performs badly with the absence of other techniques, PER is even worse. Where Memory Replay manages to learn at the beginning before forgetting, PER is constant and doesn't seem to learn at all. It is difficult to compare them in this situation. However, we will not stop the experiment here by comparing the 2 memory techniques with DDQN and Epsilon policy in a second comparison.

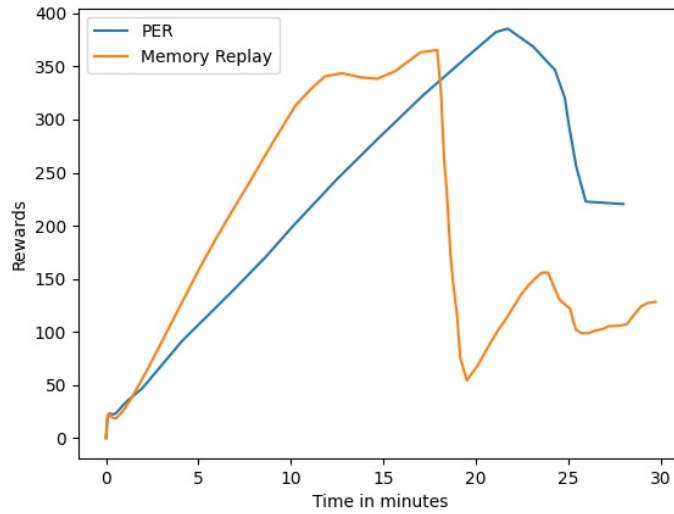


Figure 4.6: Comparison between PER and Memory replay, using all others techniques, in Cartpole environment

The Figure 4.5 show a totally different behavior. This time, both approaches have showed a good learning behaviour. We can see that Memory Replay give better result at short term but it's overtaken by PER after some time. Unfortunately, we can note in both approach that after some time, they both are affected by forgetting. Despite this problem, these results confirms that the usage of PER instead of Memory Replay is beneficial, since the learning manages to be smoother and better.

Results conclusion

These comparisons show us how sensitive is the learning to the introduction of several techniques. It also demonstrates how the individual use of these techniques like Epsilon policy or PER doesn't seem to impact the learning but combined with other techniques, can dramatically change the results. However, these experiments also show us the catastrophic forgetting problem. Despite this, the results are good, and we are confident about the architecture of our agent. We will test it in a more complex environment to see if this forgetting problem also occurs there.

4.2 Super Mario Bros

Super Mario Bros was chosen as a case to resolve because of its more complex environment than Cartpole. For the later, the agent has to analyze a reduced environment with only two things to care about: the pole and the cart. In the case of Super Mario Bros, the agent doesn't only have to care about the actor but also about the environment which is richer.

We had the idea to use the Mario Bros. game thanks to the Pytorch website which hosts several guides and one of them was about Super Mario Bros. This one gave good result by using Replay Memory, so we were curious about the results when comparing our Mario implementation (with a more advanced memory management, PER) with the existing one.

Before explaining our solution, we will provide a summary of what Super Mario Bros is. Super Mario Bros is a platform game in two dimensions where the protagonist, Mario, have to pass levels to finish the game. A level is a series of obstacles that the player must complete to finish the level. Obstacles can take different forms, like an enemy or an element of the decor (a hole, a wall...). These obstacles can be avoided and/or destroyed.

4.2.1 Environment

The gym environment for Super Mario Bros is rich. This one allows us to select specific levels to let the agent train. It also only gives rewardable frames to the agent, so we don't have to take into account noisy image like loading screen or cut-scene. It also gives us the possibility to use different environments where some frame processing is already done, like reduced resolution.

4.2.2 Adaptations to the architecture

To get an efficient model, some design choices needed to be done. For this task, they concern mainly the frames pre-processing, the neural network design and the reward function.

Pre-processing and preparing the input

Our first concern was to deal with the input. Unlike Cartpole where we have the choice between numerical input and image, this time we had to use images. This is due to the Super Mario environment where it was not possible to get specific data like the speed of an enemy or Mario. Images were thus used, but we couldn't feed them directly to the neural network. As discussed in 2.2.1, images contains too



Original frame



Pre-processed frame

Figure 4.7: On the left, the frame produced by the gym environment. On the right, the image pre-processed which is given to the agent to learn.

much information thus we need to make some transformations in order to remove noisy information. To keep a meaningful comparison with the PyTorch website, we have kept the same transformation they used.

These transformations were in the first place, skipping frames. Gym environments work by rendering every frame, which is a lot of work. Using all the frames is not necessarily interesting since two consecutive frames contains a lot of the same information. Thus, it was decided to drop some of the used frames in order to have more diversified information. In the implementation, a skipping of four frames was used. We use a stack of frames of size 4 for the same reason than when we did it in Cartpole.

The second transformation used was the grayscaling. An image contains a lot of elements, some of them are relevant and others not. Colors, for humans eyes, are a great tool to make a better analysis of a situation, like traffic lights. For an artificial intelligence, color can be detrimental, depending on the nature of the image. In the Super Mario Bros case, differentiating colors is not mandatory to finish the game. To keep image information as relevant as possible, it was decided to grayscale them to not let the AI get distracted by the colors, and to help reduce the size of data to process.

The third transformation is a simple downsizing in order to have a good size to put inside the neural network. The result of the transformations is shown on Figure 4.7.

Adapting to the specific environment

In this part, we decided to use the same neural network as the one the PyTorch website used, in order to have a meaningful comparison. The aim of this step was to see if the implementation of advanced techniques like PER had an impact on the performance, we didn't want to have more differences, like a different neural network to perturb the results. The network is described in 3 which in short is a convolutional network. The rest of the architecture is pretty much the same than with Cartpole.

The way the reward is given is different for every game, and thus is different than what we have seen in Cartpole. The reward with Mario was handled by the gym environment where the agent was rewarded if he goes to the right. The reward is given by the following equation [20]:

$$r = v + c + d$$

where r is the reward, v is the difference between the position before the step and after the step, c is the difference in the game clock between frames (this prevent the agent from standing still) and d penalize the agent if it dies. With this reward function, the agent is encouraged to go to the right fast as possible without dying.

4.2.3 Results

We will present two different results in this section. The first result will be about the learning phase of the Mario agent with a discussion about the reward impact on the agent. The second will be a comparison between the two memory management methods, the replay memory and the prioritized experience replay.

Learning result

We only trained our AI on the first level of the first world on Super Mario Bros. We decided this at first with the hope to quickly have a first good solution. In practice, it take more than 80 hours of training for our AI to get to a point where it can finish the level in 99% of the runs. The solution found is good, the AI is able to finish under 88 in-game seconds(35.2 real seconds). An in-game second corresponds to 0.4 real seconds [21]. Unfortunately when compared to the human world record (31 in-game seconds [22]), our agent is a lot slower. When analyzing the record, we saw that the human world record use shortcuts that our agent is not aware of. This is due to the reward function which prompts the agent to only go to the right and to not discover other ways.

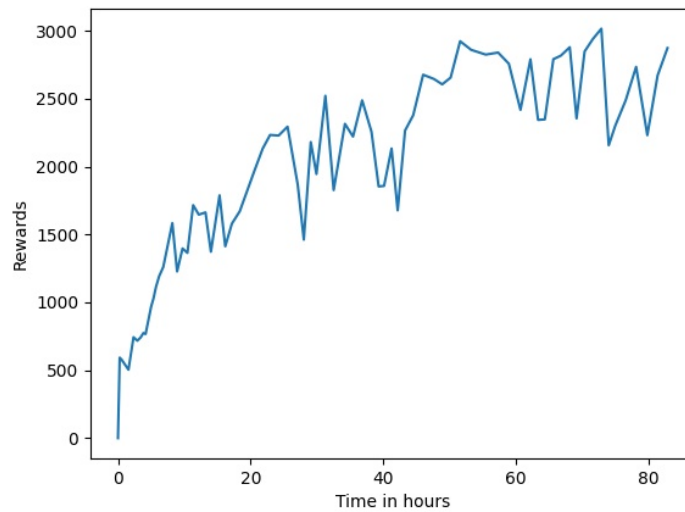


Figure 4.8: Evolution of the cumulative reward during the learning phase for Super Mario Bros, using all advanced techniques

This observation shows the impact of the reward function design. The reward function doesn't simply attribute a score to an action, but it leads the AI to a way of "thinking". A more complex reward function helps to have a faster learning but at the cost of neglecting some possibilities who could be favorable to obtain better performance. Simpler rewards explores more possibilities and thus could find better solutions [23], at the expense of learning time.

You can see on the graph of Figure 4.8 the evolution of the agent during the training phase. As you can see, even though the learning is not very smooth and stable, the learning is trending upwards. Even if there are some loss of performance during the learning, probably due to a part of random decision or maybe some case of forgetting, the agent will learn more and more in the long run.

Memory management comparison result

As stated at the beginning of this section, one of these reasons to have chosen this environment was to compare the utilisation of PER against the traditional memory replay used in the guide. To do this, we decided to run the two different techniques during a time of 8 hours. For the Cartpole comparison, we used much more shorter test time but with Mario since the environment is much more complex and images had to be used as input, we needed to test for longer. Note that in both case, Epsilon greedy policy and Double Deep Q-Networks were used.

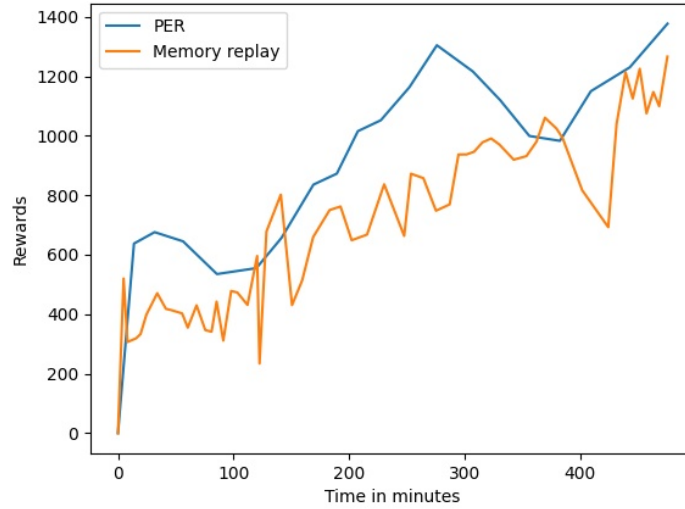


Figure 4.9: Comparison between PER and Memory Replay inside the Super Mario Bros environment

As we can see in Figure 4.9, the PER version gives a better learning than the Memory Replay version. The learning is smoother when using prioritized experience and gives better results in term of cumulative rewards. Remembering the result we had with Cartpole, this confirms the interest to use PER instead of the more classical Memory Replay.

Results conclusion

For us, the Mario experiment is a success. We have succeeded to make our agent learn a complex environment like Super Mario Bros. Even if a lot of time is required to achieve satisfying results, the learning is constant and stable. The forgetting problem present in Cartpole experiment doesn't seem to have much impact here. We see on Figure 4.8 some drop of performance but every time the agent's learning continues to grow. The Mario results makes us confident in the implementation of the agent and the design choices we made. We will finally test it inside a last environment for a first person game.

4.3 MineRL (Minecraft)

Minecraft is a video game where a player can move and interact with a 3 dimensional world. It is at the same time very simplistic as the visuals are very pixelated, but it also presents a lot of possibilities as the game is known to be a sandbox.

We have decided to use a library called MineRL that interacts with the game. Every communication with the game is handled by the library, and at the end a Gym-like interface is presented to the programmer. It was convenient to use, as we didn't have to play with linking the algorithm part of our code with the game itself (observing the screen, issuing commands to handle every action, timing the episodes...). Also, it is already doing some preprocessing of the game screen, as some less useful information is already removed (graphical user interface of the inventory items and health status, drawing of the hand and item in use). This is beneficial as it frees more area of vision to see more of the world around the agent, and it was not necessary to the task anyway. With this we can avoid more steps of preprocessing the image by ourselves.

We used version 0.3.6 of the library, it is possible that a newer version will need adaptation to our code to function (the library will return different name in the observation object).

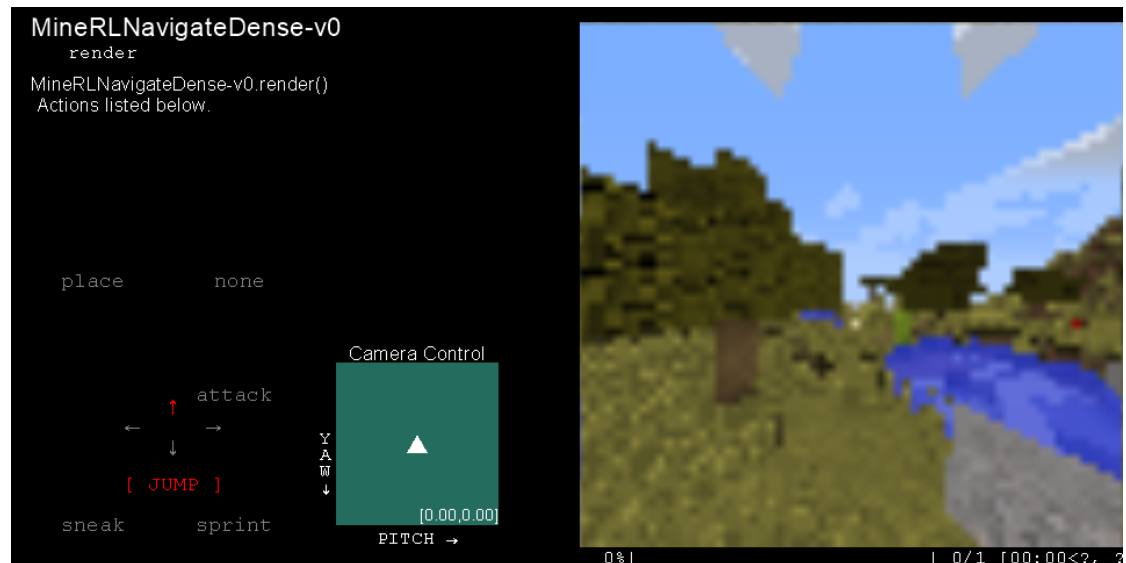


Figure 4.10: An example of the rendered environment

4.3.1 Environment

MineRL has different environments which presents various challenges. We decided to start with the NavigateDense environment, which consist of reaching a diamond block located 64 blocks away from the origin of the minigame.

With this environment, more difficulties are added and we need to handle them with care.

First, the environment has 10 actions that can be asserted at the same time, this means that we can not predict a single boolean value to represent which action to take.

Second, some of the actions are not simple boolean action. In particular, the camera needs to be moved using float values.

Third, there is 3 inputs available at the same time: the 1-st person view of the game, a compass pointing towards the destination, and the inventory of available items. None of the inputs have the same shape.

The environment used here provides dense rewards. This means that every action will generate a reward, proportional to the distance traveled towards the goal. This makes it fore feasible to train as the agent can begin its learning before reaching the goal for the first time.

The rewards are distributed throughout the game, proportionally to how many meters the agent walks toward the goal. As it is approximately 64 blocs away from the initial location, the agent will already had received a cumulative reward around 60 (actually anywhere 45-75) when it comes close, and touching the goal gives him an additional reward of 100.

One of the difficulties encountered in this environment is that when coming close to the goal, the rewards are not very precise, they could be slightly negative even when going towards the goal. Also, touching the goal is not very clear as sometimes, standing on top of it will not trigger the completion of the task, but walking to its side would. The documentation also says that it is possible that the goal would be slightly buried underground, and so, invisible.

4.3.2 Solution using images

Description of the solution

For NavigateDense, the first solution tested is to use only the videogame screen, and to feed it to a convolutional neural network (described in section 3.0.1). The goal is to reach a block of diamond.

As multiple actions can be asserted at the same time, we could try different ideas to get there, such as putting a softmax layer at the end of the network, and performing actions with a probability greater than 50%.

To help the agent a little bit in its learning, we decided to use only one action at the same time, and we reduced the options for the moves to only 5: turning the camera 1 degree left or right, or turning it 10 degrees left/right, and the last action is jumping forward. We decided to combine the action "jump" and "forward" to help it avoid small and common obstacles such as a level difference.

Observations and results

Here we used some more tools to understand the results of our algorithms: a graph of the rewards distributed over time, and some histograms of the compass angle over multiples episodes. The later should help us understand how much time is spent facing the direction we want (around 0°)

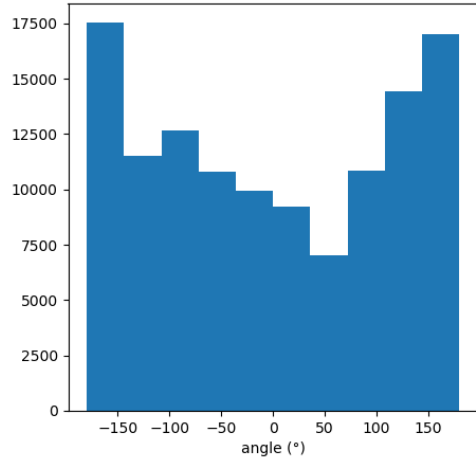
Figure 4.11 shows the distribution of the angles, and the evolution of the rewards over time when running the first approach to the problem.

We can observe that the angle of the compass tends to be a roughly uniform distribution, which would mean that the AI has not learned that a specific direction is useful, and it faces a random direction throughout its run. Also, when analyzing the cumulative rewards, it seems that as many points have a negative than a positive reward, and no trend line can be observed, which would mean no learning of its goal. Only one episode reached the goal (having more than 100 as a reward), probably by pure luck.

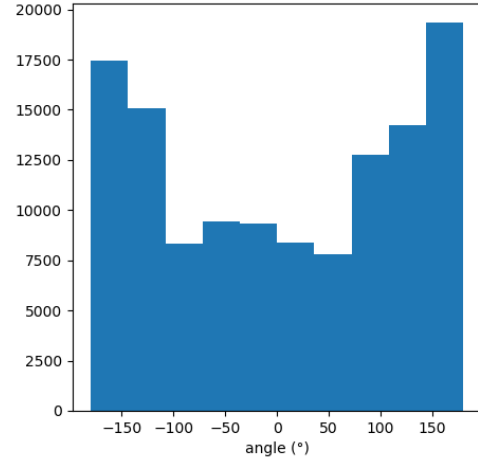
Learning using only this input might not lead to the desired results. Indeed, at first the goal won't be visible (due to distance or being hidden by the landscape), thus the agent can't locate it based only on its input. This breaks significantly the Markov Property.

Moreover, the maps created being random, the difference in state after the action is taken cannot easily be correlated with the reward received, except in the case of a visible imminent danger/death or win. A lot of specificity in the inputs (landmarks, trees, etc...) are different from a map to another and do not help to reach the goal. For example, even if the algorithm could remember that the landscape it sees looks like the one that lead to a previous victory, there is not much chance that these surroundings would also lead to a goal this time.

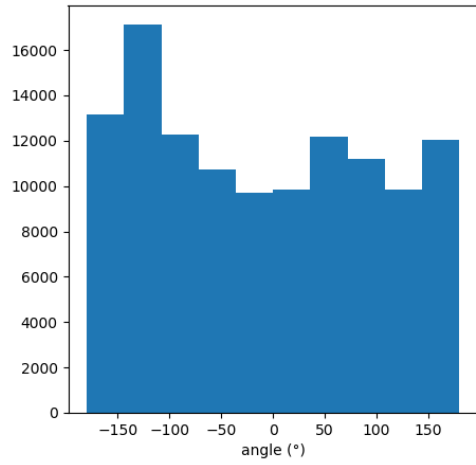
Also, sometimes the game world itself has a bug slowing the rendering, which cause the ground not to appear, which will confuse the agent even more.



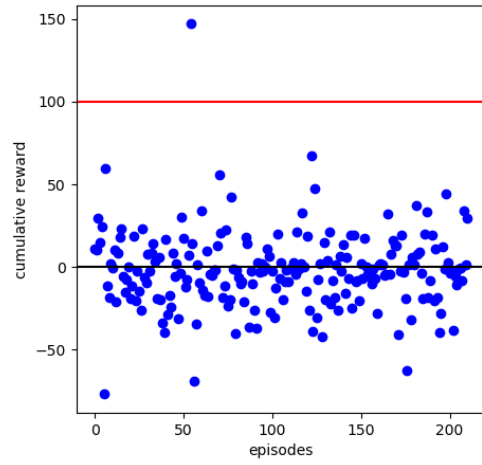
(a) Episodes 1-70



(b) Episodes 71-140



(c) Episodes 141-210



(d) Cumulative reward at the end of the episode

Figure 4.11: Data collected when running the convolutional network with PER on MineRL

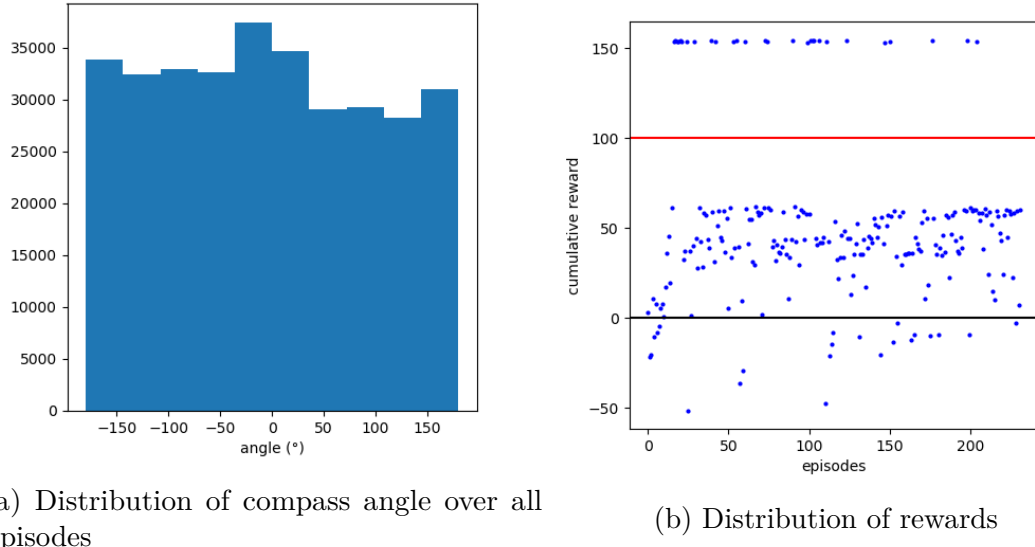


Figure 4.12: Data collected when running DDQN on MineRL, using the convolutional network with PER, by fixing the seed of the environment, *target_sync* = 2

Another idea: fixing the seed

The idea here would be to always use the same generated map for every episode. It would be interesting to see if the agent can learn what does the world looks like, and recognize the path leading to the destination. It should be a challenging task as a flat plain or a forest can look very similar all around, with not a lot of clues to differentiate the landscape.

In this part we decided to fix the seed in a forest, as it seems to be a good compromise between the amount of information available and the ease to navigate it.

In Figure 4.12, there is a big improvement over Figure 4.11, as most data points are getting a positive reward, and the agent achieves to reach the goal from time to time.

The situation is not perfect yet, as even though there is a peak around 0° , the distribution of compass angle is very uniform, meaning the agent still spend a considerable amount of time facing a random direction. This can be observed better when looking at the rendered environment, as we can see the agent often stop moving to adjust the camera angle in undesirable directions, even alternating

moves that cancel each other.

This means that the agent is not efficient with the time given to do its task, and the policy that it learns is not stable.

By observing the render, it seems that the agent tend to be conservative and favor paths it already knows, instead of learning a shorter one. It also seems that it is reluctant to take temporary negative rewards to reach its goal, when the direct path is not possible. This probably means that the agent stays too greedy with its rewards, preferring a instantaneous one, instead of a bigger one in the future. Or it could be that there was not enough exploration, as in the maze problem explained in an earlier section, which makes it favor the first paths it discovered.

Another observation made is that even though there is a high concentration of points with a reward between 45 and 60, the agent do not reach the final goal that often. It can be partly explained with the fact that the reward function doesn't point directly to the diamond block. Indeed, in this seed, this block is located at 54 meters (and a reward of 54) of the origin, but it is also possible to miss the block and reach up to 61 of reward. This can be considered as a local maxima, with the global maxima being 154 when touching the block. Here too, it seems that the agent is too greedy with rewards, it only gets the final reward if it gets really close to it.

4.3.3 Solution using numerical metadata

Description of the solution

The other observation available, the compass angle, could actually be way more useful to reach the goal, as it points to its approximate location. This second approach uses only that input. A simple algorithm walking straight towards it would have a better chance of succeeding than when using only the videogame screen. However, since the compass is only approximate (around 10 blocks away), when the agent comes close to the goal, the compass will begin to point a few meters away from the goal, so using only the compass input would also makes it impossible to reach the goal (or only by error of the agent).

Observations and results

This solution makes it possible to achieve some good results. In Figure 4.13, we can see that the AI starts by doing random actions, with the compass not pointing in the good direction, and rewards being positive as much as negative. But quickly it can learn to travel towards the goal: it spends most time facing the destination, and the cumulative rewards are mostly positive.

In some cases, it can even complete the task, though it is mostly due to luck as it cannot locate precisely where is the goal.

But Figure 4.13 shows a particularly good run, with the parameter syncing the 2 networks of DDQN set to update them very rarely. This has the effect of making a learned policy very stable. Sadly, it seems that achieving this result is not possible every time, as the Q-values can be badly estimated, and that bad policy would be kept for a long time. Sometimes the agent can also learn the right policy and forget it, or not even learn it. It is possible that since it doesn't have the sense of vision, any obstacles blocking its path can't be avoided, which is in contradiction with the information received from the estimated Q-values. The good policy learned until that point would then be replaced since there is a big difference between the expected and actual reward.

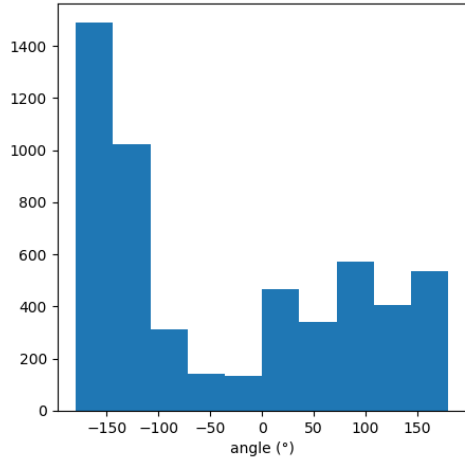
When a bad policy is learned, most of the time is used to make meaningless moves to the camera. With this, the exploration rate directly impacts performance as it can avoid getting stuck too long. The Q-values are overestimated and unstable for the camera moves. Also, it seems that the algorithm favors moves making the agent facing the opposite direction of the goal (compass between -170 and +170 degrees), which leads to big negative rewards.

Due to the high stability of the policy, if the Q-values are badly estimated, leading to a wrong result, it is nearly impossible that the algorithm will learn something better. This is why we tried to set the syncing parameter of DDQN lower, at a frequency of every 1 or 2 episodes. Like that, the policies are more likely to change over time, and to forget a badly learned behavior. It is also possible that a good behavior could disappear more easily.

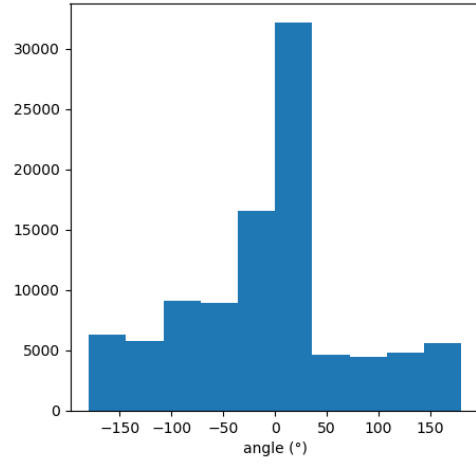
In Figure 4.14 we can observe that behavior. The learning seems at first to be pretty successful, the rewards grow more positive and the compass points to the direction we want. With time, the policy seems to get more unstable, the compass is not as focused on the goal, and a lot of points have a very small positive reward, some even have a slightly negative value too. But overall, the agent learns to walk in the positive direction, and the process is less likely to get stuck in bad policies.

4.3.4 Approaches using a combination of different inputs

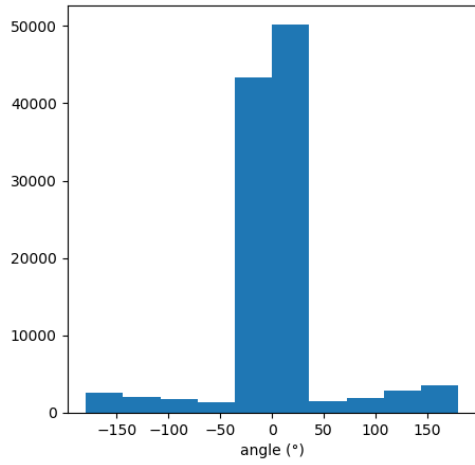
At this point, each previous solution showing some interest but being far from perfect, a solution worth trying is to create a neural network using both inputs. This way when far away from the goal, the compass angle is correlated with an increasing reward and so it is possible to train the model to converge. When coming close to the goal, the compass input is not sufficient to solve the game, but the goal should be visible on the image input. Since it is a pretty uncommon feature



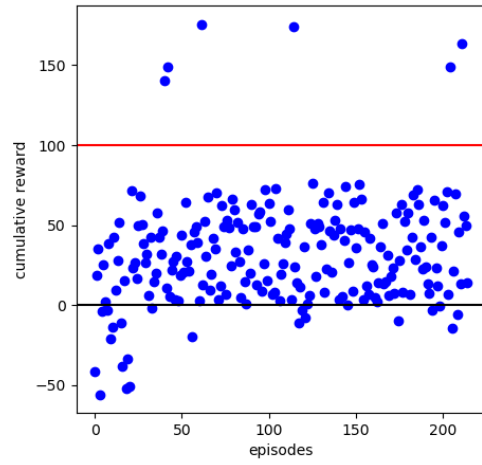
(a) Episodes 1-5



(b) Episodes 1-70

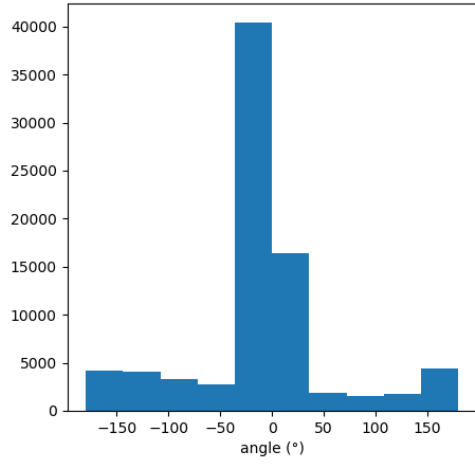


(c) Episodes 141-210

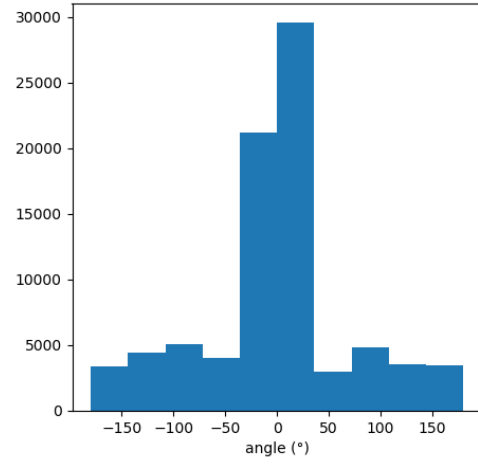


(d) Cumulative reward at the end of the episode

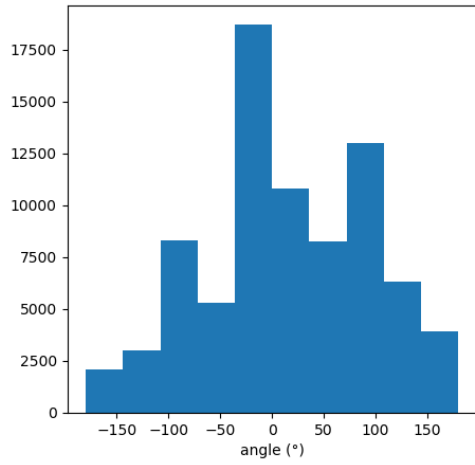
Figure 4.13: Data collected when running DDQN on MineRL, using the fully-connected network with PER, using compass data, with a *target_sync* = 10



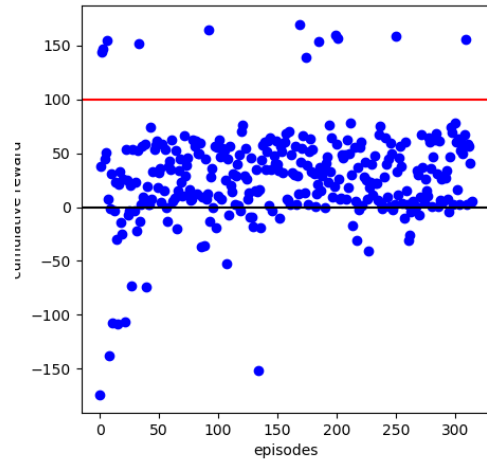
(a) Episodes 50-100



(b) Episodes 150-200



(c) Episodes 250-300



(d) Cumulative rewards at the end of the episode

Figure 4.14: Data collected when running DDQN on MineRL, using the fully-connected network with PER, using compass data, with a *target_sync* = 1

of the world (bright blue bloc), a convolutional neural network should be able to react to it, and lead the agent towards it.

For this implementation of the agent, it would be necessary to combine a convolutional network and a Multi-layer Perceptron in hope to solve the short comings of each previous methods. This seems to be a challenging task, the architecture of the neural networks will change a lot. This would also break some part of the modular code we have done, as it expects to get a single input to the neural nets and to have a single well-defined state to put inside the memory. For these reasons, we have implemented this solution yet, but it could be an interesting one to try.

Other approaches worth trying

Even though we always left the convolutional network process grayscale images, maybe in this case it would be profitable to use colored ones, as the goal has an especially uncommon blue color, it could trigger a stronger response in the network's channel treating blue pixels. Unfortunately, it also triples the amount of input data and makes the learning slower.

One of the disappointing observations we got in this environment, was the agent spending a lot of time alternating the cameras in opposite directions, thus only losing time, or failing to recognize a danger as a cliff or body of water, that could kill him. As an experiment, it would be interesting to modify the reward function to penalize him in these situations: a negative reward for standing still for too long, or for dying before the time limit was reached. We hope that this could help fixing the encountered problems.

Chapter 5

To go further

This thesis allowed us to explore the Reinforcement learning field and the many possibilities which can be added to it in order to gain better results. However, due to time restriction, choices have to be made to choose which additions we could add. This chapter will serve us to talk about and make a summary about interesting techniques that can be added to an agent to gain a better learning.

Dueling DQN

For some environment, the estimation of all actions is not necessary. However, in the traditional DQN, these computations must be done to create the model. Dueling DQN has been created to resolve this problem by separating the network in two parts [24]. The beginning of the network is the same than DQN, by having several convolutional layer and a flatten layer. But after those, the stream is divided in two branches. One to compute the value estimations and the other one to compute the advantage function. The advantage function is the difference between the Q-value for a state/action pair and the value function of the state.

$$A(s, a) = Q(s, a) - V(s)$$

The two streams are merged into a final one which produces a set of Q-values like a more classical DQN network. This allows Dueling DQN to be used with other techniques like PER. This separation allows the network to learn the states which are not valuable without computing the effect of each action.

Policy Gradients

In this thesis, we used Reinforcement learning to solve different kind of problems by optimizing the value function. However, RL area doesn't only concern value function optimization and there is other categories to explore. One of these categories is named Policy Gradient.

Policy Gradient categories differ from traditional value optimization category by directly computing the policy. On the contrary, in value optimization (such as DQN, DDQN or Dueling DQN) the policy is indirectly computed by computing the value function. To choose an action, Policy Gradient doesn't have to compute the estimations for every possible action, which is a nice thing when a large set of actions is possible or even for continuous actions. It just chooses an action and sees if it is beneficial, with this approach Policy Gradient will always converge on a local maxima. Unfortunately, discovering the global maximum could take a lot of time.

Pre-trained by human

One of the main problem about Reinforcement learning and Machine learning in general is the availability of data, and more specifically, the cost of data. They need a lot of data in order to create an efficient model. For some tasks like games, it's not too problematic since it's possible to produce a lot of frames for a cheap cost. Unfortunately, real-world application like self-driving car or drones have much more difficulty to produce data. Even more in reinforcement learning where at the beginning failing is the norm.

A solution to these costly environments could be the use of human input. By providing human experience, it could be possible to give the AI a first-hand knowledge, reducing the amount of learning time and the cost. This concept is called Human in the Loop [25]. For instance, in a game, providing the experience of a top player could help the AI to make the right choices. This was one of the approaches chosen by Google when making AlphaGo. In the case of a self-driving car, having human experience will help the car to have a good understanding of a traffic environment.

Chapter 6

Conclusion

In this thesis we learned that Reinforcement learning is an interesting approach to problem solving. It is able to learn how to reach a goal with absolutely no previous knowledge of it, which is already a nice and impressive property. In order to reach that it needs to be able to recognize images it receives, which is not trivial for a program. Then it should find a way to correlate its actions to rewards it receives and also to remember them in the long run. All of that while using a convolutional neural network, which is notoriously difficult to train, especially when few data points are available. It is a combination of many difficult tasks, and it is a beautiful sight to witness the agent succeed.

However, it is not only good, as the result can sometimes be very unstable: the program can be imprecise in its actions, leading to movements opposing each other, or even worse, leading away from the goal! We have often had problems with the agent temporarily reaching the solution, but quickly after forgetting how to do it again, and performing worse and worse. Some tasks are also nearly impossible to achieve, when the rewards are not densely available, or if the environment cannot sufficiently satisfy the Markov property. The algorithm is also fragile when using non-stationary environments, as it is quickly lost when it discovers something it hasn't seen before.

Also, training can be painfully long before seeing noticeable results. This is sometimes mitigated when the environment can be accelerated to compress time, but the neural network always need to be fed a sufficient number of observations to converge. And the programmer needs to find a way to give a strong reward function as input for the agent, in some cases that could prove to be not so straightforward. The Black box approach of it all could also be considered as a problem. The changes inside a neural network running are difficult to track and thus make the debugging a

non-trivial task. More generally, the whole design aspect of Reinforcement learning require each task to be heavily analysed in order to make the right choices.

Despite these difficulties, we still believe that Reinforcement Learning has some potential and deserves to be studied. It may need some additions and additional tweaking to make it reach more efficiently its goal, and in less time, but a lot of tasks can be done in that way, if enough clues are accessible in the state to do its tasks. Even other (and more scientific) applications that we didn't discuss here use RL to build models. The study of Reinforcement Learning should also not be limited to only one field. Even if our work was primarily focused on value based algorithms, others fields like Policy Gradient are promising approaches to break the actual limit of AI.

It also seems that we are not the only ones sharing the opinion that RL has some beautiful days ahead. We can cite the famous Google DeepMind among others who still develop their version of RL, and have already got very impressive achievements. Recently, they have even claimed in their paper "Reward is enough", that they have all the components so that their reinforcement learning algorithm could reach general AI, a form of human-level intelligence. If this is true, their AI will be able to learn any problem by trial and error, without any need of having a problem-specific formulation.[23]

We will take advantage of the end of this conclusion to briefly introduce a problematic far from the one we have covered so far. This concerns the environmental cost of reinforcement learning and more generally, the use of neural networks. Researchers [26] show the energetic problem and thus ecological problem of using large neural networks. At a time where many researchers look at ways to reduce our environmental footprint, it can seem concerning to let powerful graphics card run at full power, days at a time. Even if this thesis was not about these kind of problems, it seem important to highlight it.

Finally, this thesis gave us the opportunity to learn and work on an advanced AI concept with a playful environment. Even if Reinforcement Learning fits so well to be used with video games, since the experimental cost is low, its possibilities are not limited to that. For us, this thesis helped us to discover how powerful is RL, and how it could greatly improve the AI field.

Bibliography

- [1] Kate Lyapina. Mission possible — reinforcement learning meets the real world. <https://towardsdatascience.com/mission-possible-reinforcement-learning-meets-the-real-world-296b21ec672b>. [Online; accessed 06 June 2021].
- [2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*, chapter 3, 6, pages 47–69. The MIT Press, second edition, 2018.
- [3] Marco Saerens. *Lecture notes of Data Mining and Decision Making - Markov Decision Process*. Université Catholique de Louvain, 2019.
- [4] Christopher Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, England, 1989.
- [5] Michel Verleysen. *Lecture notes of Machine Learning : regression, deep networks and dimensionality reduction - Deep Learning*. Université Catholique de Louvain, 2019.
- [6] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. *Object Recognition with Gradient-Based Learning*, pages 319–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.
- [7] Christoph Molnar. Learned features. <https://christophm.github.io/interpretable-ml-book/cnn-features.html>, 2021. [Online; accessed 24 June 2021].
- [8] Renu Khandelwal. Convolutional neural network: Feature map and filter visualization. <https://towardsdatascience.com/convolutional-neural-network-feature-map-and-filter-visualization-f75012a5a49c> 2020. [Online; accessed 24 June 2021].
- [9] François Chollet. *Deep Learning with Python*. Manning, November 2017.
- [10] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

- [11] Elena Mocanu, Phuong H. Nguyen, and Madeleine Gibescu. Chapter 7 - deep learning for power system data analysis. In Reza Arghandeh and Yuxun Zhou, editors, *Big Data Application in Power Systems*, pages 125–158. Elsevier, 2018.
- [12] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.
- [13] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *In Proceedings of the Fourth Connectionist Models Summer School*. Erlbaum, 1993.
- [14] Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.
- [15] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Schenley Park Pittsburgh PA, United States, 1993.
- [16] Ruishan Liu and James Zou. The effects of memory replay in reinforcement learning, 2017.
- [17] Shangdong Zhang and Richard S. Sutton. A deeper look at experience replay, 2018.
- [18] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.
- [19] Wikipedia. Catastrophic interference — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Catastrophic%20interference&oldid=1037136890>, 2021. [Online; accessed 08-August-2021].
- [20] Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. [Online; accessed 05 July 2021].
- [21] MarioWiki. Time limit. https://www.mariowiki.com/Time_Limit. [Online; accessed 28 June 2021].
- [22] Niftski. Any% in 4m 54s 948ms. <https://www.speedrun.com/smb1/run/zqgxj79m>. [Online; accessed 05 July 2021].
- [23] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299, 2021.

- [24] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.
- [25] Robert Munro. *Human-in-the-loop machine learning*. Manning Publications, 2021.
- [26] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl