NTNU – Trondheim
Norwegian University of
Science and Technology

Deep Reinforcement Learning : Study of Advanced Exploration Methods

Alexandre Caulier

December 2016

TDT4501 : Computer Science, Specialisation Project

Supervisor 1: Humberto Castejon

Supervisor 2: Arjun Chandra

Supervisor 3: Massimiliano Ruocco

Preface

This document contains all the work that I have realised for the Computer Science Specialisation Project at NTNU. It was carried out during Autumn 2016 in collaboration with Telenor Research in Trondheim. The subject of this project was proposed by the researchers in the company. Briefly, the goal was to study different exploration methods in Reinforcement Learning and try to evaluate them. The cooperation with Telenor was a weekly meeting with the researchers were I explained what I read and implemented. It was very interesting talks that often lead to new insights.

To be able to fully understand this report, the reader should know the basics of Deep Learning: what is a Neural Network, how they are trained. No previous knowledge about Reinforcement Learning is necessary. The first chapter will explain the basics of Reinforcement Learning necessary to understand the papers explained in the final chapters.

Trondheim, 2016-12-14

Alexandre Caulier

Acknowledgment

I would like to thank Humberto Castejon and Arjun Chandra, both researchers at Telenor Research, for their great help to carry out this project. I enjoyed our meetings where I learned a lot. Your passion for Reinforcement Learning has been a great asset for this project.

A.C.

Contents

Preface i

Acknowledgment ii

1 Introduction **2**

1.1 Background 2

1.2 Objectives 4

1.3 Motivations 4

1.4 Limitations 5

1.5 Structure of the Report 5

2 The Reinforcement Learning Problem **6**

2.1 The Agent-Environment Interface 6

2.2 Goals and Rewards 7

2.3 Returns 7

2.4 The Markov Property 8

2.5 Markov Decision Processes 9

2.6 Value Functions 9

2.7 Optimal Value functions 10

3 Q-learning Based Algorithms **12**

3.1 The Q-learning algorithm 12

3.2 The Deep Q-Network algorithm 13

3.2.1 Overview 13

3.2.2 Experience Replay 15

3.2.3 Train and Target Networks 15
3.2.4 Model architecture 15
3.2.5 Training details 16
3.2.6 Algorithm 16
3.2.7 Previous Results 17
3.3 Double Q-Learning 17
3.3.1 Overview 17
3.3.2 Double estimator 18
3.3.3 Algorithm 18
3.3.4 Previous Results 19

4 Exploration methods **20**
4.1 The Greedy policy 20
4.2 The Epsilon-Greedy policy 21
4.3 The Boltzman Q-policy 21
4.4 The UCB1 policy 22
4.5 Value Difference Based Exploration (VDBE) 22

5 Reward Bonuses Exploration Methods **24**
5.1 Model Based Exploration 24
5.1.1 Overview 24
5.1.2 Model of the environment 25
5.1.3 Algorithm 25
5.1.4 Previous Results 26
5.2 Pseudo count 26
5.2.1 Overview 26
5.2.2 Pseudo-count 27
5.2.3 Previous Results 27

6 Advanced Exploration Methods **28**
6.1 The Dueling Architecture 28
6.1.1 Overview 28

6.1.2 Architecture 28

6.1.3 Aggregating the streams 29

6.1.4 Previous Results 30

6.2 The Bootstrapped DQN 30

6.2.1 Overview 30

6.2.2 Model architecture 31

6.2.3 Algorithm 31

6.2.4 Other similar algorithms 32

6.2.5 Sharing data in bootstrapped heads 34

6.2.6 Previous Results 34

6.3 The Prioritized Replay 34

6.3.1 Overview 34

6.3.2 Prioritizing 34

6.3.3 Annealing the bias 35

6.3.4 Previous Results 36

7 Experiments **37**

7.1 OpenAI Gym 37

7.1.1 CartPole-V0 37

7.1.2 CartPole-V1 38

7.1.3 Other environments tried 38

7.2 Experimental results 39

7.2.1 Simple explorations methods 39

7.2.2 Change of architecture 41

7.2.3 Mix 45

8 Summary **47**

Bibliography **48**

Chapter 1

Introduction

Reinforcement Learning is a field that gathers a lot of hopes towards the future of Artificial Intelligence. Indeed, it has proven to be really powerfull to play highly complex games like Chess or the game of Go. Deepmind successfully defeated the Go World Champion with an Artificial Intelligence built with Reinforcement Learning algorithms. A lot of progresses are done almost every week. The number of papers published recently dealing with ways to improve this kind of algorithms is tremendous. These algorithms can also be applied to other scenario than winning a game e.g. optimizing energy consumption in a building. In that sense, Reinforcement Learning is a major source of advancements in AI in the coming years, both in term of fundamental research and applications.

1.1 Background

Problem Formulation

The exploration-exploitation trade-off is an important dilemna when living in an unknown world. When selecting an action to perform, one has the choice between choosing the action that will produce the better outcome based on our current knowledge of the world ("exploitation") or choosing another action that might generate a better or worse outcome but that will allow to improve the knowledge of the world ("exploration").

Exploration is a big challenge in highly dimensionnal environments where a lot of explorations is needed in order to achieve good performance. Different methods have been published in order to explore the world effectively. In this report, the most important ones are going to be explained and experimented on different environments. The goal is to find the best methods to explore the environment quickly in order to decrease the number of steps necessary to achieve good performance.

Literature Survey

- The book Sutton and Barto (1998) called "Reinforcement Learning : Introduction" is the reference when starting Reinforcement Learning. It explains all the basics that are necessary to understand the Reinforcement Learning problem.

- The paper Stadie et al. (2015) called "Incentivizing exploration in reinforcement learning with deep predictive models" introduces a new way to improve exploration, a curiosoty-based exploration. The goal is to create a model of the environment and use it to measure the novelty of a state. This measure allows to give exloration bonuses.

- The paper Bellemare et al. (2016) called "Unifying Count-Based Exploration and Intrinsic Motivation" introduces a generalisation of the counting of already seen states to not enumerable observation spaces. Then, an exploration bonus is given based on this "pseudo-count".

- The paper Mnih et al. (2013) called "Playing Atari with Deep Reinforcement Learning" introduces the DQN algorithm that extends the Q-learning algorithm to not-discrete observation space. It allows to use Reinforcement Learning to solve multiple games like Atari Games. This paper is the starting point to all other papers studied here.

- The paper Van Hasselt et al. (2015) called "Deep Reinforcement Learning with Double Q-Learning" introduces an improvement of the DQN algorithm that tends to less overestimate the value of each action and then increases the overall performance.

- The paper Wang et al. (2015) called "Dueling Network Architectures for Deep Reinforcement Learning" introduces a change in the Neural Network architecture that doen't need any extra-supervision and allow the algorithm to better understand the state-value and action-value resulting in better performances.

- The paper Osband et al. (2016) called "Deep Exploration via Bootstrapped DQN" introduces a method that uses the random intialisation of Neural Networks to achieve diversity and then improving the exploration.
- The paper Schaul et al. (2015) called "Prioritized Experience Replay" introduces a method to learn faster by using more often the samples that are less known by the agent.

What Remains to be Done?

Exploration is a big field where a lot of good explorations policies already exist. The goal of the next years is to improve the way an agent explores an environment in order to be able to solve more and more complex environment.

1.2 Objectives

1. Discover the Reinforcement Learning field and be able to run simple algorithms to achieve good performances in simple environments.
2. Study different exploration methods and find out which one is the best in which scenario.
3. Study change of Neural Networks Architectures or Algorithms that can help the exploration.

1.3 Motivations

From the day Alpha Go defeated Leo Sedol, the current World Champion of Go, I have been attracted by the Reinforcement Learning field. In my mind, it is the first time people started to understand that machines could really defeat humans in complex games. It is a field that I had to study but I did not have any project about it before. Hence, when Humberto talked to me about it, I was really interested because it would allow me to learn the basics of Reinforcement Learning and read state of the arts papers.

1.4 Limitations

My main limitation is a computational one. Indeed, train an artificial intelligence on Atari games take around a week to complete. Thus, I limited my approach to easy environments. Even on this kind of environments, it can take up to an hour to solve an environment. It represents a lot of time when I have to run the same algorithms dozens of time to have useable result (mean of the runs).

1.5 Structure of the Report

The first chapter of this report will introduce the basics of reinforcement learning and the notions necessary to understand the papers. The second chapter deals with the papers that revolutionized the Reinforcement Learning field e.g. the DQN algorithm. The next two chapers explain simple exploration methods and methods based on giving a bonus to the exploration. The next chapter deals with change in the architecture and in the algorithm that lead to improvement in the exploration. Finally, the last part show the environments used for testing and the results obtained from implementing the previous papers.

Chapter 2

The Reinforcement Learning Problem

In this chapter, the basics of Reinforcement Learning are going to be explained as well as the mathematical problem it relies on.

2.1 The Agent-Environment Interface

From a Reinforcement Learning algorithm's point of view, the world is divided between two entities : the *agent* and the *environment*. The agent refers to the intelligence making the decisions. It can also be called the learner. Everything outside the agent is the environment. Actually, the agent is going to interact with the environment by selecting actions that the environment is going to perform. The actions taken will change the state of environment and, once the action is performed, the agent will face a new state of the environment where it will be able to select a new action. The environment will answer the actions by giving reward to the agent at each step. The goal of the agent is to maximize the sum of those rewards. In other words, the agent's aim is to maximize the gain in the long run.

Mathematically, the agent will face a new state of the environment at each timestep t which is a discrete time. A state at the time t is noted $S_t \in S$ where S is the set of the possible states. When facing S_t, the agent will select an action $A_t \in A$ where A is the set of the actions available. One time step later, the agent receives from the environment a reward R_{t+1} and faces a new state S_{t+1}. The Figure 2.1 represents graphically the agent-environment interaction.

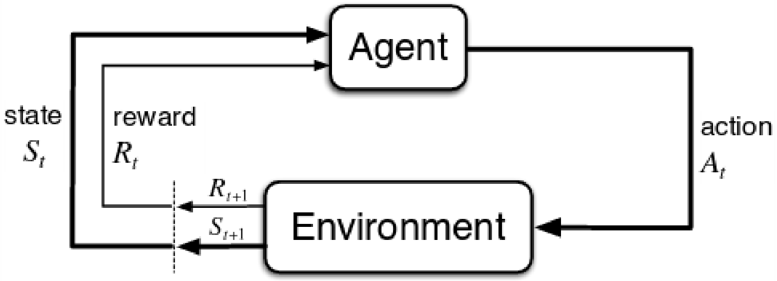

Figure 2.1: The agent-environment interaction in Reinforcement Learning

The probability of chosing each action given a state of the environment is defined by the agent's *policy*. The policy is denoted by π_t, where $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Different Reinforcement Learning methods exists to update the agent's policy given the experience received.

2.2 Goals and Rewards

At each time step, when an action is performed, the environment returns a reward to the agent that is formalized by a variable $R_t \in R$. The goal of the agent is not to maximize the immediate reward, but to maximize the cummulative reward over the long run. It is important to understand that the reward signal is the way of cummunicating to the agent, it shows *what* has to be achieved and not *how* to achieve it. For example, in a chess game, the reward function could be $+1$ for a win, 0 for a tie and -1 for a loss. In this example, we specify what we want the agent to accomplish i.e. winning the game instead of giving points for taking out opponent pieces, which is likely to change the agent's focus to taking out pieces. To actually win the game, the agent has to find a good strategy on his own.

2.3 Returns

So far, we explained that the goal of the agent is to maximize the cummulative rewards over the long run. Let's define this notion formally. If the sequence of rewards received afer the time step t is denoted by $R_{t+1}, R_{t+2}, R_{t+3}...$, then, generally, we are looking to maximize the expected return G defined by the sum of the next rewards that the agent will receive.

$$G_t = \sum_{k=t+1}^{T} R_k$$

where T is the final time step. This approach can be used when the environment interaction is divided in subsequences which are called *episodes*. An episode will end when the environment reach a state called a *terminal state* followed by a reset of the environment. Tasks with episodes of that kind are called *episodic tasks*. Sometimes, it is necessary to have a set of all non-terminal states, denoted S, and a set of all the states, denoted S^+.

Conversely, it is possible that the environment interaction can not be divided into episodes. These kind of tasks are called *continuing tasks*. This means that the final step would be infinite, i.e $T = \infty$. Hence, the goal of our agent would be to maximize an infinite sum. Thus, we are going to use another definition of the return that will work for both episodic and continuing tasks.

The new concept introduced here is the *discounting*. With this idea, the agent is going to maximize the sum of the discounted rewards over the long run given by the formula :

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

where γ $(0 \leq \gamma \leq 1)$ is the discounting rate.

The discounting rate can be understood as a way to estimate the present value of a future reward. In other words, a reward received k steps in the future is worth only γ^{k-1} times what it would be worth if it would be received immediately. If $\gamma < 1$, the infinite sum is bounded. If $\gamma = 0$, the agent tries to maximize the immediate reward, it acts greedily.

2.4 The Markov Property

The state of the environment is a representation of the environenment. It will be the only information available to the agent to select an action. An environment is said to follow the

Markov Property if the next state and the reward received can be predicted using only the current state and action. In other words, a state that follows the Markov Property is memoryless and we don't need to have an history of the states in the past to predict the next state, only the current state and the action selected are usefull.

2.5 Markov Decision Processes

A Reinforcement Learning task that follows the Makov Property is called a *Markov Decision Process* (MDP). If the state and action spaces are finite, then it is called a *finite Markov Decision Process.*

A particular MDP is defined by its state and action sets and by the one-step dynamics of the environment. In other words, all the transition probabilities need to be defined i.e. $p(s'|s,a)$ for all states s, s' and for all actions a. Also, it is necessary to define the mean reward for a triplet (s,a,s') i.e. $r(s,a,s')$. It is the mean reward received by the agent from the environment from starting at the state s, selecting the action a and ending up in the state s'. These quantities, $p(s'|s,a)$ and $r(s,a,s')$, completely specify a finite MDP.

2.6 Value Functions

The *value functions* are used to estimate how good it is for an agent to be in a given state or how good it is for an agent to perform a given action in a given state. The notion of "how good" is defined in terms of expected future rewards. Moreover, the future rewards depends on the actions that we are going to select. Hence, value functions are defined with respect to particular policies.

The value of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in the state s and following the policy π thereafter. It is defined formally as

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

This function is called *the state value function for policy* π.

Similarly, we can define the value function of taking action a in state s under a policy π, denoted $q_\pi(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π :

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

We call q_π the *action value function for policy* π.

The value functions v_π and s_π can be estimated from experience. For instance, if an agent follows the policy π and compute an average of the actual rewards received from the environment, then the average will converge to the state's value $v_\pi(s)$, as the number of times this step is encountered approach infinity. The averages can also be computed with the actions taken. Hence, it will converge to the value function q_π.

These value functions have the remarquable property of satisfying a recursive relationship. The recursive equations are called the *Bellman Equations*. The *Bellman equation for* v_π is given by :

$$v_\pi(s) = \sum_a \pi(a|s)\sum_{s'} p(s'|a)[r(s,a,s') + \gamma v_\pi(s')]$$

It is a relationship between the value of a state and the values of its successor states. The value function v_π is the unique solution to its Bellman equation.

2.7 Optimal Value functions

The goal of a Reinforcement Learning algorithms is to maximize the sum of the rewards received from the environment over the long run. To do so, it has to follow the best possible policy, called the *optimal policy*. Then, it is necassary to be able to order the performance of different policies.

A policy π is defined to be better that or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq \pi'(s)$ for all states s. The policy that is better or equal to all other policies is the optimal policy, denoted π_*.

The *optimal state-value function*, denoted v_*, is defined by

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

for all states s.

The *optimal action-value function*, denoted q_*, is defined by

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

for all states s and for all actions a.

When using the optimal value function, the Bellman equations can be written in a special form called the *Bellman optimality equation.* The expression for the state-value function and the action-value function can be found below.

$$v_*(s) = \max_{a} \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma v_*(s') \right]$$

$$q_*(s,a) = \sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \max_{a'} q_*(s',a') \right]$$

Now, let's assume that we have access to the optimal state-value function and action-value functions because we have gathered a lot of experiences. The issue here is how to find the optimal policy given those functions. Actually, the answer is to select the actions greedily. Indeed, the value functions represent the average reward over the long run. Hence, checking only the one-step consequences of the actions is enough to achieve the optimal policy. In other words, for any state s, the optimal policy is achieved by selecting the action a that maximizes $q_*(s,a)$.

Chapter 3

Q-learning Based Algorithms

In this chapter, the Q-learning algorithm will be explained. We will also deal with the recent papers from Deepmind who introduced the Deep Q-Network algorithm (DQN) and the DoubleDQN. Indeed, they are the first algorithms that successfully defeated a human in several Atari games.

3.1 The Q-learning algorithm

In the Q-learning algorithm, an agent tries to learn the optimal action-value function $q_*(s,a)$. The idea is to gather a lot of transitions from the environment. A transition is an element (s_t, a_t, r_t, s_{t+1}) that represents an experience when the agent was at the state s_t, selected the action a_t and ended up in the state s_{t+1}. It also received a reward r_t from the environment. The core of the algorithm is a value iteration update based on the following rule :

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

The q-value is updated based on the difference between what was predicted and the reward actually received. One can see the term α that is the learning rate (always less or equal to one). It allows to resize the amount of the updates. The pseudo code of the algorithm is the Algorithm 1.

Algorithm 1 Q-Learning algorithm

```
$Q(s,a)$ initialized randomly
Intialized the learning rate $\alpha$, the decay rate $\gamma$ and the $\epsilon$
for each episode do
    S = starting state of the environment
    for each time step in the episode do
        Select the action $a$ with an $\epsilon$-greedy policy from $Q$
        Perform the action $a$. Observe the reward $r$ and the new state $S'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha \left[ r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$
        $S \leftarrow S'$
    end for
end for
```

One can note that the term ϵ-greedy policy has not been explained before. The idea of this policy is to select with a small probability ϵ a random action and with a probability $1-\epsilon$ the best action according to the Q-values. In other words, the agent is going to act greedily with a probability $1-\epsilon$.

A limititation of this algorithm is, that it can be applied only to environment with discrete observation space and discrete action space. Now, let's move on to an improvement of this algorithm that will allow to use Q-learning on not enumerable observation spaces.

3.2 The Deep Q-Network algorithm

3.2.1 Overview

The Deep Q-Network algorithm, referred as DQN, was introduced for the first time in the paper Mnih et al. (2013). A more detailed version was also publised here : Mnih et al. (2015). In these papers, Deepmind researchers show how they adapted the Q-learning algorithm to train an agent to learn from the images generated by an Atari emulator. In other words, the agent receives an image as input and learn from it to predict the right action. This is particularly amazing because before, reinforcement learning agents had achieved some successes in a few domains that are fully observable and with low dimensional space. With this novel approach, they were able to train reinforcement learning agents that outperform humans in not less than 49 Atari games just by learning from raw-pixels.

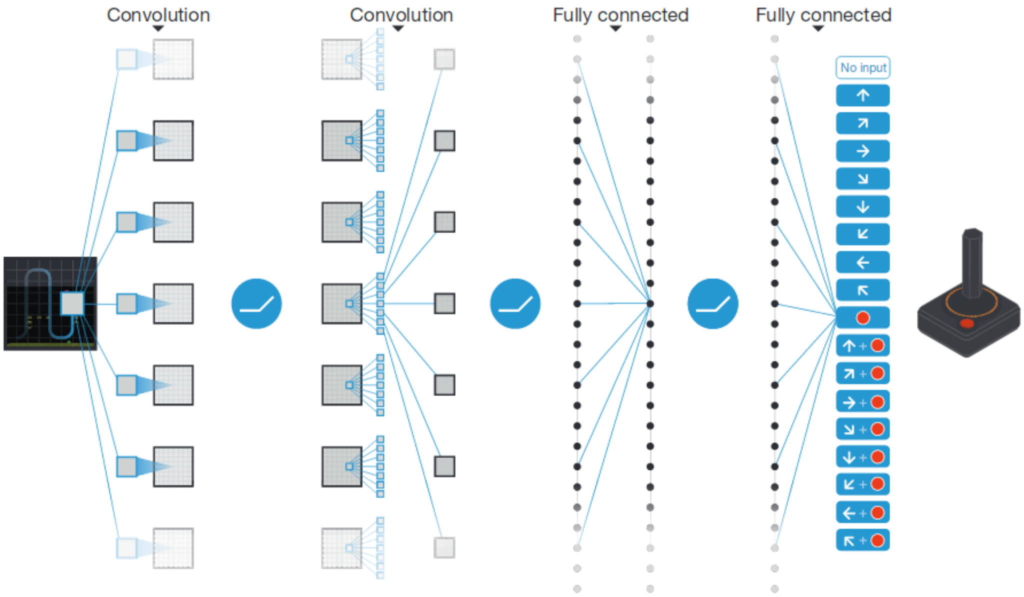

Figure 3.1: Deepmind model learning from images and generating an action

The goal of an agent is to select actions that will maximize cumulative future reward. In order to do that, the optimal action-value function, denoted Q^*, has to be known. One of the main idea of this paper is to use a deep neural network to approximate Q^*.

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{k=0}^{+\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a, \pi \right]$$

In the Reinforcement Learning field, it is well known that using a non-linear function to approximate the optimal Q function may cause the agent of be unstable or to diverge. There are several reasons like the correlations in the sequence of observations and the fact that a small update in the Q-network may significantly change the policy. Indeed, like most of the time, the action selected is the one that maximizes $r + \gamma \max_{a'} Q(s', a')$, then a change in Q can totally change the behavior of the agent. To resolve these issues, Deepmind researchers introduced two key ideas that successfully allowed a neural network to be used to approximate the optimal action-value function. The first one is called the experience replay. Every time step, the experiences are stored in a data-set and, for training, a mini-batch is generated by uniformally sampling from the replay. The second one is to use two networks, one for the training and one the predicting the q-values. The target network is updated periodically which allows to decrease unstability.

3.2.2 Experience Replay

The DQN stores the agent's experiences at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$, in a data-set $D = e_1, ..., e_N$ called the replay memory. It has a fixed size, so when it is full, samples are randomly removed. When the neural network is trained, a fixed amout of experiences is sampled uniformally from the experience replay.

Using experience replay has multiple advantages. Firstly, it allows to use the same experience in potentially several weights updates which allows for greater data efficiency. Secondly, learning from consecutive samples is ineficient because there is a strong correlation between the samples. Indeed, two consecutive images generated by the emulator will be necessary close because we can move from a state to the other in a single action. It is better to use samples with high diversity to train a Neural Natwork because it increases the variance of the updates.

3.2.3 Train and Target Networks

Another modification to the original Q-learning algorithm intends to increase the stability of the algorithm after every update. The idea is to use two neural networks in the algorithm. The first one is the Train Network, denoted $\hat{Q}$, it is the one where the weights updates will be applied. The second one is the Target Network, denoted Q, this one will be used to predict the q-values in the learning update.

A constant parameter C has to be specified. Every C time steps, the weights of the Train Network will be copied and will be the new weights of the Target Network. Using this methods allows a better stability of the algorithm and also decreases significantly the oscillations.

3.2.4 Model architecture

The Deep Neural Network approximating the Q-function is referred as the Q-Network, denoted by $Q(s, a; \theta)$ where θ are the weights of the network. There are several different architectures possible for the network. Previous approaches used the states and the actions as inputs for the network and it produced a single number wich represents the q-value for this particular state and action. The main drawback of this approach is the necessity to compute a different forward pass for each available action in order to have the q-value of each action in a particular state. It results in a cost that scales linearly with the number of actions.

DeepMind researchers introduced a new architecture that allows to have all the q-values in a particular state in a single forward pass. The idea is to give the Q-network a state in input and it will produce exactly number of actions outputs that will be the approximated action-values of each available actions for this particular state. Following this structure, the input layer must have the same shape as the shape of one state of the environment and the output layer must have the same number of neurons as the number of actions available at each step.

3.2.5 Training details

Every time step, a mini batch is generated from the experience replay ($U(D)$). Then the Q-learning update at iteration i is computed using the following error-function :

$$L_i(\theta_i) = \mathbb{E}_{(s,a,,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right]$$

where θ_i are the weights of the Q-network at iteration i and θ_i^- are the weights of the Target Network.

3.2.6 Algorithm

The pseudo code of the algorithm is written in the Algorithm 2. At each time step, a feed-forward pass is performed over the Q-network to estimate the Q-values of each action. Then, an epsilon-greedy policy is used to select an action which is performed by the agent. The agent receives a reward and arrives in a new state. This new experience can now be stored in the experience replay. The training Q-network is trained by sampling a mini-batch from the experience replay. Finally, the weights of the target Q-network are updated periodically. And it goes like this for every step. Of course, if the agent is in a terminal state, the environement is reset.

Algorithm 2 Deep Q-learning with Experience Replay

```
Initialilize replay memory $D$ to capacity $N$
Intialize action-value function $Q$ with random weights $\theta$
Intialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
Intialized the learning rate $\alpha$, the decay rate $\gamma$ and the $\epsilon$
for episode = $1, M$ do
    $s_1$ = starting state of the environment
    for time step $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        Select action $a_t = \arg\max_a Q(s_t, a; \theta)$
        Perform the action $a_t$. Observe the reward $r_t$ and the next state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$
        Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q(s_j, a_j; \theta)\right)^2$
        Every $C$ steps reset $\hat{Q} = Q$
    end for
end for
```

3.2.7 Previous Results

This new algorithm achieves state of the art performance in almost all the Atari Games it was tested on by the writers. Moreover, they use exactly the same architecture and the same hyper parameters for all the experiments showing how general this new algorithm is.

3.3 Double Q-Learning

3.3.1 Overview

The Double Q-Learning algorithm was introduced by the paper Van Hasselt et al. (2015). According to this paper, the use of the max operator to determine the q-values of each action given a state can cause large over-estimations of the actions values. Indeed, the maximum expected value is approximated by using the maximum operator which has a positive bias. It induces a large peformance penalty because it is necessary to explore more to have a better estimate. The researcher proposes a double estimator method to find an estimate of the maximum expected value that sometimes underestimates rather than overestimates the maximum expected value.

One can note an important difference between the heuristic exploration technique of optimism in face of uncertainty and the overestimation bias. Optimism is about improving the expected q-values of not explored actions whereas Q-learning can overestimate actions that have been tried a lot of times. The result of overestimation is an unrealistic estimator.

3.3.2 Double estimator

In the paper, it is proven that combining two differents sets of estimators results in a better estimate than using only one. Here, the estimators can be understood as the Q-networks trained to predict the Q-values. Thus, using multiple Q-networks would allow to decrease the bias of the estimation. The natural candidate for this job is the Target Network.

3.3.3 Algorithm

In this new approach, there are two Q-networks, denoted Q^A and Q^B. When training a network, the value predicted by the other Q-network is used for predicting the next state and then computing the error. The full algorithm is written in Algorithm 3.

Algorithm 3 Double Q-learning

```
Initialize $Q^A$, $Q^B$, $s$
repeat
    Choose $a$, based on $Q^A(s,.)$ and $Q^B(s,.)$, observe $r$, $s'$
    Choose randomly between UPDATE(A) and UPDATE(B)
    if UPDATE(A) then
        Define $a^* = \arg\max_a Q^A(s', a)$
        $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \gamma Q^B(s', a^*) - Q^A(s,a))$
    else if UPDATE(B) then
        Define $a^* = \arg\max_a Q^B(s', a)$
        $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s', a^*) - Q^B(s,a))$
    end if
until end
```

At each time step, an action has to be chosen based on the estimated q-values predicted by both q-networks. The paper advises to compute a mean of the q-values and then select the action with an epsilon greedy policy. It is also possible to use another exploration policy.

One can see that the update rule for the Q^A is using the value of $Q^B(s', a^*)$. In other words, the maximum expected value approximated by the other network. A simple DQN would use $Q^A(s', a^*)$. It is also important to note that each Q-network have their own experience replay and a mini-batch is sampled from the network's experience replay to perform the learning at each time step.

3.3.4 Previous Results

This new algorithm is the first one without a positive bias in the Reinforcement Learning field. It allows to train a lot quicker than DQN algorithm to achieve better or equal performance.

Chapter 4

Exploration methods

The exploration-exploitation trade-off is an important dilemna in Reinforcement Learning. The dilemna is between choosing what you know and getting a reward close to what you expect ('exploitation') and choosing something you aren't sure about and possibly learning more or less ('exploration'). In this chapter, we will deal with this trade-offs by studying different exploration methods that are deeply used in the Reinforcement Learning field.

4.1 The Greedy policy

The greedy policy is the policy that never explores and always exploits the knowledge previously acquired. When being in a state s, the idea is to select the action a that maximizes the action-value function $Q(s,a)$. In other word, we have the formula :

$$a = \arg\max_{a'} Q(s,a')$$

The main limitation of this policy is to never look for getting better rewards. Indeed, in a particular environment, it could be necessary to collect a small reward in order to have a bigger one at the next time steps. Thus, with this policy, the optimal policy will almost never be achieved. It is then necessary to look for other policies that will explore the environment looking for better results over the long run.

4.2 The Epsilon-Greedy policy

The epsilon-greedy policy is the idea of exploring with a probability ϵ and exploiting with the probability $1-\epsilon$. When exploring, a random action is selected. The random generator is uniform and not based on the q-values.

This policy works pretty well in practice. Unfortunately, it is not adapted for complicated environment. For example, let's imagine an environment where it is necessary to perform a big number of correct actions before receiving the big rewards. Then, with this policy, like the actions are selected randomly, the probability of receiving the big reward will decrease exponentially toward 0. Thus, this policy is not adapted for all the environments.

One can see that if we use a constant ϵ, then the agent will still be exploring even when the optimal action-value functions would have been found. Hence, it is necessary better to decrease the ϵ over time. It can be undestood as, the agent knows better and better its environment over the time and the need of exploring can be decreased. Multiple methods exist to decrease the probability of exploring, the most used are exponential or linear decay at each time step or at each end of episode.

4.3 The Boltzman Q-policy

One of the drawbacks of the epsilon-greedy policy is to sample uniformaly the actions when exploring. The Boltzman Q-policy uses the current q-values to sample the actions. Hence, an action with a bigger q-value will have a bigger probability to be selected than another. More formally, it is the softmax of the q-values that is used. Given a state s, the probability to select the action a is given by:

$$p_s(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{a'} e^{\frac{Q(s,a')}{T}}}$$

The term T is called the Temperature and allows to resize the rewards over the probabilities.

One drawback of this method is that it never stops exploring by itself. Indeed, we have to specify a time step, when the agent won't be using this policy anymore but the greedy one. It is clear, that if we have two actions that bring two different rewards but with one that is bigger, then, there will always be a probability of selecting the wrong action even when the environment is perfectly known.

4.4 The UCB1 policy

The policy we deal with here is called UCB1 and can be summed up by the principle of *optimism in face of uncertainty*. That is, despite our lack of experience in what actions are best in each state, we will construct an optimistic guess as to how good the expected payoff of each action is. If our guess is wrong, then our optimistic guess will quickly decrease and we'll be compelled to switch to a different action. But if we pick well, we'll be able to exploit that action and incur little regret. In this way we balance exploration and exploitation.

One way to do that is to use the formula written below which is based on the "Chernoff-Hoeffding inequality". Being in state s, the agent will select action s with the following formula:

$$a = \arg\max_{a'} Q(s,a') + \sqrt{\frac{2\log(t)}{n_{a'}}}$$

where t is the current time step and $n_{a'}$ is the number of times the action a' has already been selected.

4.5 Value Difference Based Exploration (VDBE)

The basic idea of VDBE is to expend the ϵ-greedy policy by controlling a state dependant exploration probability $\epsilon(s)$. The desired agent is intended to explore more in not known states and almost always exploit in well-known states. The formula used is:

$$\epsilon_{t+1}(s) = \delta f(s_t, a_t, \sigma) + (1-\delta)\epsilon_t(s)$$

where σ is a positive constant called *inverse sensitivity* $(0 \leq \sigma < 1)$, a parameter determining the influence of the selected action on the exploration rate. δ is a constant parameter that is often fixed to the inverse of the number of actions available as it performs well in practice. f is a function based on a (softmax) Boltz-mann distribution of the value-function estimates that is computed after each time step and defined as follow:

$$f(s,a,\sigma) = \left| \frac{e^{\frac{Q_t(s,a)}{\sigma}} - e^{\frac{Q_{t+1}(s,a)}{\sigma}}}{e^{\frac{Q_t(s,a)}{\sigma}} + e^{\frac{Q_{t+1}(s,a)}{\sigma}}} \right|$$

The main drawback of this approach is the necessity to have an enumerable observation space because the ϵ is state dependant. It could be tried on not enumerable observation space by discretising the space.

Chapter 5

Reward Bonuses Exploration Methods

In this chapter, we are going to deal with several models that use rewards bonuses to achieve efficient exploration. In other words, they add a small bonus to the reward returned by the environment to change the action that will be selected by the policy like epsilon greedy policy for instance.

5.1 Model Based Exploration

5.1.1 Overview

The paper Stadie et al. (2015) introduced a new way to deal with the tradeoff exploration/exploitation. If one assumes that an agent will tend to choose the action with the best reward, it is possible to guide exploration by adding a bonus to the reward based on the novelty of the states. In other words, the bonus will measure how well the state is known by the agent. It can be accompplished using the following reward function:

$$\mathcal{R}_{bonus}(s,a) = \mathcal{R}(s,a) + \beta\mathcal{N}(s,a)$$

where $\mathcal{R}(s,a)$ is the reward sent by the environment, β is a positive constant and $\mathcal{N}$ is a novelty function with values between 0 and 1. In this paper, they learn a model of the environment to able to estimate how well a state is known.

5.1.2 Model of the environment

Let $\mathscr{M}$ be a function that takes in input an action and a state. $\mathscr{M}$ will have a single output that is the next state predicted by our model of the environment. The model of the environment is not known and can be often a very complex one. That's why, the agent will have to learn the model dynamics with his own experience without no prior knwoledge. To do that, a Neural Network can be used for the function $\mathscr{M}$ to approximate the model dynamics.

Given a state, s_t, and an action a_t, $\mathscr{M}(s_t, a_t)$ will be the next state predicted. It is then to be compared with s_{t+1}. That's why the predicition error is given by:

$$e(s_t, a_t) = \parallel s_{t+1} - \mathscr{M}(s_t, a_t) \parallel_2^2$$

This error, is also normalized to avoid huge values using the following formula:

$$\bar{e}_T = \frac{e_T}{\max_{e \leq T} e_t}$$

where e_T is the error at time T and $\bar{e}_T$ is the normalized error at time T. We can now write the new reward function:

$$\mathscr{R}_{bonus}(s, a) = \mathscr{R}(s, a) + \beta \left(\frac{\bar{e}_t(s_t, a_t)}{t * C} \right)$$

where C and β are two positive constants. The idea behind this formula is that the precision of the model dynamics will improve with the time.

5.1.3 Algorithm

The pseudo code for the algorithm is available in Algorithm 4. The approach described in this paper can be used with any reinforcement learning algorithm. At each time step, an experience $(s_t, a_t, s_{t+1}, \mathscr{R}(s_t, a_t))$ is observed. The model of the environment is used to predict the next state and with it the error of prediction is computed. This allows to compute the new reward that will be stored in the memory bank. At this point, any reinforcement learning algorithm can be used (DQN for instance). And periodically the model of the environment is updated.

Algorithm 4 Reinforcement learning with model prediction exploration bonuses

```
Initialize $\max_e = 1$, EpochLength, $\beta$, $C$
for iteration $t$ in $T$ do
    Observe $(s_t, a_t, s_{t+1}, \mathscr{R}(s_t, a_t))$
    Compute $e(s_t, a_t) = \| s_{t+1} - \mathscr{M}(s_t, a_t) \|_2^2$ and $\bar{e}(s_t, a_t) = \frac{e(s_t, a_t)}{\max_e}$
    Compute $\mathscr{R}_{bonus}(s_t, a_t) = \mathscr{R}(s, a) + \beta\left(\frac{\bar{e}_t(s_t, a_t)}{t * C}\right)$
    if $e(s_t, a_t) > \max_e$ then
        $\max_e = e(s_t, a_t)$
    end if
    Store $(s_t, a_t, \mathscr{R}_{bonus})$ in a memory bank $\Omega$
    Pass $\Omega$ to the reinforcemement learning algorithm to update $\pi$
    if $t \mod EpocLength == 0$ then
        Use $\Omega$ to update $\mathscr{M}$
    end if
end for
return optimized policy $\pi$
```

For high dimensional states, it is necessary to encode them. A natural way to do it is to use an auto-encoder to decrease the dimension of each state.

5.1.4 Previous Results

This new approach has been tested on the particularly complex Atari benchmark and improves the state-of-the-art on several games. It uses a learned model of the environment to encourage the agent to visit new states.

A drawback of this approach is that it can't be used in stochastic environments. Indeed, misprediction might be caused by stochastic reward and not only by errors in the model of the environment.

5.2 Pseudo count

5.2.1 Overview

A new approach for exploration was introduced in the paper Bellemare et al. (2016). Some environments can be called tabular environments because it is possible to enumerate all the states available and then remember which state has been visited and how many times. The idea of this paper is to generalize the count method for enumerable observation spaces to continuous observation spaces by introducing a notion called the *pseudo-count*. It is then possible to apply a reward bonus in function of this pseudo-count.

5.2.2 Pseudo-count

The pseudo-count is denoted by $\hat{N}_n(s)$ where s is a state. They use sequential density models to estimate uncertainty and then the value of the pseudo-count. This method can be applied to high dimensional observation space.

The exploration bonus used in this paper is linked to the pseudo-count and is defined by:

$$R_n^+(s,a) = \beta(\hat{N}_n(x) + \alpha)^{-1/2}$$

where α and β are two small constants. It is now possible to use a Q-learning based algorithm except that the reward of each action will be the sum of the reward returned by the environment and the exploration bonus defined above.

5.2.3 Previous Results

The researchers focused their experiments on one Atari Game that is known to be the hardest one, MONTEZUMA'S REVENGE. The game is divided into three levels, each one composed of 24 rooms. In each room, a lot of challenges have to be faced in order to access the next room. Thus, the number of rooms reached by an agent is a good indicator of the exploration level of an algorithm.

The Double DQN algorithm is able to access only two rooms whereas this new algorithm based on pseudo-count is able to access 15 rooms. In that sense, the pseudo-count approach seems very promising because it induces a very efficient exploration. Moreover, it can be combined with every reinforcement learning algorithm.

Chapter 6

Advanced Exploration Methods

In this chapter, instead of changing the policy used, we arre going to perform changes in the algorithm and in the neural network architecture in order to achieve better exploration. We are going to deal with three methods that have been published at most one year ago.

6.1 The Dueling Architecture

6.1.1 Overview

The Dueling Architecture comes from the paper Wang et al. (2015) showed at the ICML 2016. It is a change in the Q-Network architecture that doesn't need any extra-supervision i.e it is invisible from outside the network because it doesn't change the input or the output of the network. It can then be used in any reinforcement learning algorithm. The key idea behind this new architecture is that for many states, it is unecessary to evaluate the value of each action choice. For example, in a game where our agent has to avoid collision with other cars, knowing whether to move left or right only matters when a collision is eminent.

6.1.2 Architecture

The proposed architecture, called the *dueling architecture*, divides the action values and (state-dependent) action advantages. A representation of the dueling architecture is shown in Figure 6.1. The idea is to keep the first part of the neural network as it is. And add two parrallel fully connected layers (two streams) that are going to represent the state value and the action adavntage (value for telling how good it is to take this action). Finally, they are combined to produce the q-values.

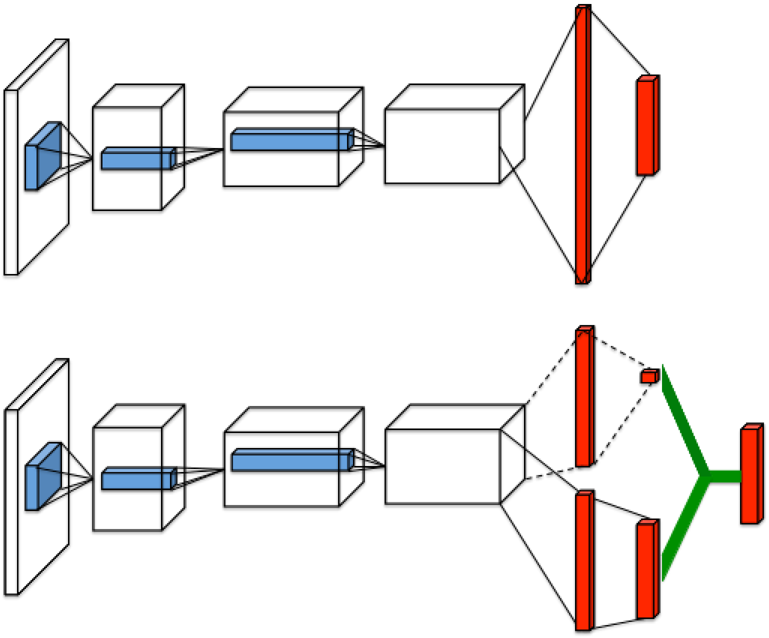

Figure 6.1: Dueling Architecture

The layer representing the state value is composed of a single neuron and the layer representing the actions advantages is composed of number of actions neurons because each action has a different advantage estimate.

6.1.3 Aggregating the streams

Before the split to differentiate state-value and action-value, there is a network with weights denoted θ. The state value function approximated by the first stream with the weights β is denoted $V(s;\theta,\beta)$ and the actions value function approximated by the second stream with the weights α is denoted $A(s,a;\theta,\alpha)$.

Combining V and A by summing them is tempting but works poorly. Indeed, there is a lack of identifiabilty in the sense that given Q, it is not possible to recover Q and A uniquely. To address this issue, we can force the advantage function estimator to have zero advantage at the chosen action. That is, we substract by the maximum advantage possible.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a' \in \mathscr{A}} A(s,a';\theta,\alpha) \right)$$

With this module, and in accordance with Bellman optimality equations, the optimal q values (i.e. optimal action's value) will be equal to the optimal value of the state, thus making the advantage of the optimal action zero. For other actions, the advantage will be negative. It is exactly the purpose of this change of architecture.

It is important to note, that this computation is done internally by the neural network. It is not an additionnal step performed by the algorithm. Hence, there is no extra-supervision needed for this change or architecture and it can be used with every already existing reinforcement learning algorithms like DQN because the Q-network is updated using a gradient descent algorithm.

6.1.4 Previous Results

The main advantage of this new architecture is that it allows to compute efficiently an approximate of the state-value function. At each update, tha stream approximating the value-function is updated. In the single stream approach, only the value for a single action is updated. All the others remained untouched. Thus, the value function is learned a lot quicker. In the experiments lead by the writters, the advantage of the dueling architecture over the single stream approach grows with the number of actions available.

6.2 The Bootstrapped DQN

6.2.1 Overview

The boostrapped DQN was introduced in the paper Osband et al. (2016) which was published in NIPS 2016. The purpose of the algorithm is to use a strategy of exploration over multiple time steps and not a single one like it is the case if we use an epsilon greedy policy. In other words, this paper introduces a new efficient reiforcement algorithm for complex environments.

6.2.2 Model architecture

In the original paper from DeepMind, a Deep Q-Network is used to approximate the action-value function $Q(s,a)$. The neural network architecture used is a neural netork with the shape of a state in input and an output layer with number of actions neurons. The architecture proposed here, is to use a shared network for the first part of the network. And then use multiple fully connected laeyers (referenced as "heads") so that each head can output all the q-values. The architecture is shown in Figure 6.2.

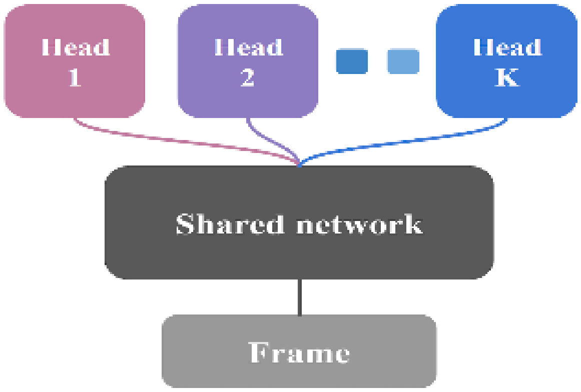

Figure 6.2: Shared network architecture

Each head is trained only on a sub-sample of the experience lived by the agent. In other words, each time an experience is stored in the experience replay, a mask is added to to specify which heads will use this experience for training. This kind of architecture can be implemented easily and trained efficiently in a single forward/backward pass. Indeed, one can add a mask that will act as a data dependant dropout and the mask is to be stored in the experience replay with every experience.

6.2.3 Algorithm

The full algorithm is written in Algorithm 5. There are a few significant differences with the original DQN. Firstly, at each episode, an head is selected randomly. This head will be used to predict the q-values during the whole episode. Then, at each time step, it is exactly like the simple DQN using a greedy-policy. The action with the best q-value according to the previously selected head is performed by the environment. Finally, at the end of each time step, in addition to the classic experience stored, a mask is sampled using a Bernouilli law and added to the transition to be stored in the experience replay. This mask is composed of K (number of heads) booleans. If the ith boolean is equal to one then the ith head will use this experience to train.

Algorithm 5 Bootstrapped DQN

```
Input: Value function networks $Q$ with $K$ outputs $\{Q_k\}_{k=1}^{K}$. Masking distribution $M$.
Let $B$ be a replay buffer storing experience for training
for each episode do
    Obtain initial state from envoronment $s_0$.
    Pick a value function to act using $k \leftarrow \text{Uniform}\{1,...,K\}$
    for step $t = 1,..$ until the end of the episode do
        Pick an action according to $a_t \in \arg\max_a Q_k(s_t, a)$
        Receive state $s_{t+1}$ and reward $r_t$ from environment, having taking action $a_t$.
        Sample bootstrap mask $m_t \leftarrow M$
        Add $(s_t, a_t, r_{t+1}, s_{t+1}, m_t)$ to replay buffer $B$.
    end for
end for
```

6.2.4 Other similar algorithms

One can note the choice of selecting the head for the whole episode and not at each time step and the choice of relying only on the selected head for predicting the q-values and not even looking at the q-values predicted by the other heads. In that sense, the Bootstrapped DQN algorithm can be compared with two other algorithms. The first one is Thomson Sampling which is the same as Boostrapped DQN except that a new head is sampled every time step. And the second one is Ensemble DQN which is the same as Boostrapped DQN except it uses an ensemble policy to select the next action to perform. A common ensemble policy is the majority vote. In other words, the network predict the q-values with each head, the action selected will be the action with the higher q-value in most of the heads.

In order to compare the exploration performance of the different algorithms, an environment needing deep exploration is necessary. We can think of a chain of state where only the last state will give a reward and all the other states won't give any reward. Thus, in order to have a global reward over an episode greater than zero, the agent has to reach the last state i.e the deeper one and then needs deep exploration. A lot of real environment like this exist in the real world. For example, an environment where an agent has to control a lander to successfully land on a planet. The environment can be implemented such as the only reward given will be when the lander successfully lands.

In the paper, they used this kind of environment to compare the algorithms. They compare the number of episodes necessary to solve the environment in function of the number of states available between the initial state and the final one (the only one returning a reward). Their results are shown in Figure 6.3. An environement is considered solved when the agent had received the optimal reward (reach the deep state here) one hundred times. The red dots are unsolved environments.

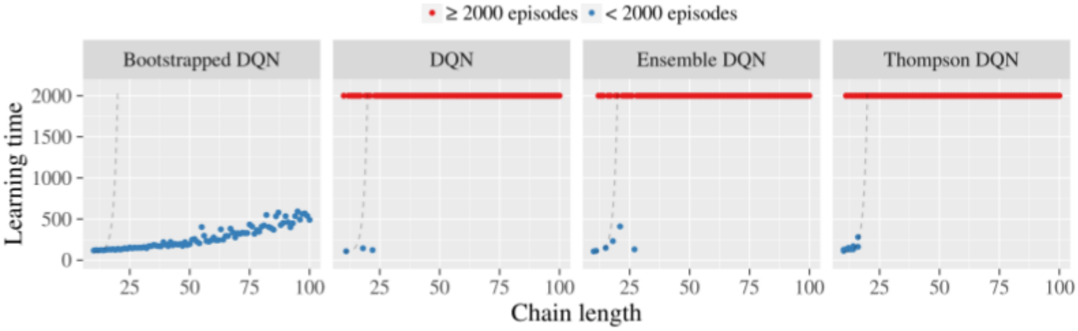

Figure 6.3: Experiments lead in the paper Osband et al. (2016)

In their experiments it is clear that the Bootstrapped DQN is the only algorithm able to solve the environment when the chain length grows. A possible explanation to this phenomenon is that the Boostrapped DQN is the only algorithm using deep exporation. Indeed, deep exploration means exploration which is directed over multiple time steps. In that sense, like the the Boostrapped DQN samples a new head at each episode and not at each time step (Thomson Sampling), the Bootstrapped DQN is able to achieve deep exploration and then explores a lot better than the Thomson Sampling. It allows to have a strategy of exploration over a whole episode and not over a single time step. The second one is that Bootstrapped DQN uses the random initialisation of its heads to induce diversity which allow the agent to direct its exploration at potentially informative states and actions even if they are not immediately rewarding.

6.2.5 Sharing data in bootstrapped heads

In the experiments lead by the researchers, we can see that sharing the experience between all the heads can improve the results of the algorithm. Indeed, in the Boostrapped DQN algorithm, at the end of each time step, a mask is sampled using a Bernouilli law to determine which head will use the current experience. Here, they show that using a constant mask filled with one improved the results in their experiments. It means that all the heads are updated with all the experiences. Indeed, like the heads are initialized randomly, even if they are trained with the same experiences, they will produce different q values. Moreover, it improves the speed of the algorithm because it avoids to compute the Bernouilli law at each time step.

6.2.6 Previous Results

This paper introduced a new algorithm able to achieve deep exploration. In other words, it is an efficient reinforcement learning algorithm for complex environments. This algorithm has also be tried on several Atari Games and it is able to learn much faster than the simple DQN. Also, it improves the state of the art on various games.

6.3 The Prioritized Replay

6.3.1 Overview

The Prioritized Experience Replay was introduced for the first time by DeepMind researchers in the paper Schaul et al. (2015). The main idea is that in the original DQN algorithm, the experiences are sampled uniformally from the experience replay. Here, the researchers propose to sample the transitions based on the knowledge it could bring to the agent. Thus, it will improve the efficiency of the algorithm.

6.3.2 Prioritizing

In order to sample experiences from the experience replay not uniformally, we have to give priority to each experience. Idealy, the priority should be how much the agent can learn from each transition. Even though we can't know it exactly, it is possible to approximate it using the Temporal Difference error (TD-error). This measure is really interesting for Q-learning based algorithms because they are already using this value and then no extra computation is needed. Also, it won't be possible to update the priority of each experience at each time step because it would take too long. Then, only the replayed transitions will have their priority updated.

However there are several issues with this metric. Firstly, experiences that have a TD-error of zero will never be replayed even if recent updates in the network changed the TD-error associated to the transition. Secondly, this metric is not adapted for stochastic reward (the same action in the same state can produce different rewards). Finally, at the beginning of training there are huge TD-errors and during th training, errors decrease slowly. Then, initial high errors transitions will be replayed frequently which can lead to overfitting.

To overcome this issue, the probability of sampling is defined by:

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

where $p_i > 0$ is the priority of the ith transtion and α defines how much prioritization is used. Indeed, $\alpha = 0$ is the uniform case.

In the paper, they selected two variants for the priority of a transition. The first one is called proportional prioritization where $p_i = |\delta_i| + \epsilon$ and ϵ is a small constant avoiding a zero priority and δ_i is the TD-error. The second one is called rank-based prioritization where $p_i = \frac{1}{rank(i)}$ where $rank(i)$ is the rank of the ith transition when the experience replay is sorted according to $|\delta_i|$.

6.3.3 Annealing the bias

Sampling transitions not uniformally creates an issue in the convergence of the algorithm. Indeed, it changes the distrubition and then the policy the algorithm will converge to. This bias can be corrected using the following formula called Importance Sampling (IS) :

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

that totally compensates for non-uniform probabilities $P(i)$ if $\beta = 1$.

Importance sampling can be used in a Q-learning based algorithm by updating the weights with the error $w_i\delta_i$ instead of δ_i. Moreover, like we don't control the value of the importance sampling, the value of w_i is also normalized to only scale downward. In other words, w_i is divided by $\max_j w_j$.

Finally, a small bias is usefull at the beginning of training but it must be removed to converge to the right policy. In order to do that, a schedule can be defined to change the value of β at each time step. A common one is to linearly anneal β from its initial value β_0 to 1.

6.3.4 Previous Results

The Prioritized Experience Replay can be used with any DQN based algorithm. In their experiments, DeepMind researchers achieved a new state of the art in Atari Games. Moreover, they reached a mean speed up learning by a factor of two. Finally, both variants (proportional and rank based) achieved almost the same speedups.

Chapter 7

Experiments

In this chapter, the test environments will be described. Also, the implementation choices and results will be shown. Thus, conclusion about the performance of each algorithm will be explained.

7.1 OpenAI Gym

OpenAI Gym is a library for evaluating Reinforcement Learning algorithms developped by OpenAI. It supports a lot of different environments like teaching an agent to walk or to play games like Pong or Go. It allows to compare the performances of any reinforcement learning algorithms with anyone in the world.

7.1.1 CartPole-V0

CartPole is a very simple environment and is the starting point to every test. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over.

The agent has to control the cart during 200 time steps and prevent the pole from falling. This input is composed of four decimal values (they are bounded) and there are two actions available any time. We can acknowledge that the physic of the world is quite simple. Hence, it is a really good environment to start testing an algorithm.

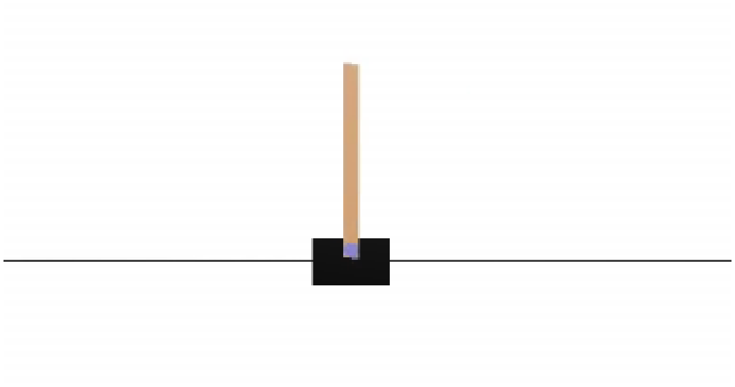

Figure 7.1: CartPole-V0 environment

7.1.2 CartPole-V1

This environement is the same world of the CartPole-V0. But, the pole has to stay on the cart for 500 time steps and not 200 like it was the case before. It is a lot more complicated for the agent to solve this environment because, here, the cart has to learn to not move and stay at the middle else the pole would fall. Indeed, in the above environement, if the cart moves slowly on a side, the pole won't fall before the end of the 200 time steps. In that way, this environment is much more complicated to learn for an agent.

7.1.3 Other environments tried

In addition to these two environments the Game of Go in a small board of size nine times nine was experimented with. In this environement, the agent has to choose between more than 80 actions possible and it has to defeat the opponent which behaviours follows a constant AI. I run a DoubleDQN on this environment. Unfortunately, 4 hours after the launch, the agent hadn't won a single game. Due to the training time, I decided to remove this environment from the test set because I need to run a big number of times each algorithm on the same environment in order to be able to evaluate it.

I also tried solving Atari environment but for the same reason, I decided to remove it too. Indeed, after a whole night of training, the agent had barely learned anything. After some research, I discovered that Deepmind needed a week of training to solve an Atari Games using a DQN algorithm.

7.2 Experimental results

I implemented several algorithms showed in this report to test their performances. All my implementations are written in Python and use the TensorFlow library when a Neural Network is needed.

7.2.1 Simple explorations methods

The purpose of this first set of experiments is to use a DoubleDQN algorithm with different simple exploration methods. Indeed, I implemented from scrach a Double Q Learning algorithm. Then, I wanted to see the difference of performances when using an epsilon greedy policy or a boltzman-Q policy or a UCB1 policy on top of the Double Q Learning to select the actions to perform.

Implementation

The first algorithm to implement was the Double Q Learning algorithm. I adapted the Double Q Learning to Deep Neural Networks. As in the paper Van Hasselt et al. (2015), I used two Q-Networks where each one has his own experience replay. Since my agents did not learn from raw images, the Q-Networks were only composed of fully connected layers instead of convolutional layers. There is no Target Network in my implementation. Finally, I used the mean of the q-values predicted by each Q-Network to select the next action.

Regarding the implementation of the Boltzman Q policy, I had to modify the algorithm implementation to prevent overflow. Indeed, in the Boltzman Q Policy, it is needed to compute the exponential of the q values to have the probabilities. But, in CartPole, a reward of one is given at each time step which lead some q-values to be at more than one hundred. Then, taking the exponential of such big values caused overflow in the memory because the math library was not able to handle such big numbers. To solve it, before computing the exponential, I substracted the minimum of the q-values. It was enough to solve the problem here.

CartPole-v0

The first environment I tried was CartPole-v0. I used a Neural Network with a single hidden layer with 150 neurons. The Figure 7.2 shows the result obtained for the three policies on this environment. The rewards shown here are the mean of my five best run over 20 runs.

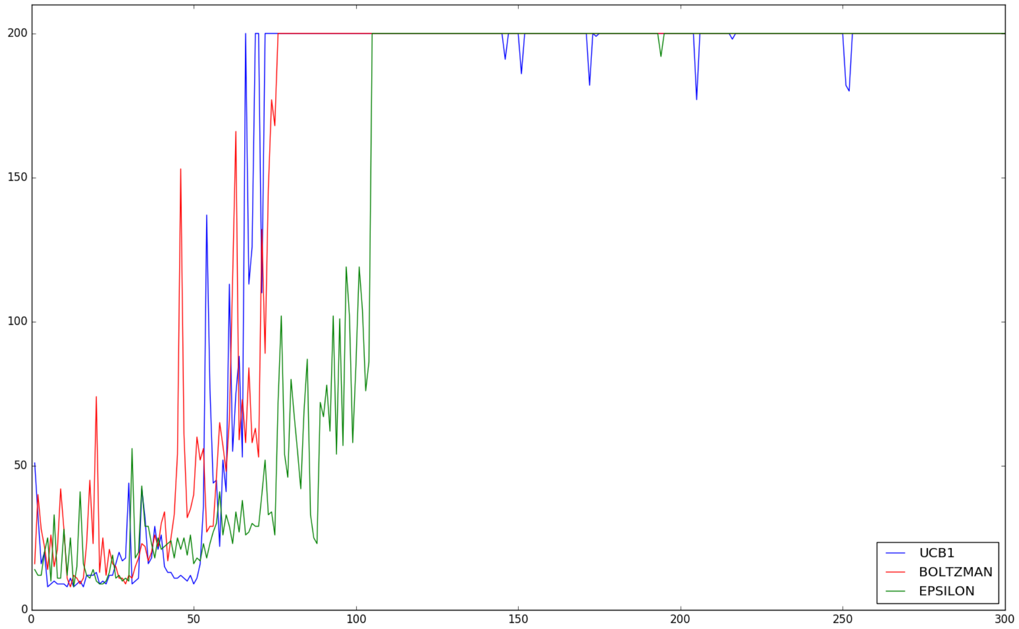

Figure 7.2: Mean reward per episode in the CartPole-v0 environment

In this first environment, we can see that the epsilon greedy policy perfoms poorly and that the two other one are quite the same. However, the UCB1 policy is slighly better. In these experiments, the hyper parameters[1] of the algorithm are the same for the three policies. The only

[1] $\gamma = 0.99$, $batch\_size = 200$ and the learning rate starts at $2e-3$ and decreases to $1e-7$ over 50000 steps

parameters that I tuned for each policy are the parameters inside the policy like the decay of epsilon for instance.

Regarding the hyper-parameters of the Double DQN, I used a future decay, denoted γ, of 0.99. Plus, when using a constant learning rate, I saw that sometimes the agent will achieve poor rewards over an episode even after having learned the optimal policy. In other words, the rewards per episode decreased a lot after a lot of consecutively optimal rewards. To solve this problem, I used a linear decay of the learning rate preventing the agent to bring big changes to his policy when the optimal one is reached.

CartPole-v1

Then, it is interesting to look at the behaviour of the same algorithms on a more complex environment. Before running, I slightly changed the architecture of the neural network by switching to a single hidden layer with 400 neurons. The results of those algorithms on the CartPole-v1 environment are available in Figure 7.3. Also, it is the mean reward of my three best runs over 20 runs.

Again, the UCB1 policy slightly outperforms the Boltzman Q Policy. However, the epsilon greedy policy is not able to solve the environment.

Conclusion

We have seen that the UCB1 policy seems to be the best one among the simple policies tried. As a results, it is the policy that is going to be used in the following experiments.

7.2.2 Change of architecture

In this part, we are going to see the performances of the Bootstrapping DQN, the Experience Prioritized Replay and the Dueling Architecture. The idea is to implement them and compare their performances on our test environments.

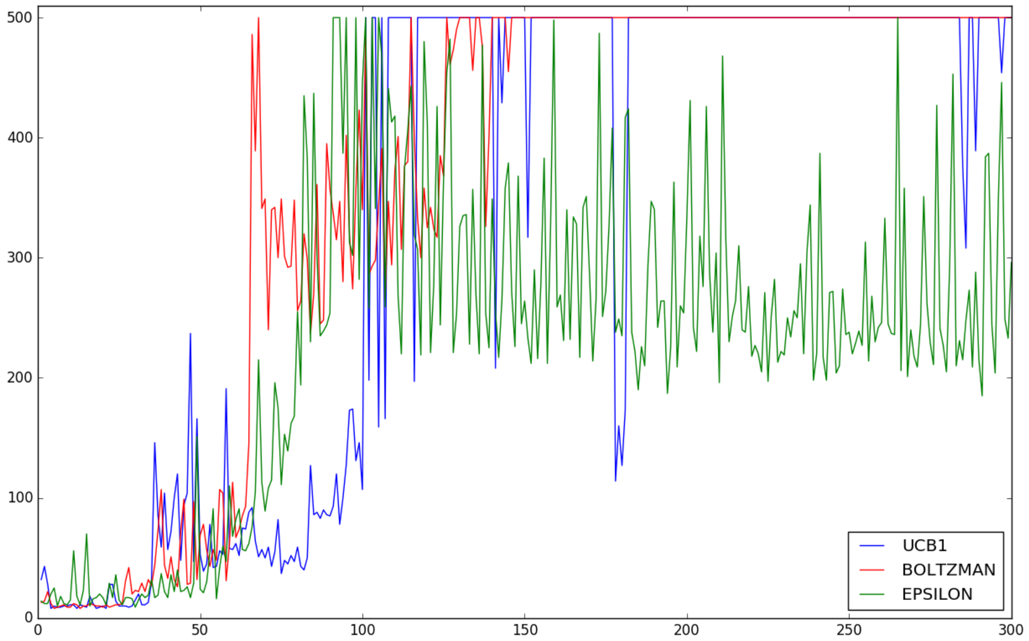

Figure 7.3: Mean reward per episode in the CartPole-v1 environment

Implementation

I implemented the Bootstrapped DQN by sharing the datas between all the heads because it seems to work better on the paper and was a lot easier to implement. The Bootstrapped DQN is combined with the DoubleDQN. Indeed, in the DoubleDQN, there are two Q-Networks. Then, here I have two Q-Networks with K heads each. At each episode, a new index is sampled uniformally that is the head that will be used in both network. When predicting, the mean of the q-values predicted by the selected head of each Q-Network is sent to the UCB1 policy to select the action.

I implemented the Prioritized Experience Replay using a max Binary Heap. Indeed, it allows to access in $O(1)$ time the top priority and to update a priority in $O(\log N)$ time. Moreover, I used the rank based version for the prioritiy of the transitions. Finally, the complexity for sampling from the distribution can not depend on the size of the replay if we want it to be quick. That's why, I approximated the cumulative density function with a piecewise linear function with k

segments of equal probability. To sample a transition, we sample a segment, then we sample uniformally among the transitions within it.

CartPole-v0

Firstly, I run those three algorithms on CartPole-v0 and I compared the performances with the simple DoubleDQN. For the Dueling Architecture, I added a 200 neurons fully connected layer in each stream, just before the state-value and the action-advantage streams. For the Bootstrapped DQN, the shared network is composed of 150 neurons. Each head has a fully connected layer of 100 neurons before the output layer. I noticed that increasing the number of heads tend to improve the performances of the algorithms. Here, I used 20 heads. The results are shown in the Figure 7.4. It is the mean of the 5 best runs over 20 runs.

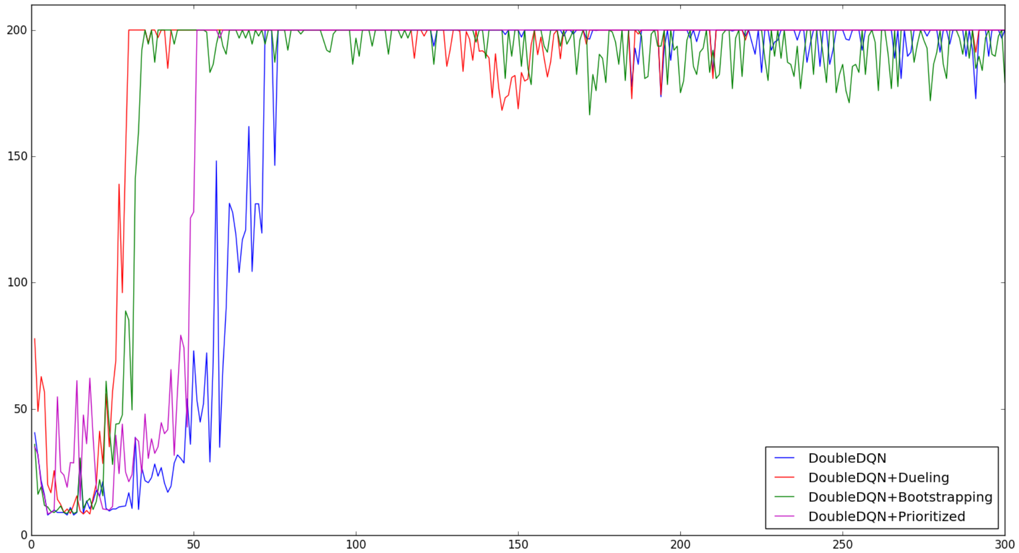

Figure 7.4: Mean reward per episode in the CartPole-v0 environment

We can see that the three new algorithms outperform the standard DoubleDQN. On this environment, the best one is the Dueling Architecture.

CartPole-v1

It is the same configuration of the Bootstrapping and of the Dueling except that the main Network is a single hidden layer of 400 neurons. The results are shown in Figure 7.5. It is the mean of the 5 best runs over 30 runs.

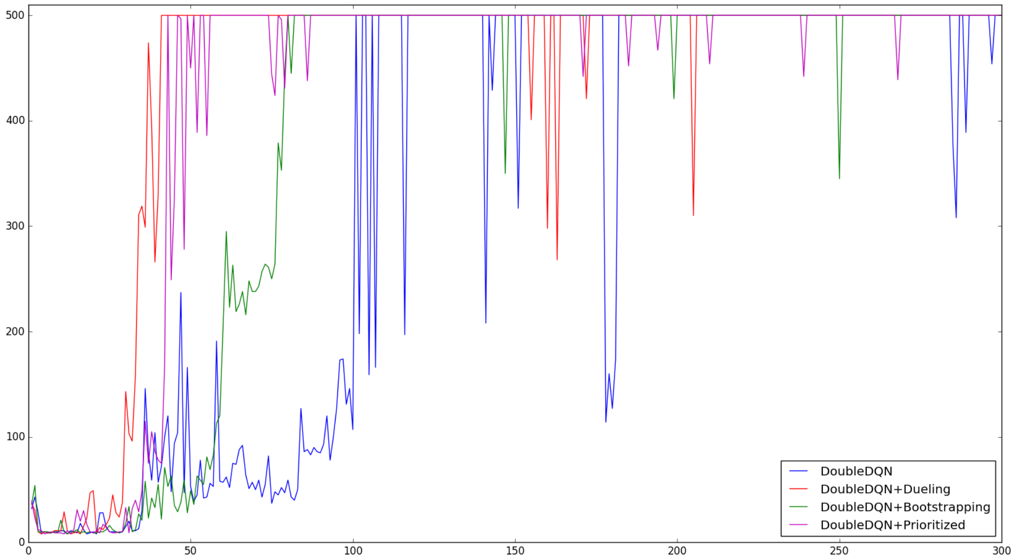

Figure 7.5: Mean reward per episode in the CartPole-v1 environment

Again on this environment, the three algorithms outperform the DoubleDQN. But this time the Prioritized Replay is better than the bootstrapping.

Conclusion

The first conclusion possible is that the Prioritized Experience Replay, the Bootstrapped DQN and the Dueling Architecture all learn quicker than the DoubleDQN. This is good because it was the main goal of this experiment to confirm the efficiency of those algorithms.

The second one is that the Dueling Architecure is the best algorithm for both environments. It is really interesting because the efficiency of Dueling is supposed to increase with the number of possible actions. But, here only two actions are available. It is then very promising for more complex games where a lot of actions are available. Nevertheless, an intuitive explanation of this good result is that when using a single stream network, only the network for one action is updated whereas with a two streams, the state-stream is updated and is used for every actions. This allows to learn the value function more quickly.

Finally, we can see that the Prioritized Replay performs poorly in CartPole-v0 but is a lot better and close to the Dueling in the CartPole-v1. This gain of performance is very interesting and further experiments with this algorithm on more complicated environment is needed.

7.2.3 Mix

Now that we have seen the performances of the different change in the algorithm. Let's try to combine them and improve our results. The same architectures as before are going to be used. This is not an ensemble method. Here, the architecure of each algorithm is combined. The Dueling and the Bootstrapping are combined by adding a Dueling architecture inside each head. Indeed, I explained before that each head was composed of a single hidden layer and of an output layer. Then, this layer is replaced by two streams of each two layers, the first one is an hidden layer and the value-functionn, the second one is an hidden layer and the action-advantage function. The results of the experiments are available on Figure 7.6 for CartPole-v0 and on Figure 7.7 for CartPole-v1. It is the mean of the 5 best runs over 20 runs.

Conclusion

In both environments the combination of Dueling and Bootstrapping brings the best results. It is this combination that brought me my best results on OpenAI Gym. Again, we can see that all the combination are able to do better than the standard DQN but are not able to much better than the DoubleDQN with only the Dueling Architecture.

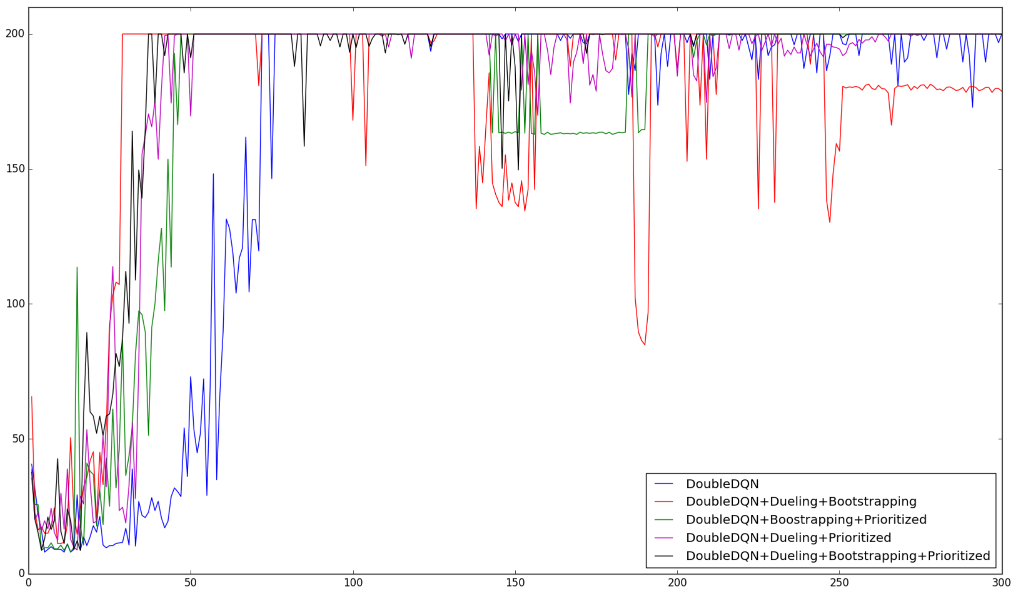

Figure 7.6: Mean reward per episode in the CartPole-v0 environment

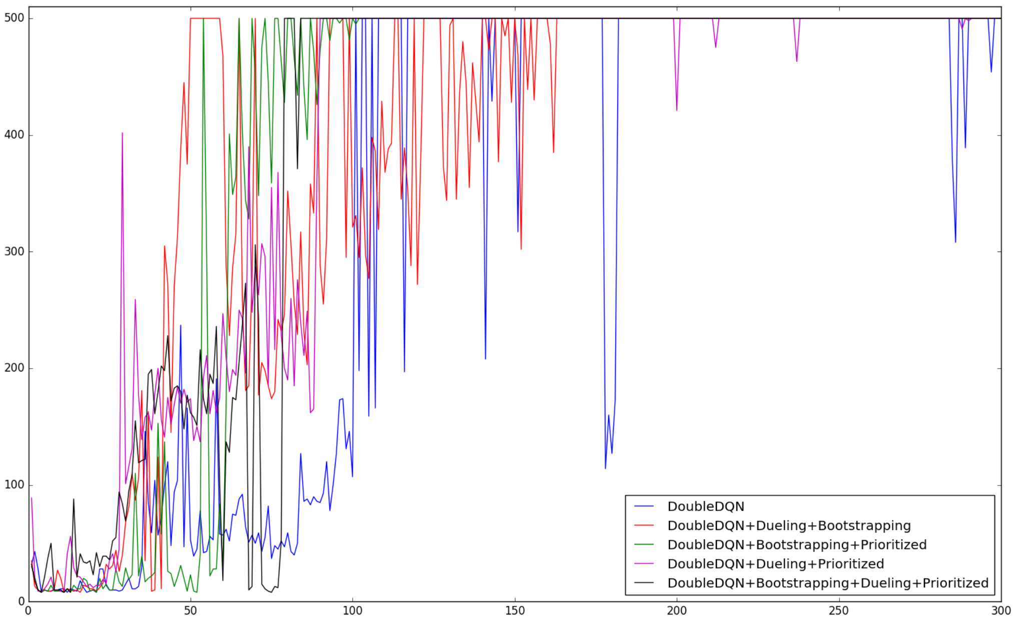

Figure 7.7: Mean reward per episode in the CartPole-v1 environment

Chapter 8

Summary

This report showed my work for the Specialisation Project. I deeply studied the Reinforcement Learning field. This project allowed me to learn a lot about it. I started with the basics of Reinforcement Learning and progressively went to more complex methods. I was able to implement the DQN algorithm from DeepMind which is used by almost every other algorithms. I even upgrated it by going to a DoubleDQN algorithm. From my experiments, the UCB1 methods that uses the phenomenom of Optimism in Face of Uncertainty was the most successfull of the easy one. I then tried several change of architecture and change in the algorithm like Bootstrapping, Prioritized Replay and the Dueling Architecture. It is the latter that brought me the best results when combined with a Double Q Learning and a UCB1 policy. Finally, I tried mixing those algorithms to have a better one and only the combination of Dueling and Bootstrapping improved my results.

Bibliography

Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. *arXiv preprint arXiv:1606.01868*.

Houthooft, R., Chen, X., Duan, Y., Schulman, J., De Turck, F., and Abbeel, P. (2016). Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks. *arXiv preprint arXiv:1605.09674*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. (2016). Deep exploration via bootstrapped dqn. *arXiv preprint arXiv:1602.04621*.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Stadie, B. C., Levine, S., and Abbeel, P. (2015). Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning. *CoRR, abs/1509.06461*.

Wang, Z., de Freitas, N., and Lanctot, M. (2015). Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.