Deep Q-Learning for Hard Exploration Problems

Palacios Caballero, Joan

Curs 2019-2020

Director: Miquel Junyent

GRAU EN ENGINYERIA INFORMÀTICA

Treball de Fi de Grau

UPF : Department of Information and Communications Technologies
University Pompeu Fabra

Deep Q-Learning for Hard Exploration Problems

Joan Palacios Caballero

Degree Final Project, September 2020

Deep Q-Learning for Hard Exploration Problems

Joan Palacios Caballero
Supervised by Miquel Junyent

14th September 2020

Bachelor in Computer Science
Department of Information and Communication Technologies

Universitat Pompeu Fabra
Barcelona

Abstract

State-of-the-art methods for deep reinforcement learning have demonstrated the ability to learn complex game strategies for Atari's games, Chess, Go... Deep Q-Learning was the first deep reinforcement learning method to outperform the human level at some of Atari's games, however, it performs poorly in environments with sparse rewards or with the facility to die. In both cases, the algorithm is not able to learn from low probability rewards.

In this project, a modified version of Deep-Q Learning which implements Deep Q-Learning with Boltzmann Count-Based exploration (DQL-B-C) has been developed. The algorithm has been tested and compared in a grid environment with different grid sizes. The experiments demonstrate that DQL-B-C explores better and learns more quickly.

Acknowledgements

I wish to offer my thanks:

- To my family, for supporting me in all the possible ways.
- To my friends, for making me happy.
- To Miquel Juyent, for supervising me even with the COVID-19 adversities.

Thanks.

Contents

List of Figures

CHAPTER 1

Introduction

Intelligence is probably one of the most important characteristics of humanity, it's a hard term to describe since involves a lot of philosophical questions. According to Oxford Languages, intelligence is the ability to acquire and apply knowledge and skills. A very useful characteristic that allowed our ancestors to survive and adapt until today.

Artificial intelligence (AI) is a research field that tries to understand and develop intelligence for machines. To achieve this goal, different approaches have been made, for example, some researchers, inspired by the brain structure, have developed a mathematical model where a combined artificial neurons connected together generate an artificial neural net. This last approach has been useful and successful in order to solve a bunch of different problems in multiples fields, for example, in computer vision, a huge improvement in image classification has been made, where a neural network can be trained to figure out what objects an image contains, this process is well described at Simonyan and Zisserman (2014) and Szegedy et al. (2015).

Recently, an area of AI called Reinforcement Learning (RL) has become really popular, this area is concerned about teaching an agent to do a task inside an environment. The methodology to teach the agent is usually based on the interactions between the agent and the environment. The agent executes an action inside the environment and by observing the effects of this action it can modify his behaviour.
One of the most significant improvements in RL was made by Mnih et al. (2015) implementing a new algorithm called Deep Q-Learning (DQL) which was able to play Atari's games outperforming, in some of them, the human level.

However, DQL did not perform well at some games such as Montezuma's Revenge or Private Eye. The reason for poorly performing at those games was because the rewards inside the environments were very sparse, and also because the agent died very frequently without learning anything useful.

This thesis aims to improve DQL in sparse rewards environments by applying Boltzmann exploration and Count-Based exploration (M. Bellemare et al. 2016), and it is structured in the following way:

- Background: The description of the mathematical tools needed to understand, implement and improve DQL.
- Methodology: The description of the environment, the algorithm, and the improvements.
- Results: The analysis of the improvements made in the algorithm.
- Conclusions and Future Work: The sum up about the thesis and the future work.

CHAPTER 2

Background

In this chapter, we describe the mathematical tools needed to understand and extend the state of the art of deep reinforcement learning. We are going to cover two important topics which are:

- Reinforcement learning: In this section, we are going to explain the basics about RL. Concepts like Sequential Decision Process, Markov Decision Process, policy, value function...
- Deep learning: In this section, we are going to explain the basics about DL. Concepts like artificial network, multilayer perceptrons, different types of layers...

This knowledge will be applied in the next chapter in order to formally describe the implemented algorithm.

2.1 Reinforcement learning

2.1.1 Sequential Decision Process

A *Sequential decision process* (SDP) describes a situation where the decision maker makes successive observations of a process before a final decision is made. In most sequential decision problems there is an implicit or explicit cost associated with each observation. The procedure to decide when to stop taking observations and when to continue is called the 'stopping rule'. The objective in sequential decision making is to find a stopping rule that optimizes the decision in terms of minimizing losses or maximizing gains, including observation costs (Smelser, Baltes et al. 2001).

The first concept that we need in order to define properly our problem is a model that allows us to represent it, and this is exactly what the Agent-Environment interface does.

The agent and environment interact with each other in a sequence of discrete time steps, $t = 0, 1, 2, ...$ At each time step t, the agent receives some representation of the environment's state, $s_t \in S$, where S is the set of possible states, and on that basis selects an action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions available in state s_t. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in \Re$, and finds itself in a new state, s_{t+1} (Sutton and Barto 2018). Notice how the reward could be negative, implying that the agent will try to avoid those states. The graphic representation of the Agent-Environment Interface is shown at Figure 2.1.

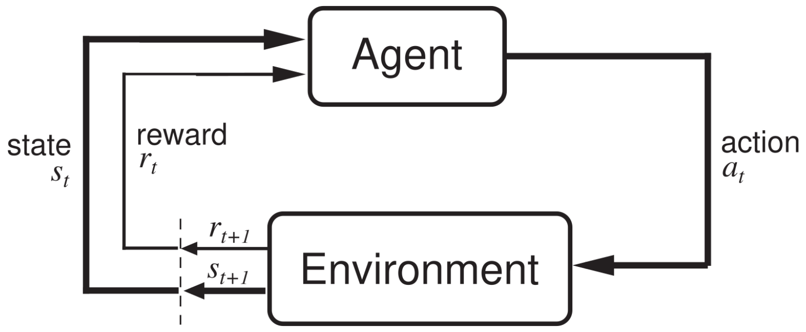


Figure 2.1: Diagram of interaction between agent and environment (Sutton and Barto, 1998, Section 3.1)

This model is very powerful since it can accept very flexible sets of states and actions, we can define our states as we want, for example in a computer game the state can be a screenshot about the actual situation of the game or some specific values of the RAM which have some direct correlation with the game. Moreover, the time step control need not refer to fixed intervals of real time, as in chess for example, which every time step could be defined by a move done. In addition, the actions can be defined as complex as we want, such as the voltages applied to the motors of a robot arm or a high level decisions like turn right or turn left such as Messias et al. (2013).

2.1.2 Characteristics of SDPs

Fully observable vs partially observable

The first important characteristic about SDPs is that they can be *fully observable* or *partially observable.* In a fully observable SDP the information received by the agent at any point of time is sufficient to make the optimal decision, in the other hand, an environment is partially observable when the agent needs memory in order to make the best possible decision.

Episodic vs continuing SDPs

There exist two types of SDPs according to their duration, the *episodic* SDPs are those where the agent has to accomplish a task and achieve a final state. In contrast *continuing* SDPs never stop, in theory.

Deterministic vs stochastic SDPs

Depending on how the next state of an SDP is determined it can be deterministic or stochastic. The *stochastic* SDP are those that the next state is given from a probability distribution over all possible S, when this distribution has all its weights on a single state the SDP is no longer stochastic, it's *deterministic*.

2.1.3 Rewards and returns

A *reward* is a numerical value that the agent receives when performs an action at some state. The *return* is the total sum of rewards. The goal of the agent is to maximize return. In order to achieve that, we have to define the expected return R_t in a way that encourages the agent to learn, we define R_t as the sum of all rewards until the last time step T:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T = \sum_{i=t+1}^{T} r_i \tag{2.1}$$

As we defined previously, the last time step is T, but in a continuing task, $T = \infty$ implying that the return will also be infinite. To manage this problem, a discount factor γ is usually introduced to the previous equation, generating a new one called the *discounted return*.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

Notice that γ is a parameter, $0 \leq \gamma \leq 1$ such that defines how important are the future rewards taking in account how far they are, basically if $\gamma = 1$ the Equation 2.1 and Equation 2.2 are the same, so it means that every reward it's equally important and if $T = \infty$ the return is going to diverge.
However if $\gamma < 1$, we solve our infinite return problem: as the time step approaches infinity, the weight its reward is scaled by approaches zero, and R_t converges. In addition we also benefit from $\gamma < 1$ in the episodic setting, since we'll endorse shorter trajectories. (In the case $\gamma = 1$, there is no difference for the agent between navigating forever before getting the reward and getting it immediately).

2.1.4 Markov Decision Process

A *Markov Decision Process* (MDP) is a restricted case of SDPs. In SDPs the state s_{t+1} may depend on all the past states, actions and reward. However, in an MDP the s_{t+1} only depends on action a_t and state s_t. This fact is named as the Markov property and is formally described as:

$$\begin{aligned} P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, r_1, s_0, a_0\} \\ = P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t\} \end{aligned} \tag{2.3}$$

A MDP can be represented with the following tuple $(S, A_s, P^a_{ss'}, R^a_{ss'}, \gamma)$:

- The set of possible states, S.
- The available actions for each state, as a function of the state, A_s.
- The matrix of transition probabilities from each state to another, given an action, $P^a_{ss'} = P(s_{t+1} = s', a_t = a)$.
- The expected reward given a state, an action and the next state, $R^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$.
- The discount factor for calculating returns, γ.

Note that the model does not explicitly represent the probability distribution on rewards, only the expectation.
S and A_s may be infinite sets which implies that $P^a_{ss'}$ and $R^a_{ss'}$ may also be infinite. If this happens the MDP is named infinite MDP.

2.1.5 Policy and value functions

A *policy* π is a probability distribution for each state $s \in S$ over the possible actions to take $a \in A_s$. In other words, the policy π defines the agent's behaviour at each state.
The *value* $V^\pi(s)$ is the expected return for an agent that is currently in state s and follows the policy π. In other words, the $V^\pi(s)$ is an estimate of how good it is for the agent to be in a given state according to his policy.

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} = E_\pi\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \tag{2.4}$$

The *action-value function* $Q^\pi(s,a)$ is the expected return for an agent that is currently in state s and takes the action a and then follows the policy π.

$$Q^\pi(s,a) = E_\pi\{R_t | s_t = s, a_t = a\} = E_\pi\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \tag{2.5}$$

2.1.6 Optimal policy

There is a very special policy that we are concerned about, the *optimal policy*. The optimal policy for a state s is that one with the maximum value for that state, and is denoted by π^*_s.

$$\pi^*_s = \arg\max_\pi V^\pi(s) \tag{2.6}$$

Furthermore, according to the previous equation, the $V^\pi(s)$ must be the true value so it can be denoted by $V^{\pi^*}(s)$. So if we know the true value function of all states and the transition model, we can calculate the optimal policy:

$$\pi^*(s) = \arg\max_{a \in A_s} \sum_{s' \in S} P^a_{ss'} V(s) \tag{2.7}$$

This is the main goal of value-based reinforcement learning algorithms which will try to learn the $V^{\pi^*}(s)$ in order to converge to the optimal policy.

2.1.7 Q-Learning

Q-learning is an algorithm that approximates the optimal policy by learning the action value function due the experiences that it gets by interacting with the environment. It updates the Q function in each step t using:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.8}$$

Where α is a constant step-size parameter, named learning rate, that indicate how much has to take in account the followings Q values in order to update the actual.
The previous formula shows that the Q value is being updated based on rewards and the max Q of all actions in s_{t+1}, which implies, in the long term, Q converge to Q^*. Notice, how the update process did not depend on the policy followed, this is very important, since it will converge even if we are using a random action selection. The Q-learning algorithm is shown in procedural form in Algorithm 1.

Algorithm 1 Q-Learning Algorithm

```
 1: procedure BUILDING Q-MATRIX
 2:     Set $\alpha$ and $\gamma$ parameters.
 3:     Initialize Q-Matrix to zero.
 4:     repeat
 5:         while Goal/terminal state not reached do
 6:             Select $a$ randomly from all possible actions in current state
 7:             Consider going to state $s_{t+1}$ from state $s$ using action $a$
 8:             Get maximum Q-Value from $s_{t+1}$ considering all possible actions
 9:             $Q(s, a) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1}, a))$
10:         end while
11:     until Policy good enough
12: end procedure
13: procedure USING Q-MATRIX
14:     $s \leftarrow$ initial state
15:     while Goal/terminal state not reached do
16:         $a \leftarrow max_a Q(s, a)$
17:         $s \leftarrow s_{t+1}$ taking action $a$
18:     end while
19: end procedure
```

2.1.8 Exploration Vs. Exploitation

While the agent is interacting with the environment, they face up the exploration-exploitation dilemma, which is the trade-off between the need to get new knowledge to obtain bigger rewards, and the need to use the acquired knowledge to get the known rewards.
A strategy that helps to solve this dilemma is the ϵ-greedy strategy, which suggests that our agent should use an ϵ-greedy policy defined as:

- Take $\pi^*(s)$ with probability $1 - \epsilon$.

- Take an uniformly randomly sampled $a \in A_s$ with probability ϵ.

Where ϵ is a parameter, $0 \leq \epsilon \leq 1$ (Sutton and Barto 2018).

2.2 Deep learning

2.2.1 Artificial neuron

An *artificial neuron* can be defined by Figure 2.2, which has the following sections (Haykin et al. 2009):

- A set of connecting links (synapses), each one associated with some weight (synaptic weights). A signal x_j at the input of connection j related to neuron k is multiplied by the weight w_{kj}.
- An adder (summing junction) for summing the input signals, weighted by the respective weight strengths of the neuron.
- An activation function $\varphi(\cdot)$ that limits the amplitude of the output of a neuron.
- An externally applied bias b_k which has the effect of increases or reduces the input of the activation function.
- An output signal y_k which is the result of the processed information.

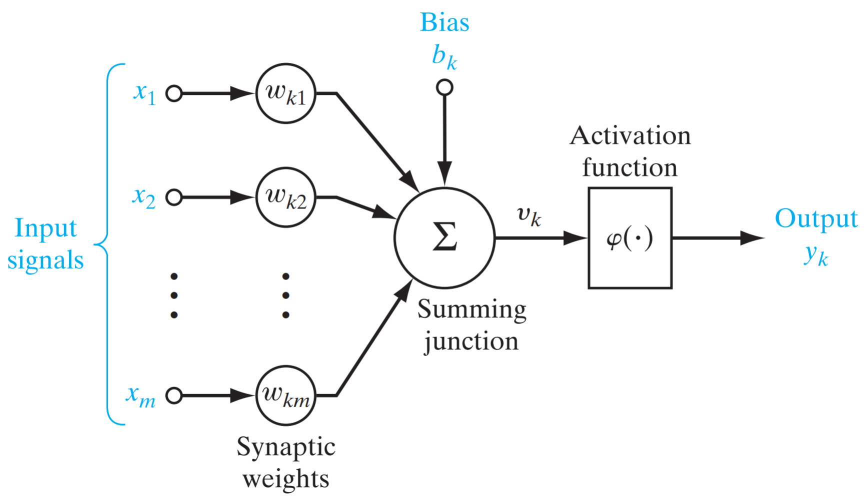


Figure 2.2: Nonlinear model of a neuron, labeled k. (Neural Networks and Learning Machines, 2009, Section 3)

Mathematically it can be described as:

$$y_k = \varphi\left(b_k + \sum_{j=1}^{m} w_{kj} x_j\right) \tag{2.9}$$

Where $x_1, x_2, \cdots, x_m$ are the input signals; $w_{k1}, w_{k2}, \cdots, w_{km}$ are the respective weights of neuron k; b_k is the bias; $\varphi(\cdot)$ is the activation function; and y_k is the output signal of the neuron.

2.2.2 Multilayer Perceptrons

A *multilayer perceptron* is made by several layers of neurons. It has three types of layers, the input layer, which is where the input signal comes, the hidden layers, which are those between the input layer and the output layer, and the output layer which emit the output signal. When each neuron has as input all neurons of previous layer the network is named fully connected. The previous concepts are illustrated by Figure 2.3.

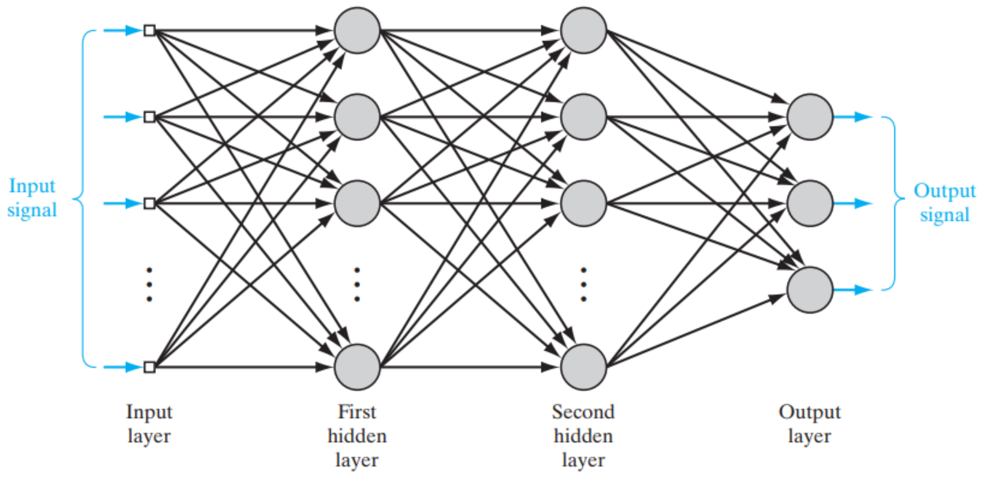


Figure 2.3: Architectural graph of a multilayer perceptron with two hidden layers (Neural Networks and Learning Machines, 2009, Section 4).

In a multilayer perceptrons exists two types of signal (Haykin et al. 2009):

1. **Function Signals**. A function signal is an input signal (stimulus) that comes in at the input end of the network, propagates forward (neuron by neuron) through the network, and emerges at the output end of the network as an output signal. We refer to such a signal as a “function signal” for two reasons. First, it is presumed to perform a useful function at the output of the network. Second, at each neuron of the network through which a function signal passes, the signal is calculated as a function of the inputs and associated weights applied to that neuron. The function signal is also referred to as the input signal.

2. **Error Signals**. An error signal originates at an output neuron of the network and propagates backward (layer by layer) through the network. We refer to it as an “error signal” because its computation by every

neuron of the network involves an error-dependent function in one form or another.

The process of adjusting the weights and biases of each neuron to approximate a required output signal, given some input signal, is named *training.* In order to train the net and balance her weights, a gradient descent algorithm is used.

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. The size of these steps is called the learning rate. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A Loss Functions tells us "how good" our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to update our parameters to make the model more accurate. (Ruder 2016)

2.2.3 Convolutional Neural Network

Convolutional Neural Network is a variation of the multilayer perceptron, which it has been designed to recognize visual patterns. In order to achieve this goal, it is necessary to implement different types of layers: *convolutional layers*, *pooling layers* and *fully connected layers.*

Convolutional layers apply a *convolution* to the input signal, this convolutions are characterized by a *filter* and this filter is characterized by a *kernel size*, *stride*, *padding* and *values.*

- The *kernel size* refers to the shape of the filter mask.
- The *stride* defines how the kernel moves across the input.
- The *padding* is the amount of extra information, normally zeros, to add around the border of each dimension in the input.
- The *values* defines the value of each position in the filter mask.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as:

$$\mathrm{out}(\mathrm{N_i}, \mathrm{C_{out_j}}) = \mathrm{bias}(\mathrm{C_{out_j}}) + \sum_{\mathrm{k}=0}^{\mathrm{C_{in}}-1} \mathrm{weight}(\mathrm{C_{out_j}}, \mathrm{k}) \star \mathrm{input}(\mathrm{N_i}, \mathrm{k}) \tag{2.10}$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels. (Paszke et al. 2019)

Pooling layers generate a down-sampled version of the input by applying some function (usually a max function) to different regions from the input. Its function is reduce the spatial size of the input in order to decrease the amount of computations and parameters in the network without losing the essential information. It also reduces overfitting (Paszke et al. 2019).
In the simplest case, the output value of the layer with input size (N, C, H, W) and output (N, C, H_{out}, W_{out}) and kernel size (kH, kW) can be precisely described as:

$$\mathrm{out}(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \mathrm{input}(N_i, C_j, \mathrm{stride}[0] \times h + m, \mathrm{stride}[1] \times w + n) \tag{2.11}$$

2.2.4 Activation function

Finally, another important concept to understand, are the activation functions, which are functions applied at the end of a layer in order to decide which are going to be the activation value (the output).

A very popular activation functions is the *Rectified Linear Unit* ($ReLU$) which increases the nonlinear properties of the model. They are usually used after convolutional layers because they mitigate the *vanishing gradient problem*, which is characterized by an exponential reduction of the gradient through the layers. $ReLU$ can be mathematically described as (Equation 2.12 and Figure 2.4):

$$f(x) = max(0, x) \tag{2.12}$$

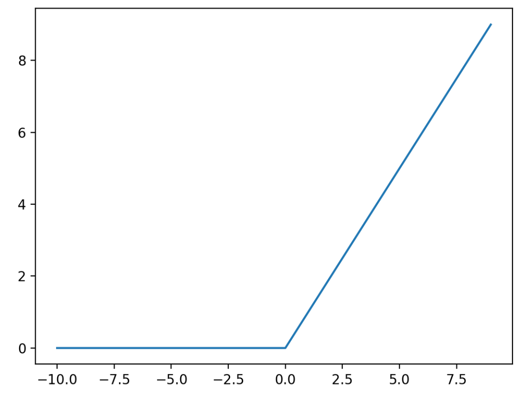


Figure 2.4: The Rectified Linear Unit function

CHAPTER 3

Methodology

In this chapter we will describe the algorithms used and the environment where we tested it. We will also describe the improvements that we made to the environment and the implementation details about the new proposed algorithm.

3.1 Algorithm

3.1.1 Deep Q-Learning

Deep Q-Learning (DQL), a modification of online Q-learning (Subsection 2.1.7), is introduced in two papers, Mnih et al. (2013) and Mnih et al. (2015).
DQL uses a Deep Q Network (DQN) which is a Deep Convolutional Neural Network that outputs the Q values. So, the input of the network will be an image, which represents the state in our RL problem, and the output will be the Q values of each action. So, the input of the network will be an image, which represents the state in our RL problem, and the output will be the Q values of each action. Once DQL learns the Q values it is possible to extract the policy from there (Section 2.1.6).

In order to train the network and overcome unstable learning, DQL implements two important features:

- **Experience Replay**: In supervised learning, we want the input to be **independent and identically distributed** (i.i.d.), i.e.samples are randomized among batches and therefore each batch has the same (or similar) data distribution. Samples are independent of each other in the same batch.

  In order to accomplish the previous statement DQL uses a technique known as *experience replay*, in which it stores the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$, in a data set $D = \{e1, \cdots, e_t\}$, pooled over many episodes (where the end of an episode occurs when a terminal state is reached) into a replay memory. During the inner loop of the algorithm, DQL applies Q-learning updates, or minibatch updates, to samples of experience, $(s_t, a_t, r_t, s') \sim U(D)$, drawn at random from the pool of stored samples (Mnih et al. 2015).

- **Target Network**: The second modification to online Q-learning aimed at further improving the stability of the method with neural networks is to use a separate network for generating the targets y_j in the Q-learning update. More precisely, every C updates DQL clones the network Q to obtain a target network $\hat{Q}$ and use $\hat{Q}$ for generating the Q-learning targets y_j for the following C updates to Q. This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$ for all a and hence also increases the target y_j, possibly leading to oscillations or divergence of the policy (Mnih et al. 2015).

The loss function is defined in the following way:

$$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right] \tag{3.1}$$

Where θ_i and θ_i^- are the weights of two DQN with the same architecture. θ_i is updated each training step while θ_i^- only changes when the iteration finishes, following $\theta_i = \theta_i^-$ after this happens.
The Deep Q-Learning algorithm is fully described by:

Algorithm 2 Deep Q-Network Algorithm with Experience Replay and a Target Network

```
1:  Initialize replay memory $D$ to capacity $N$
2:  Initialize $Q$ with random weights $\theta$
3:  Initialize $\hat{Q}$ with weights $\theta^- = \theta$
4:  **for** episode = 1, M **do**
5:      Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6:      **for** t = 1, T **do**
7:          With probability $\epsilon$ select a random action $a_t$
8:          otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
9:          Execute action $a_t$ and observe reward $r_t$ and image $x_{t+1}$
10:         Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
11:         Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
12:         Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
13:         **if** episode terminates at step $j + 1$ **then**
14:             Set $y_j = r_j$
15:         **else**
16:             Set $y_j = r_j + \gamma max'_a \hat{Q}(\phi_{j+1}, a'; \theta^-)$
17:         **end if**
18:         Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect
    to the network parameters $\theta$
19:         Every $C$ steps reset $\hat{Q} = Q$
20:     **end for**
21: **end for**
```

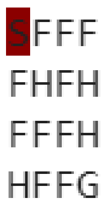
3.2 Environment

3.2.1 Frozen Lake

The chosen environment to implement and improve DQL is *Frozen Lake*, which is an environment provided by **gym** (Brockman et al. 2016), a toolkit for developing and comparing reinforcement learning algorithms designed by OpenAI.
The agent controls the movement of a character in a grid world. Some cells of the grid are walkable, and others lead to the agent falling into the water. The agent starts at the top left corner and his goal is to reach the right bottom corner without falling into a hole.

At Figure 3.1 we can see an example of the grid world, where the starting point is at left top, the agent is at the starting point and the goal is at right bottom.

- S: starting point.
- F: frozen surface, safe.
- H: hole, unsafe.
- G: goal, where the agent has to go.



```
SFFF
FHFH
FFFH
HFFG
```

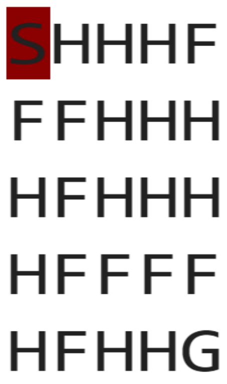
Figure 3.1: A 4x4 grid

3.2.2 Improving Frozen Lake

In order to improve Frozen Lake and adapt it to DQL we have programmed some additional functions that we will use later in order to test, visualize and improve DQL.

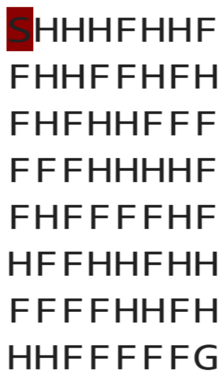
Random grid generator

The first additional function is a random map generation. This tool will allow us to test our algorithm in different grids, it can create random grids given the size of the grid. Some examples:



```
SHHHF
FFHHH
HFHHH
HFFFF
HFHHG
```

Figure 3.2: A 5x5 grid generated randomly



```
SHHHFHHF
FHHFFHFH
FHFHHFFF
FFFHHHHF
FHFFFFHF
HFFHHFHH
FFFFHHFH
HHFFFFFG
```

Figure 3.3: A 8x8 grid generated randomly

Changing the state, number to matrix

The main problem about the environment provided by OpenAI is the representation of the **state**, (Subsection 2.1.1) it's only a number, although the environment allows you to render it as a matrix such as in Figures 3.2 and Figure 3.3, that's only for visualization, and Frozen Lake does not have access to that. It has access to the actual state, which is the agent's position inside the grid, for example, if the agent is at the starting point, then, the state is 0. In order to start implementing our first version of DQL we have built a function that transforms the state, a number, into a matrix, where the numbers inside the matrix are equivalent to:

- 0: goal.
- 1: hole.
- 2: agent.
- 3: frozen surface.

The Figure 3.4 denotes the state representation.

$$M_{4\times 4} = \begin{pmatrix} 2 & 3 & 3 & 3 \\ 3 & 1 & 3 & 1 \\ 3 & 3 & 3 & 1 \\ 1 & 3 & 3 & 0 \end{pmatrix}$$

Figure 3.4: Representation for state 0

Changing the state, matrix to image

At this point we have already a matrix representation for the state, which will allow us to implement a simpler version of DQL for learning purposes. But this is not enough if we want to implement the real version of DQL described at Mnih et al. (2015). We need an image in order to represent the state, so, we have implemented a function that receives a matrix representation for the state and transform it into an image, where the colors inside the image are equivalent to:

- green: goal.
- black: hole.
- red: agent.
- blue: frozen surface.

The following images denotes how the state representation has changed.

$$M_{4\times 4} = \begin{pmatrix} 2 & 3 & 3 & 3 \\ 3 & 1 & 3 & 1 \\ 3 & 3 & 3 & 1 \\ 1 & 3 & 3 & 0 \end{pmatrix}$$

Figure 3.5: Previous representation of the state, a 4x4 matrix

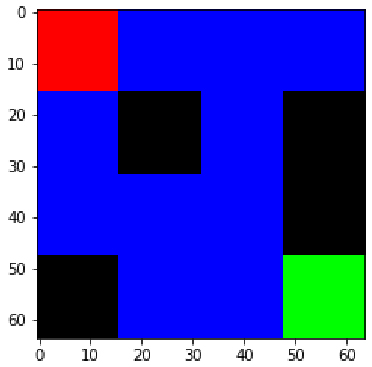


Figure 3.6: Actual representation of the state, a 64x64 RGB image

In addition, and for computational cost reasons, we transform the RGB image into a grayscale image (Figure 3.7).

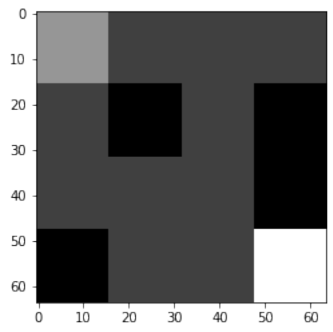


Figure 3.7: Final representation of the state, a 64x64 grayscale image

Plotting Q values

Finally, we have developed a grid which can plot the Q values for each possible action at each cell (Figure 3.8). This can be very useful in order to understand if the algorithm is working properly, moreover it can be helpful for debugging purposes. The starting cell at left top of the grid is identified by a 0 in the middle, the goal cell at right bottom of the grid is identified by a 10 in the middle. The cells are delimited by red lines, white cells represents holes and the other ones are frozen surface. Each cell is a 3x3 subgrid where the Q values are shown for each possible action (up, down, right, left). The Q values are represented by colors from dark purple to green to light yellow. So it appears that the maximum reward lies in the yellow and the least in the purple. So effectively the yellows are leading the way to the solution.

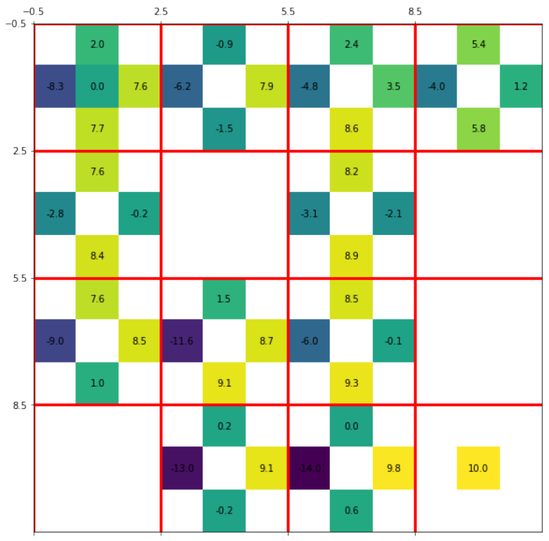


Figure 3.8: A 4x4 grid showing the Q values for each cell

3.3 Implementation

In this section we will describe the process of the implementation and improvement of DQL, all the code was made with Pytorch (Paszke et al. 2019), a deep learning research platform. For learning purposes, the first version of the algorithm was designed to manage states as matrices, the second version of the algorithm uses images instead.

3.3.1 Deep Q-Learning with a simpler state

To begin solving frozen lake, first, we have to formalize it as a MDP (Subsection 2.1.4). That means, defining the following tuple: $(S, A_s, P^a_{ss'}, R^a_{ss'}, \gamma)$:

- Each state is a $n \times n$ matrix $s_t = M$ (Subsection 3.2.2) where n is the grid's side. S is the set of all possible s, $|S| = n^2$.
- $\forall s \; A_s = \{\text{LEFT, DOWN, RIGHT, UP}\}$ and each action $a_t \in A_s$ corresponds to the movement of the agent at time t.
- This game is deterministic, which means that $\forall s \forall a \exists s' \mid P^a_{ss'} = 1$ implying $\forall s \forall a \forall s'' \neq s' \; P^a_{ss''} = 0$.
- The agent only receives a reward if it reach a final state, if it's a hole it receives -1, if it's the goal it receives +10.
- The discount factor γ is 0.95.

The neural network used for this implementation is described at Figure 3.9, the algorithm was tested with a 4×4 grid, which is represented by a 4×4 matrix, then, the input of the net is a 1 dimensional array of 16 numbers, the flatten matrix.

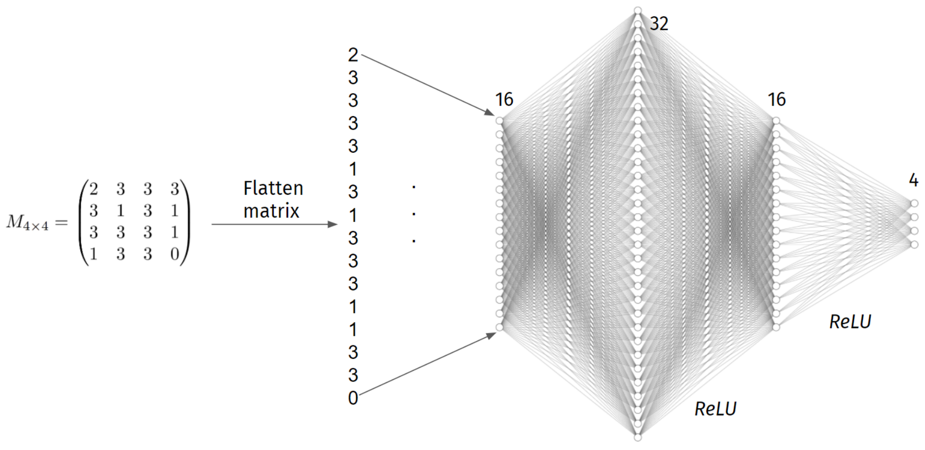

$M_{4\times4} = \begin{pmatrix} 2 & 3 & 3 & 3 \\ 3 & 1 & 3 & 1 \\ 3 & 3 & 3 & 1 \\ 1 & 3 & 3 & 0 \end{pmatrix}$
Flatten matrix
2 3 3 3 3 1 3 1 3 3 3 1 1 3 3 0
16
32
16
4
ReLU
ReLU

Figure 3.9: Neural network architecture

The net is composed by the input layer which receives the flatten 4×4 matrix as a 1 dimensional array of 16 numbers, then, it pass through a fully connected layer with 16 inputs and 32 outputs, then, a ReLU is applied and it pass through another fully connected layer with 32 inputs and 16 outputs and another ReLU is applied. Finally, it pass thought the last layer which receives 16 inputs and emits a signal of 4 outputs (the 4 possible actions).

3.3.2 Deep Q-Learning with ϵ-greedy

After the simpler version of DQL is time to add additional layers to the network and also, change the architecture (Subsection 2.2.3) to implement the real version of the algorithm. The MDP is defined in the same way as before but now, the state representation has changed.

- Each state is a 64×64 grayscale image (Subsection 3.2.2).

As we mentioned before, the net architecture has changed, first, the input layer receives a 64x64 image with 1 input channel (grayscale), then, the image pass through the first convolutional layer with a filter of size 5x5, and emits 32 output channels. After the first convolutional layer, a ReLU and a batch normalization are applied in order to improve the net's stability. Then, the output signal pass thought a MaxPool layer with a kernel size 2x2. Then, the output signal pass again thought another convolutional layer with 32 input channels, 16 output channels and a 5x5 filter. After this, a ReLU and a batch normalization are applied again and finally, the output signal pass through another MaxPool layer with a 2x2 kernel.
Once the signal pass all the previous layers, the output is flattened in order to fit in the followings fully connected layers. There are three fully connected layers: the first one has 2704 input features, 30 output channels, the second one has 30 input channels and 10 output channels, and the third one has 10 input channels and 4 (number of possible actions) output channels.
The scheme about the network architecture is in the following page (Figure 3.10).

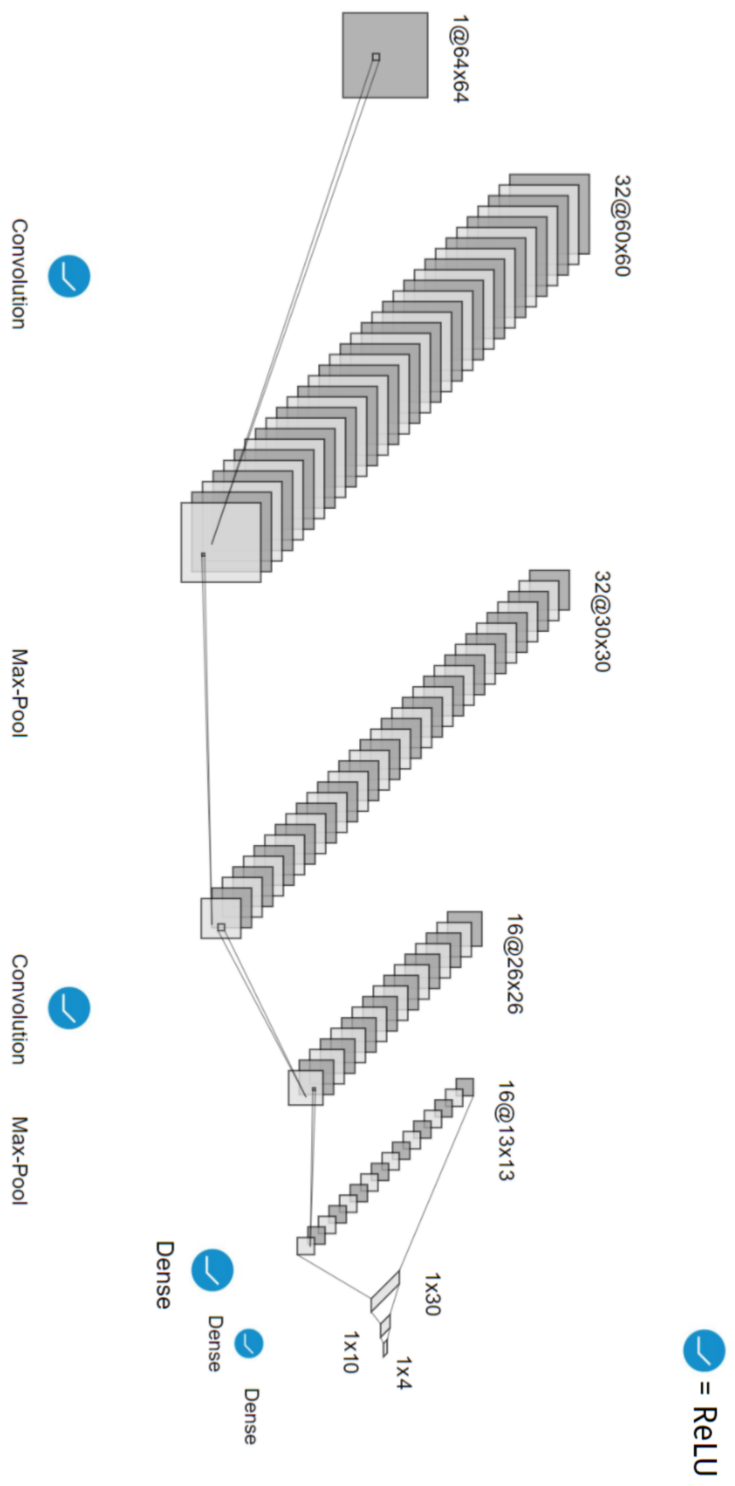


Figure 3.10: Neural network architecture with new layers

3.3.3 Deep Q-Learning with Boltzmann Count-Based exploration

The final implementation of our algorithm is named **Deep Q-Learning with Boltzmann Count-Based exploration** (Algorithm 3), it's very similar to DQL but we mainly change two concepts:

1. We use a **Boltzmann exploration** technique instead of an ϵ-greedy strategy.
2. We introduce an intrinsic exploration bonus r^+ using a **Count-Based exploration**.

Boltzmann exploration

The agent draws actions from a Boltzmann distribution (softmax) (Equation 3.2) over the learned Q values, regulated by a temperature parameter τ.

$$P_t(a) = \frac{exp(q_t(a)/\tau)}{\sum_{i=1}^{n} exp(q_t(i)/\tau)} \tag{3.2}$$

We introduce this technique in order to allow a better exploration, since the agent will choose more frequently the best action according on what it learns but it will keep exploring other actions with a minor probability. The results later indicated that this option was the best compared to ϵ-greedy strategy.

Count-Based exploration

Since we want to encourage exploration in new states, we will give to the agent a bonus reward when it visits states that it has never seen before. In order to do this, we define the bonus reward function as follows (Equation 3.3) (Tang et al. 2017):

$$r^+(\phi(s)) = \frac{1}{\sqrt{counter(\phi(s)) + \beta}} \tag{3.3}$$

Where $counter(\phi(s))$ is the number of times that the agent has seen the preprocessed state s and β is a small constant to avoid $\sqrt{0}$.
The implementation of the counter is made by a dictionary where every time it receives a not seen preprocessed state, it initialize the counter of that preprocessed state to 1, if the preprocessed state has already seen, it increments +1 to the pertinent entry of the dictionary.

Algorithm 3 Deep Q-Network Algorithm with Experience Replay, a Target Network and Boltzmann Count-Based exploration

```
 1: Initialize replay memory $D$ to capacity $N$
 2: Initialize *counter*
 3: Initialize $Q$ with random weights $\theta$
 4: Initialize $\hat{Q}$ with weights $\theta^- = \theta$
 5: for episode = 1, M do
 6:     Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
 7:     Store $\phi_1$ in *counter*
 8:     for t = 1, T do
 9:         Select $a_t$ acording to distribution $softmax(Q(\phi(s_t), a; \theta))$
10:         Execute action $a_t$ and observe reward $r_t$ and image $x_{t+1}$
11:         Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
12:         Set $r_T = r_t + r^+(\phi_{t+1})$
13:         Store $\phi_{t+1}$ to *counter*
14:         Store transition $(\phi_t, a_t, r_T, \phi_{t+1})$ in $D$
15:         Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
16:         if episode terminates at step $j + 1$ then
17:             Set $y_j = r_j$
18:         else
19:             Set $y_j = r_j + \gamma max'_a \hat{Q}(\phi_{j+1}, a'; \theta^-)$
20:         end if
21:         Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect
    to the network parameters $\theta$
22:         Every $C$ steps reset $\hat{Q} = Q$
23:     end for
24: end for
```

CHAPTER 4

Results

In this chapter we show enough evidence about how Deep Q-Learning with Boltzmann Count-Based exploration (DQL-B-C) algorithm learns faster and explores better than DQL in sparse-rewards environments.
In order to provide the evidences, we are going to compare the following algorithms:

- Deep Q-Learning with ϵ-greedy exploration strategy (DQL).
- Deep Q-Learning with Boltzmann exploration strategy (DQL-B).
- Deep Q-Learning with ϵ-greedy exploration strategy and Count-Based exploration (DQL-C).
- Deep Q-Learning with Boltzmann Count-Based exploration strategy (DQL-B-C).

For **ϵ-greedy** exploration strategy we use the following hyperparameters:

- The starter epsilon $\epsilon_o = 1$
- The last epsilon $\epsilon_n = 0.01$
- The decay $\epsilon_{decay} = 0.01$
- $explorationRate(step) = \epsilon_n + (\epsilon_o - \epsilon_n)exp(-\epsilon_{decay}step)$

For **Boltzmann** exploration strategy we use the following hyperparameter:

- The temperature $\tau = 1$

For **Count-Based** exploration strategy we use the following hyperparameter:

- The exploration bonus hyperparameter $\beta = 1$

For **Deep Q-Learning** we use the following hyperparameters:

- The memory capacity $N = 200$.
- The number of episodes $M = 2000$.
- The discount factor $\gamma = 0.95$.
- The episodes before updating target network $C = 15$.

- The learning rate $lr = 11^{-4}$.

We have chosen the hyperparameters with an informal technique which is about pinning the other hyperparameters and try to get the best performance one by one.

4.1 Test environment

In order to test and compare the previous algorithms we have generated three testing maps, which are increasingly more difficult since each one is bigger than the previous.

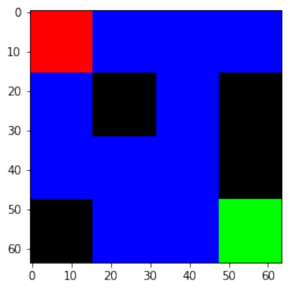


Figure 4.1: The 4×4 map

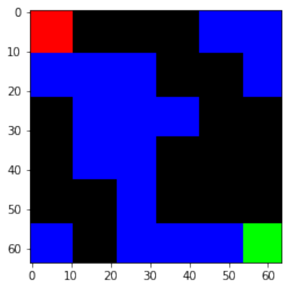


Figure 4.2: The 6×6 map

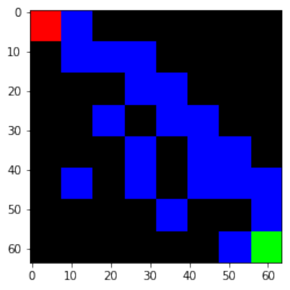


Figure 4.3: The 8×8 map

We are going to test the DQL, DQL-B, DQL-C and DQL-B-C in this three environments, 4 runs for each algorithm, then, we are going to analyze the obtained results and understand the graphs.

4.2 Results 4×4 map

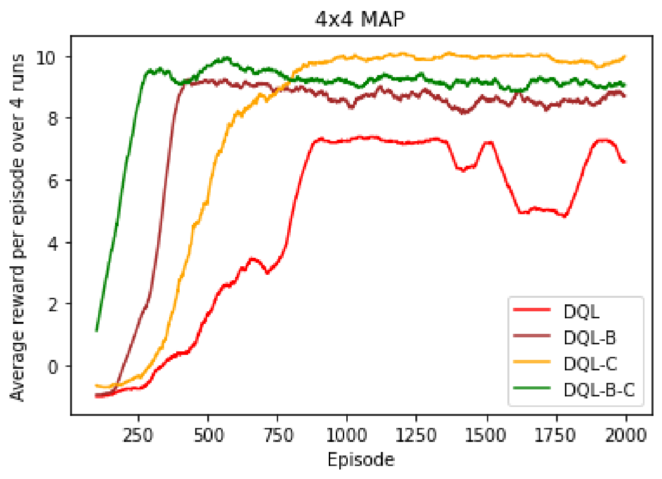


Figure 4.4: Results 4x4 map

As we can see at Figure 4.4, almost all the algorithms achieve the goal and converge after 2000 episodes. The first algorithm which learns the optimal policy after 250 episodes is DQL-B-C. In the other hand, DQL has some troubles to achieve the goal in 1 of the 4 runs. This map is small, and make sense that almost all the algorithms learn the path to the goal.

4.3 Results 6×6 map

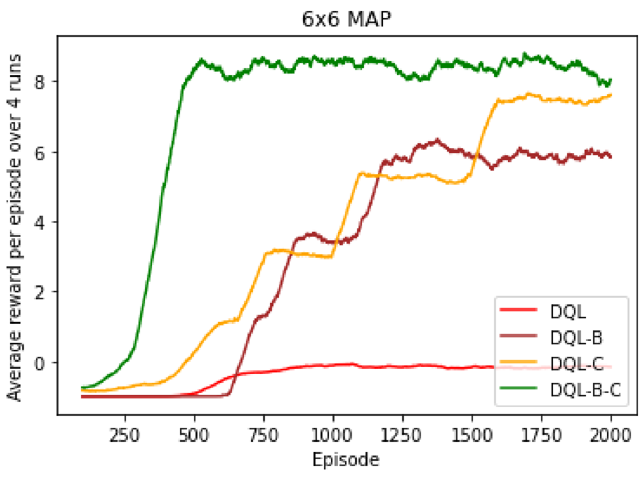


Figure 4.5: Results 6x6 map

In the 6×6 map results shown at Figure 4.5, DQL can't achieve the goal and only learns to survive, which means to receive 0 reward. In the other hand, DQL-B has learned the path to the goal in 3 of the 4 runs, which suggest that Boltzmann exploration it's a better strategy than ϵ-greedy. The others algorithms achieve the goal, the first algorithm which learns the optimal policy is DQL-B-C again.

4.4 Results 8×8 map

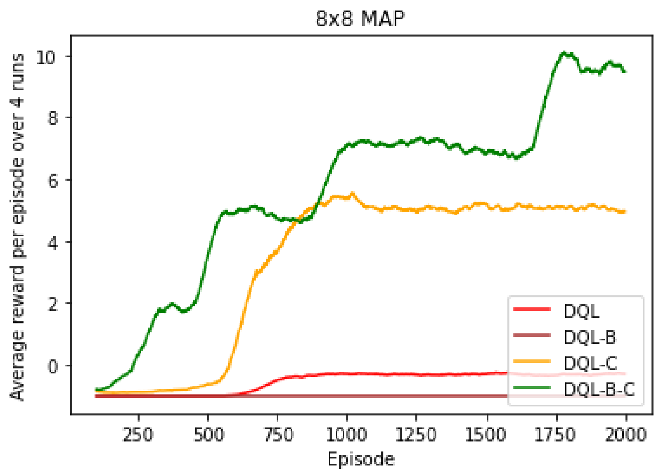


Figure 4.6: Results 8x8 map

Finally, in the 8×8 map results shown at Figure 4.6, only those algorithms with **Count-Based** exploration strategy, DQL-C and DQL-B-C, achieve the goal, consequently, we can conclude that this additional feature helps to solve sparse-reward environments as we expected. One more time, the first algorithm to learn the optimal policy is DQL-B-C, which is the only algorithm that achieve the goal in 4 of the 4 runs.

4.5 Q values of DQL-B-C

DQL-B-C is the algorithm which learns faster, but it's important the relation between the bonus reward and the goal reward. The reason it's simple, if the agent receives more reward from transitioning between states it will never reach the goal since it will get a bigger bonus reward than the goal reward. It is true that all the bonus reward will converge to 0, but it can pass a lot of time before this fact happens, so, if we want to make our agent's exploration optimal we must adjust the bonus reward.
If the bonus reward is adequate, the Q values will converge, as they have done with test environments. In Figure 4.7, Figure 4.8 and Figure 4.9 for maps 4×4, 6×6 and 8×8 respectively, the final Q values generated by DQL-B-C at test environment are shown.

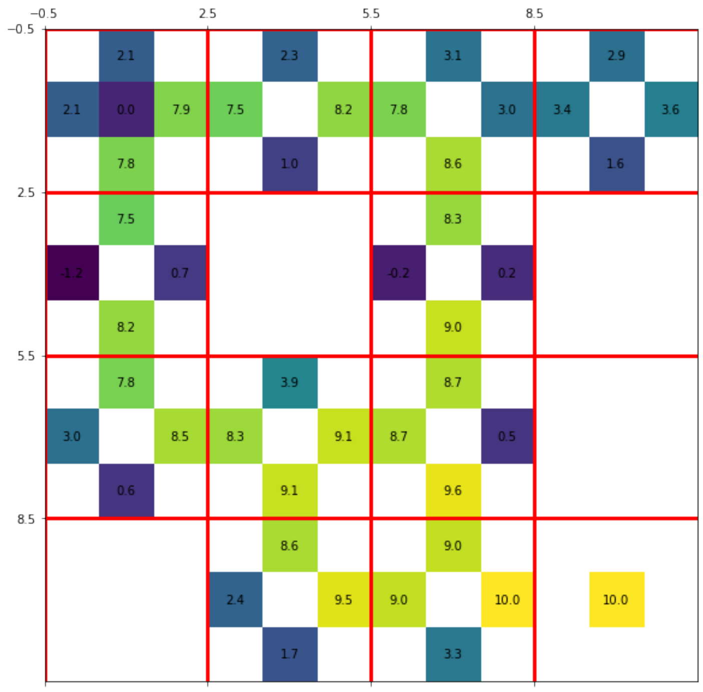


Figure 4.7: Q values generated by DQL-B-C at 4x4 map

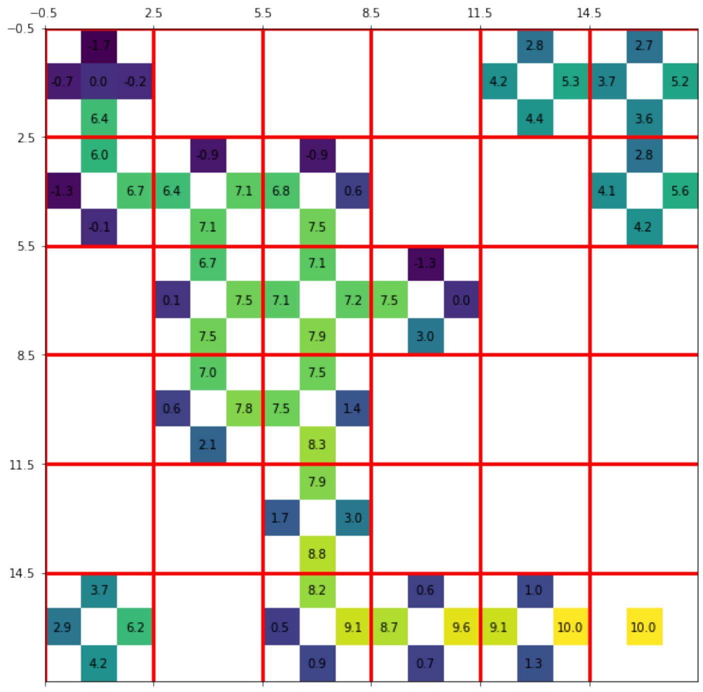


Figure 4.8: Q values generated by DQL-B-C at 6x6 map

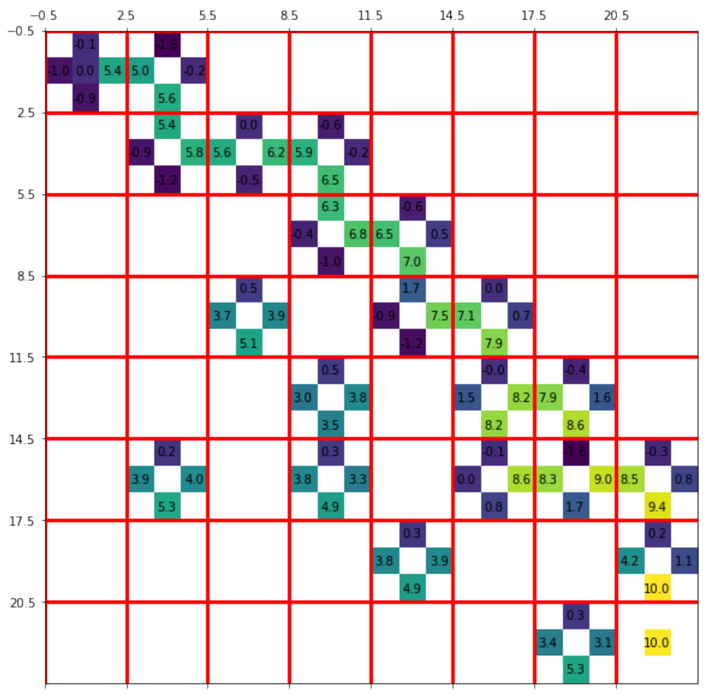


Figure 4.9: Q values generated by DQL-B-C at 8x8 map

CHAPTER 5

Conclusions and Future Work

5.1 Conclusions

We presented and improved the Frozen Lake environment offered by OpenAI. We also programmed additional features such a random map generator, new multiples state representations and a new displayable method to print the Q values.
Furthermore we explained very precisely how Deep Q-Learning works, we defined their pros and cons and the two key features which it makes so powerful, the experience replay and the target network.

We also implemented DQL and proposed two important changes to improve the algorithm in sparse-reward environments. The first proposal was to change the ϵ-greedy strategy to a **Boltzmann exploration**, the second proposal was to introduce an intrinsic exploration bonus r^+ using a **Count-Based exploration**.
In order to prove the improvement of our changes we implemented four different versions of DQL. The DQL, DQL-B, DQL-C and DQL-B-C, where a B in the names means that this algorithm has Boltzmann exploration and a C in the names means that this algorithm has Count-Based exploration. We compared the previous algorithms in three different environment test, where the environments were increasingly more difficult due their increasingly size.
Finally, the results shows us three important facts about DQL in sparse-reward environments:

- Boltzmann exploration works better than ϵ-greedy (Figure 4.5).
- The most significant improvement was made by the implementation of Count-Based exploration (Figure 4.6).
- The DQN-B-C was the algorithm that obtained the best results, which implies that the union between Boltzmann exploration and Count-Based exploration obtained the best results (Figure 4.4 Figure 4.5 Figure 4.6).

We speculate that Boltzmann exploration works better for a couple of reasons. The hyperparameters chosen for ϵ-greedy probably are not optimal and maybe with other configuration ϵ-greedy can work better. Also, it is more difficult to calibrate the hyperparameters of ϵ-greedy for every map due to the increasing size of each one. More explicitly, if we are working in small maps like 4x4, the ϵ-greedy strategy will find easily the path to the goal, but once we start increasing the map's size, ϵ must decay more slowly since if it's not the case it will never find the goal. To sum up, we think that Boltzmann exploration works better because it's a more suitable method for the test environment.

Furthermore, we speculate that the most significant improvement was made by the implementation of Count-Based exploration because this provides some knowledge to the agent about how many times it sees each state. For this reason, the more it sees a state the less interested it will be in, allowing the exploration for other states.

5.2 Future work

The next step after this simpler approach could be implementing this algorithm in others environments such Montezuma's Revenge. However, others challenges will appear, such counting more complex states or it will be even impossible to count those states because they will be too much. So, other models must to be applied in order to create, for example, a pseudocounter, like Ostrovski et al. (2017) does. An other option could be creating a model to get the probability of seeing a state, and in base of that fact generate a bonus reward, this last approach has been presented by M. Bellemare et al. (2016).
Another interesting path with more practical applications could be robotics. Since this reinforcement learning technique for sparse reward problems can be very useful in order to train, for example Riedmiller et al. (2018), a robotic hand which can move and organizes a different kind of objects.

Finally, we think that the most general way to solve this complex environments it's not by adding a bonus reward based on counts, because this implies to count states and represent it in a general way. We think that the optimal direction in order to solve this kind of problems is to create a model that produce this bonus rewards by itself based on the agent's experiences.

Bibliography

Bellemare, M. et al. (2016). 'Unifying count-based exploration and intrinsic motivation'. In: *Advances in neural information processing systems*, pp. 1471–1479.

Brockman, G. et al. (2016). 'Openai gym'. In: *arXiv preprint arXiv:1606.01540*.

Haykin, S. S. et al. (2009). *Neural networks and learning machines/Simon Haykin*.

Messias, J. V., Spaan, M. T. and Lima, P. U. (2013). 'GSMDPs for Multi-Robot Sequential Decision-Making.' In: *AAAI*.

Mnih, V. et al. (2015). 'Human-level control through deep reinforcement learning'. In: *nature* vol. 518, no. 7540, pp. 529–533.

Mnih, V. et al. (2013). 'Playing atari with deep reinforcement learning'. In: *arXiv preprint arXiv:1312.5602*.

Ostrovski, G. et al. (2017). 'Count-based exploration with neural density models'. In: *arXiv preprint arXiv:1703.01310*.

Paszke, A. et al. (2019). 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by Wallach, H. et al. Curran Associates, Inc., pp. 8024–8035.

Riedmiller, M. et al. (2018). 'Learning by playing-solving sparse reward tasks from scratch'. In: *arXiv preprint arXiv:1802.10567*.

Ruder, S. (2016). *An overview of gradient descent optimization algorithms.* URL: https://ml-cheatsheet.readthedocs.io/ (visited on 20/08/2020).

Simonyan, K. and Zisserman, A. (2014). 'Very deep convolutional networks for large-scale image recognition'. In: *arXiv preprint arXiv:1409.1556*.

Smelser, N. J., Baltes, P. B. et al. (2001). *International encyclopedia of the social & behavioral sciences.* Vol. 11. Elsevier Amsterdam.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT press.

Szegedy, C. et al. (2015). 'Going deeper with convolutions'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Tang, H. et al. (2017). '#Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning'. In: *Advances in Neural Information Processing Systems 30.* Ed. by Guyon, I. et al. Curran Associates, Inc., pp. 2753–2762.