Master Business Analytics

Master thesis

Performance of exploration methods using DQN

by

Tim Elfrink

July 30, 2018

Supervisor: MSc Ali el Hassouni

Second examiner: Prof.dr. Rob van der Mei

Host organization: Mobiquity Inc.

External supervisor: Dr. Vesa Muhonen

Business Analytics
Faculty of Sciences

VU UNIVERSITY AMSTERDAM

Abstract

In this thesis, we compare different exploration methods in deep Q-learning networks. To this end, we select a subset of existing exploration strategies. This selection contains algorithms of Deep Q-Network and we compare ϵ-greedy, annealed, noisy networks and bootstrapped DQN. The networks are described, details discussed and the differences explained. We compare the different algorithms in different test environments. We show that in different environments the score is depending on the exploration method. As an end result of this research, we summarise the results in a framework as recommendations that can be used by companies to decide which algorithm to use for different problems. This framework is based on looking at the difference in performance, hardware requirements and problem setting. We show that different methods perform differently depending on the environment and we explain why this happens.

Title: Performance of exploration methods using DQN
Author: Tim Elfrink, t.elfrink@student.vu.nl, 2521040
Supervisor: MSc Ali el Hassouni
Second examiner: Prof.dr. Rob van der Mei
Host organization: Mobiquity Inc.
External supervisor: Dr. Vesa Muhonen
Date: July 30, 2018

Business Analytics
VU University Amsterdam
de Boelelaan 1081, 1081 HV Amsterdam
`http://www.math.vu.nl/`

The only stupid question
is the one you never ask.

RICH S. SUTTON

Preface

After witnessing Alpha Go's [19] win against the best human Go player in the world, my interest in reinforcement learning was sparked. As the number of possible moves in a game is larger than the number of atoms in the universe, the computer can not calculate all the possibilities. Despite those limitations, it had beaten the number one player which was never been achieved before. So how did Alpha Go do this and what techniques have been used to achieve this accomplishment? The main techniques that made this breakthrough possible was using reinforcement learning and other techniques like deep learning. Throughout my study, I had one course that explained this algorithm, but I knew there was so much more to discover in this field of machine learning. I started to experiment with implementing my own algorithms and soon found out this is a subject I wanted to spend my 6 months of research on.

Acknowledgements

This thesis is written as part of the requirements for obtaining the Master Business Analytics at the VU University Amsterdam. The goal of the Master's program in Business Analytics is to improve business performance by applying a combination of methods that draw from mathematics, computer science, and business management. The internship was performed at and sponsored by Mobiquity and I would like to thank Mobiquity for giving me the opportunity to complete my thesis. Especially the whole analytics team which provided me with guidance, motivation and a lot of fun. I would also like to thank Rob van der Mei for being my second reader and Coen Jonker to provide me great feedback on my report. Finally, I want to give special thanks to my external supervisor Vesa Muhonen and VU supervisor Ali el Hassouni. They have provided me with great guidance throughout the internship and useful feedback on the process and the report. I am very grateful for that.

Contents

List of Figures **8**

List of Tables **9**

1 Introduction **10**
1.1 Basics 10
1.2 Mobiquity Inc. 11
1.3 Reading guide 11

2 Background **12**
2.1 Reinforcement learning 12
2.2 Q-learning 13
2.2.1 Exploitation and Exploration 14
2.3 Deep Q-learning 15
2.3.1 Experience replay 16
2.3.2 Periodical updating the Q-values 16
2.3.3 State representation in DQN 17

3 Methods **19**
3.1 Environments 19
3.1.1 OpenAI Gym 19
3.1.2 Chain 22
3.2 Exploration methods 23
3.2.1 ϵ-greedy 24
3.2.2 Noisy networks 24
3.2.3 Bootstrapped DQN 25
3.3 Evaluation 26
3.3.1 Score 27
3.3.2 Area under the curve 27
3.3.3 Speed 27
3.3.4 Integration 28
3.3.5 Environment fit 29

4 Experimental setup **30**
4.1 Network architecture 30
4.1.1 Atari 30
4.1.2 Bootstrapped DQN 32
4.1.3 Others 33

5 Results & Conclusion **34**
5.1 Gym 34
5.2 Chain 35
5.3 Atari 38
5.4 Framework 39

6 Discussion **41**

Bibliography **43**

List of Figures

2.1 Reinforcement learning [21] 12
2.2 Algorithm 1: Deep Q-learning with experience replay 17

3.1 Mountain Car 20
3.2 Breakout in Atari 22
3.3 Pong in Atari 22
3.4 Chain 23
3.5 Shared bootstrapped network 26

4.1 DQN network Atari [3] 32

5.1 Results Mountain Car 35
5.2 Plots for different chain lengths 36
5.3 Results Pong 38
5.4 Results Breakout 39

6.1 Results rainbow [9] 42

List of Tables

5.1 Results Mountain Car 35
5.2 Results Chain 37
5.3 Results the mean of the Chain 37
5.4 Results Atari 39
5.5 Framework: How to choose which algorithm 40

1 Introduction

Reinforcement learning is learning what to do - how to map situations to actions - so as to maximise a numerical reward signal [2]. The learner is not told which actions to take but instead must discover which actions yield the most reward by trying them [23]. Reinforcement learning is a unique part of the machine learning area. It differs from other machine learning methods as it is optimising for the future reward instead of the current one. Optimising for the future can be extremely useful in a lot of use cases and hard to achieve in a different way. Let us look at a chess game. One could make a simple program that evaluates of every board, the outcomes of all possible moves. When this program only evaluates the board one step ahead it cannot see certain things. In chess you can, for example, sacrifice an important piece to get a better position in a couple moves away. This cannot be seen by our simple program as it would never do such a move because it would not be a better position in the next move. This can be a perfect example where reinforcement learning can come into play as it optimises for the future reward (win or lose) instead of the current board evaluation. This way of solving games has been shown to be better than humans in different games such as chess and Go [20].

1.1 Basics

Reinforcement learning can be applied to a large number of different problems. In order to apply reinforcement learning to a problem, three parts of the problem need to be defined clearly. As long as there is a known reward function, a state and actions which can influence the outcome of the goal. These three components together form a reinforcement learning problem. As an example, we will use a fitness app which where people want to lose weight. For this app, the amount of lost weight is the goal that you want to maximise and all the notifications the app gives you, are the actions. The state is the different kind of information the app collects of a user, its activity and

app usage. In chess the setup of a reinforcement learning algorithm would be to have an action of all the possible moves, the reward is a win or lose and the state is the current situation of the board.

1.2 Mobiquity Inc.

This research is sponsored by Mobiquity Inc., which is a professional services company that creates compelling digital engagements for customers across all channels. Mobiquity's core business is to make engaging mobile apps in combination with consultancy. With their 5 different end-to-end services; strategy, experience design, product engineering, cloud services and analytics, Mobiquity looks at innovation all the time. They are interested in how reinforcement learning can help their clients, how to integrate its products and what to implement. The aim of this thesis is to support quick decision making for companies that want to implement reinforcement learning. A framework will be provided which they can use to map their problem onto which algorithm will be the best to apply. Different aspects and performances will be looked at which will be described in the methods section. This is all done by giving the company access to the code, a research and evaluation report of the different algorithms. Also, it will give a good overview of all the different factors they should look at when thinking of implementing one.

1.3 Reading guide

First, we will discuss the background in section 2. We will discuss the main idea of reinforcement learning and the mathematical foundation behind it. After that the different methods will be explained in section 3. All environments will be discussed as well as the different models and the way we will evaluate the different algorithms. In section 4 we will explain how we implemented our algorithms exactly and in section 5 we will show our results and conclusions which contains the framework. Finally, we will discuss all further improvements that can be made in the discussion section 6.

2 Background

2.1 Reinforcement learning

Now that we introduced reinforcement learning on a high level and explained where it is applicable let us go a bit deeper into how it works. A reinforcement learning problem as a sequential decision-making problem under uncertainty [5]. This can be written down in the form of a Markov decision process. It has state space which is denoted as S, a set of actions of the agent A, transitions from state $s \in S$ to state $s' \in S$ with action $a \in A$ at time $t \in T$ denoted as a_t and which is denoted as $P(S_{t+1} = s'|s_t = s, a_t = a)$. Next to that there is a reward function $R(s, s', a)$ that returns the reward which is given from s to s' with action a. This is shown in figure 2.1.

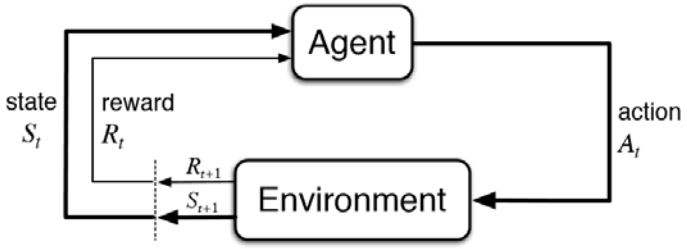


Figure 2.1: Reinforcement learning [21]

There are two different types of models which will determine the action an agent takes. A model-based algorithm learns the transition probability from each state-action pair. A model-free algorithm relies on a trial-and-error strategy, where it keeps updating its knowledge. A policy π denotes a method by which the agent determines the next action based on the current state. Instead of a reward function, we define a value function $V_\pi(s)$ that maps the current state s to the expected long-term return under policy π. In the next section, we

will go deeper into this value function. But first, we need to explain that there are two different types of policies, on-policy and off-policy. An on-policy agent learns the value based on its current action A derived from the current policy, whereas its off-policy counterpart learns it based on the action A' obtained from another policy [22].

2.2 Q-learning

Ideally, the reward function can be found, but in most cases that is infeasible. Due to randomness and other factors which cannot be influenced by the actions, an exact reward function cannot be found in most real-life problem settings. This is why a value function is introduced. It calculates the expected reward based on the state and action. This value function needs to be learned based on the collected data. There are multiple ways of learning a value function and one way is known as Q-learning [30]. For every state-action pair, it has a Q-value which is the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state [11]. Q-learning is an off-policy and model-free reinforcement learning algorithm. The following equation is the rule how the value is updated of a state-action pair.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{A_t} Q(S_{t+1}, A_t) - Q(S_t, A_t)] \tag{2.1}$$

After every time step t the Q-function can be updated by the new information it obtained. The first part of the function contains simply the old value of the Q-value. The learning rate α is a value between 0 and 1 and determines to what degree new information influences the Q-value. This α is multiplied by the actual reward, R, added to the estimate of the optimal future value $Q(S_t, A_t)$. This estimation is adjusted by the discount factor γ. The discount factor γ determines the importance of future rewards. It is also a number between 0 and 1. When the discount factor is close to zero, it only cares about the current rewards. When it is close to 1 it will try to optimise on the long-term reward. Next, to choosing the appropriate α and γ there needs to be selected initial Q-values for every state-action pair. There are multiple ways to initialise Q-values. How to choose the best approach depends on the problem statement. If exploration is

encouraged in the beginning, high initial Q-values are chosen assuming that always the highest Q-value is selected. This is because an update of the Q-value will lead to a lower Q-value. The next time another action will be chosen until the Q-values will converge to their (local) optimum value. When choosing low initial Q-values, it can happen that the exploration will be minimal.

Now we have Q-values that represent our current estimation of the reward given a state and a policy. Next to that, we have a Q-function that is updating the Q-values based on new data. These values are all stored in for example a matrix which has a Q-value for every state-action pair. This is called the tabular implementation of Q-learning [7]. After every step, the Q-function will be updated with all the corresponding effects for the Q-values. The more data it processes the more accurate predictions of the policy will be. The higher the Q-value, the higher estimation of a reward. When all Q-values are low in a certain moment, the Q-function cannot estimate which action will lead to a high reward.

2.2.1 Exploitation and Exploration

Depending on the environments, just performing the action which has the highest Q-values will likely converge to a local optimum. For example, when in the first step a certain state-action pair has a high Q-value, it will never perform another action in that state. To avoid this there are multiple ways to perform "exploration". The most implemented way to do this is having a random element when deciding which action to take. Instead of looking at the Q-values a fraction ϵ of the times a random action is performed. With ϵ between 0 and 1 This is called the ϵ-greedy exploration method. Next to the ϵ-greedy exploration method, there are various ways to perform exploration. In section 3.2.1 we explore this.

A common dilemma in reinforcement learning revolves around balancing exploiting known information and exploring the problem space to find new information [2, 26]. In this research, this will be the main focus of the different algorithms that are being compared. All different algorithms have a different method to deal with this dilemma.

2.3 Deep Q-learning

Before starting to compare the different exploration methods we should first set up an algorithm that will solve the problem with at least one of the different exploration methods. The algorithm we have chosen in this thesis is a deep Q-Network (DQN) as this algorithm has proven to work in different environments. The different exploration methods are all modification of this original algorithm.

Deep Q-learning is a form of the classic Q-learning model. At every step, the state is evaluated and will give a Q-value which approximate the reward of each possible action. Traditionally Q-learning was designed to have a value for every state action pair which called tabular Q-learning. But this is not extendable when the state space is increasing as the possible pairs are growing exponentially. For example, when having 100 different binary variables which define the state space and 10 different actions to take it means that there should be more than $1.26 \cdot 10^{31}$ different Q-values, which also needs to be updated at every time step. Also a lot of problems do not have binary variables as input space but have more values for a variable or even continues. This kind of problems makes the Q-learning explode and not feasible for a lot of problems. To avoid these limitations a machine learning technique is used to approximate the rewards at every action. Machine learning is applied to makes a supervised model with input state and output action. In this case, a deep neural net is used as a machine learning model, hence the name Deep Q-learning. The model learns from previous experience in mini-batches, which avoids the model to train after every step. This approach makes sure the algorithm is time efficient and stays feasible, now the only things that will be stored are all the history state-action pairs and the model which are for a deep neural network only the different weights and not all possible state-action pairs of the model. The input of this model is the state-action pairs and is optimised on the obtained reward. The output of the model are the Q-values for the different possible actions.

Next to having the ability to tackle problems with a bigger state space, not only discreet but also continues variables, it is also more scale-able. The method of using a nonlinear function approximate such as a neural network which represents an action-value is known to be unstable or even diverge [27]. There are two main ways to avoid this behaviour. The first one is the use of experience replay

and the second one is periodically updating the Q-values. These improvements and others will be discussed in the next sections.

2.3.1 Experience replay

Every time step the agent stores the obtained data. The data which is stored is $e_t = (s_t, a_t, r_t, s_{t+1})$, with experience, state, action, reward and next state at time t correspondingly. The dataset is defined as $D = \{e_1, ..., e_t\}$. To introduce a new mechanism to remove correlations in the observation sequences and to smooth over the data distribution a sample is taken from D, $(s, a, r, s') \sim U(D)$. This uniform sample is taken when the algorithm is updating the Q-values in mini-batches. The last N experiences are stored which is called the replay memory. The replay memory does not make any disquisition in the importance of the different transitions and overwrite the oldest transitions with the newest when the memory buffer reached N. This is also the case with the mini batches. So although there can be made some improvements, this solution tackles the main problems.

Prioritized Experience Replay

One improved that can be made to learn more from some transitions than from others is to use prioritized experience replay [18]. This can be done to look at transitions which do not fit well. Instead of uniformly sampling from the replay buffer, there is been looked at the error which is made by the value function. The bigger the error the higher probability to get in the mini-batch. The selection is efficiently done by having a binary tree which has the error for each index of the memory buffer which does not slow down the algorithm by much.

2.3.2 Periodical updating the Q-values

Another way to improve the stability and especially reduce the observation sequence is to introduce two different networks: Q and $\hat{Q}$. This is called Double Q-learning [28]. At every C number of updates, the network Q is cloned to $\hat{Q}$. This makes sure that the data is not including too recent observations in there Q-values. A direct consequence of that is that the current state is not influencing the predictions of the last couple of states. This works well as $Q(s_t, a_t)$ is highly correlated with $Q(s_{t+1}, a_t)$, so adding a time delay the Q-values will not be influenced by the recent observations.

2.3.3 State representation in DQN

The state can be represented in various ways when using DQN. It really depends on the available data in the environment. It can be observed data by sensors. This can be all data which represents the current state. Examples are temperatures of sensors, coordinates of certain objects or even screenshots of videos. There are even video games that can learn on the RAM[1] of the game, so the input state is an array with bytes [24]. All data that represents the state space can be helpful to obtain the most optimal Q-values. In this research, we will also look at the pixels of a screen and transform that to a format that can be processed by our DQN, such as an array of RGB-values[2].

Next to the already told differences of Q-learning and DQN there are some other details not discussed but explained in the original publication [14]. In figure 2.2 the full algorithm is described which we call deep Q-learning.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 2.2: Algorithm 1: Deep Q-learning with experience replay

An episode is a complete play from one of the initial state to a final state. In every episode, the same steps will be checked. The most

[1] Acronym for random access memory, a type of computer memory that can be accessed randomly.

[2] RGB (red, green, and blue) refers to a system for representing the colours to be used on a computer display.

important steps will be explained. First the action will be decided, this is done based on the highest corresponding Q-value or there will be chosen a random action. This action will be performed and this will be stored in the replay memory. Then the network is trained by performing a gradient descent based on the returned reward.

3 Methods

In this chapter we discuss and explain all the different environments we used in this research, the different exploration methods we compare and how we will evaluate which algorithms are better and in which ways.

3.1 Environments

There are 3 different types of environments used in this research. Classic controlling, game solving by screenshots and an experimental setup to measure exploration in a policy. Now we shall take a look at each in more detail.

3.1.1 OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents in different ways, from walking to playing games like Pong or Pinball [4]. It is an open-source library that can be used to compare different reinforcement algorithms in different designed environments. In our research 2 different types of environments are compared: Mountain Car and the Atari 2600 environment. The first one is a classical controlling game. There is a small factor of randomness [8] and after some exploration it should be easy to solve. Atari games are different, there is some randomness in the games and the input size is bigger. In the next paragraphs, the differences will be explained more.

Mountain Car

In our research, the Mountain Car environment is part of the classic controlling problems. The goal is to drive up a big hill with a car. It has to build up its own momentum to be able to do this. The version which is used is called: MountainCar-v0, this specific version of the environment, has the following description:

> A car is on a one-dimensional track, positioned between two "mountains", see figure 3.1. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum [15].

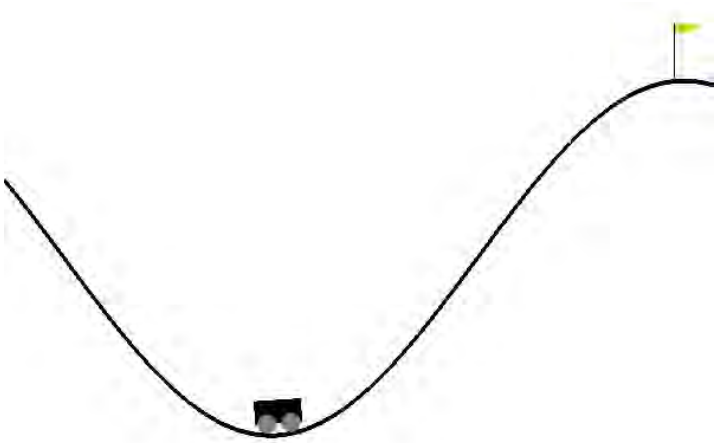


Figure 3.1: Mountain Car

The input values of the model are the current position and the velocity. To create an extra level of difficulty the car is placed at a random position without velocity. The episode is terminated if 200 iterations are reached or if the car reached the top. The actions which can be taken at every timestep are push left, push right and no push.

Atari 2600

Based on the popular old school games on Atari, OpenAI implemented a selection of the games in the Gym environment. specific description:

> Maximise your score in the Atari 2600 game. In this environment, the observation is an RGB image of the screen, which is an array of shape (210, 160, 3) Each action is repeatedly performed for a duration of 4 frames [12, 1].

There are different games which can be played and they represent a set of different problems. Because the input space and controls are all the same, there can be a generic model applied which can be used for all different games. Games vary from Pong to Pacman. They all have different objectives and different ways to achieve high scores. When model-free models are used which achieve good results, it can be said that they perform well in different environment settings.

In this research two Atari games are tested, Breakout and Pong. In Breakout, a layer of blocks is in the top third of the screen. A ball travels across the screen, bouncing off the top and side walls of the screen. When a brick is hit, the ball bounces away and the brick is destroyed. The player loses a turn when the ball touches the bottom of the screen. To prevent this from happening, the player has a movable paddle to bounce the ball upward, keeping it in play. Rewards are accumulated when the ball breaks a block and it stops when the player misses the ball and when all the blocks are broken. Pong is a game that simulates table tennis. The player controls paddle by moving it vertically across the left or right side of the screen. They can compete against a computer that is controlling the second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The goal is for each player to reach 21 points before the opponent. Points are earned when one fails to return the ball to the other. Two illustrations of both games are shown in figure 3.2 and 3.3.

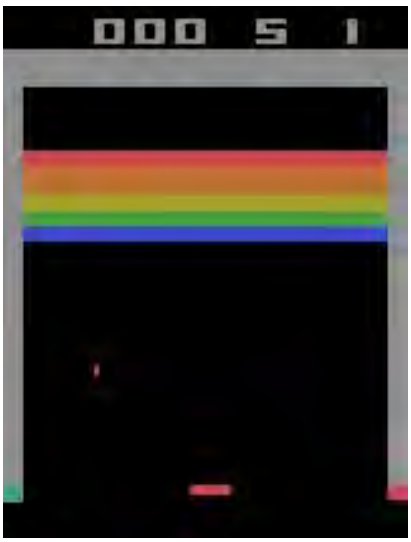


Figure 3.2: Breakout in Atari

Figure 3.3: Pong in Atari

3.1.2 Chain

This third environment is specially designed to test whether different models show signs of deep exploration [17]. In this environment, a Markov chain is made with length N with $N > 3$. The agent has to go left or right in every step and starts at state 2. At state 1 and N the agent receives a reward of 0.001 and 1 respectively, see figure 3.4. The episode ends after $N + 9$ steps. The goal is to maximise the cumulative reward of each episode. The greater N is, the less likely for the algorithm to reach state N. The most optimal score is obtained by only go to the right and stay in N until the end. When the algorithm reaches that state it will be rewarded with a big reward and the algorithm will find a way to go back to that state. The hard part is to find that state in the first place. This is why the algorithm shows signs of deep exploration when it finds this state after a time. Deep exploration means exploration which is directed over multiple time steps which are indeed needed to solve this environment.

Figure 3.4: Chain

3.2 Exploration methods

In this section, different exploration methods are compared. The different models are distinct in the sense of how they estimate the Q-values and interpret those values. Because the Q-values are estimations of the reward and those estimations are especially in the beginning not accurate and should not be trusted that much. There are different ways to do that, but the main idea is that you should exploit the information which you have already learned or explore new states in the hope you will result in a greater reward. Another reason why it is important to choose different actions is that the algorithm can be in a local minimum. To avoid this exploration can be a very useful way to go out of this minimum.

There are traditionally two different categories of exploration: exploring undirected and directed [31]. The undirected exploration relies on following random moves instead of looking at the given Q-values. The most commonly used implementation of this method is the ϵ-greedy method which will be explained in this section. The directed exploration has tracks different values which helps it determine if it should exploit or explore. There are 3 types: frequency based, recency-based and error based. If we look at the recency-based we will check the Q-values and take in the recency into account. It will give more recent information more value than old information. If a certain action is not chosen in a long time it might want to explore this action again. Directed exploration brings an extra complexity to the problem as there has to be recorded and/or calculated information about one of the three different types.

In recent years there has been a lot of research being done in the field of reinforcement learning and there are two recent papers that show more exploration than the traditional papers, namely: noisy

networks and bootstrapped DQN. We will explain both of these methods in this section.

3.2.1 ϵ-greedy

As already explained the different Q-values in the algorithm are an estimation and might also converge to a local minimum. The first problem can be solved by gathering more data and improving the neural network by training it on that data. The second one can be solved by using the following method. Performing a random action completely independent of the Q-value. If you do this often it will reach state action pairs which would not be reached otherwise. The trade-off here is to perform this random action with probability ϵ and perform the action of the policy with probability $1 - \epsilon$. This is called ϵ-greedy. The higher ϵ the more exploration and the lower ϵ the more exploitation is performed. Sometimes you would be doing a random action instead of doing the optimal one, independent of the Q-values.

There are multiple implementations of this method. The most used one is to keep ϵ low and constant throughout the experiment. Another way is to use linear annealed exploration. It will start with a high ϵ (i.e. 1) and will decrease a small value linearly over every step. After a set number of steps, it will stay at a fixed ϵ. This implementation makes sure that there is a lot of exploration at the beginning of the experiment and will exploit all that information in the end. This can be useful as you first want to explore the state space and after you have a good view of the optimal actions.

3.2.2 Noisy networks

Noisy networks is an implementation of the original DQN with different dense layers. Those layers are replaced with a new kind of layer which is described in noisy networks for exploration [6]. The main idea is that the layers are getting parametric noise added to the weights of the layer. This noise is influencing the Q-values which causes that the predicted value is different than in a normal DQN network. The randomness has a direct influence on the Q-values. This means that when the algorithm is not confident between different possible actions it will be more random than when it is confident. When the Q-values for an action are high without noise, the noise will not influence this too much because it will still be higher than

other actions. For this reason, the algorithm is exploring more when not certain and follows the optimal policy when is more confident.

The dense layers are replaced by the noisy layers according to an implementation by Andrew Liao [10].

$$y = wx + b \tag{3.1}$$

We transform the normal layer

$$y \stackrel{\text{def}}{=} (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b \tag{3.2}$$

Where $\mu^w + \sigma^w \odot \epsilon^w$ replaces μ and $\mu^b + \sigma^b \odot \epsilon^b$ replace the bias in the normal dense layer. As in the original for noisy networks the DQN algorithm is chosen to use factorised Gaussian noise. Where $\odot$ represents element-wise multiplication. This function is used to generate the ϵ values, the noise. The μ values are uniformly initialised values $U(-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}})$. The σ values are initialised as a constant value $\frac{\sigma_0}{\sqrt{p}}$ with $\sigma_0 = 0.4$. These values are chosen as described in the original paper. The parameter are chosen carefully for a specific problem. This requires a parameter optimisation process.

3.2.3 Bootstrapped DQN

Bootstrapped DQN [17] empowers the different strengths of single DQNs. The idea is that you start with k different DQNs. All are initialized with random values in the networks. During each episode, one of the k networks is selected and the actions will be performed by the optimal Q-value of that network. After every step, there is a probability p that will determine if that state, action, reward pair is added to the memory for each of the k networks. Because all networks are different the network will reach different state spaces which are shared with a fraction p with all other networks. So, if a certain action results in better results for one network it will help the other networks which it is added to. The network uses that information in the next episodes. This loop will make sure that all networks will have this information when the time passes. With actions that have negative results, it's a bit different. That information is not shared as much as the positive actions as the Q-values for those actions will be low when it is already in the memory of a network. This behaviour of the collaboration between networks can lead to a positive effect of score performance, but also has some memory problems. It uses k

times as much networks which lead to slower processing time. When the networks are parallelized it only has an increase of 1.2 times a default DQN which is acceptable in most situation. This is possible as there was a shared network and only at the last layer a split between the different heads. Every iteration had an effect on the shared layers and fraction p of the k heads. This is shown in figure 3.5.

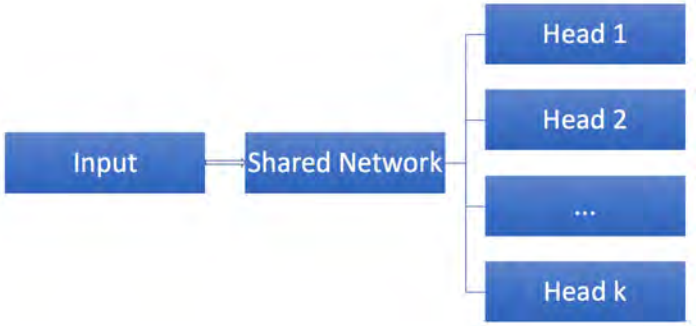


Figure 3.5: Shared bootstrapped network

This is done to make use of the exploration advantages of the bootstrap DQN, but not to make the network $k * p$ times slower. With the shared network, the authors of the original paper show that the performance of the scores improves more than the extra consumed time. An efficient way of storing the data also does not make the algorithm slower.

3.3 Evaluation

We compare different exploration methods based on different criteria. These criteria are described in this section. These criteria should result in an overview which Mobiquity can use to evaluate which algorithms to use in different business cases. We have in total 5 different elements we will use in our framework to evaluate the different algorithms. The first 2, score and area under the curve, will be explained in this section and the obtained results will be discussed in the next section. The other 3, speed, integration and environment fit, will be discussed in this section. These criteria do not need any results so we will already explain how the different algorithms perform in it.

3.3.1 Score

Every environment has a reward function. You can look at the maximum score that is obtained over time. This can be a good indicator of the potential of a certain algorithm. All scores are aggregated to the mean of the last 100 episodes. This is done to really see the trends and not just accidental behaviour.

3.3.2 Area under the curve

Usually the performance of a game is fluctuating over time, ideally increasing. The area under the curve (AUC) can help in quantifying the performance over time in one single number. It looks at all the scores over time and adds them up from timestamp 0 to n. This value can be good when you compare the overall performance of an algorithm and not the maximum score. When an algorithm is learning fast the AUC will also be higher when reaching similar maximum scores, than an algorithm with a slow learning curve. This can be good to know when you want to train an algorithm with fewer iterations.

3.3.3 Speed

Training speed is an important factor when implementing algorithms in business perspectives. Sometimes it is not that important to get the optimal answer as long as it does not take a lot of time to get. In other examples, it does not matter how long it takes as long as the answer is the best. Also, there can be some hardware limitations that will result in different kinds of algorithm requirements.

All simulations are done on a shared cloud environment on Amazon Web Services (AWS). This gives the limitation that the speed on a certain machine is also dependent on other AWS users. Because of this we will refer to other results of the original papers when discussed and explain the speed in a theoretical way.

In theory, the speed difference of the annealed, greedy and noisy methods are more or less the same. The whole algorithm is the same except the already discussed variations. Maybe we can say that annealed exploration can be a fraction faster than greedy as there is more often a random move instead of a prediction of a model. This can be safe some time but this is negligible.

The bootstrap algorithm is slower than all other algorithms. This is because the model is just bigger, which results in slower training

and predicting. It can the number times as slow as the number of heads. But due to efficient implementation and parallelization over multiple cores, this can be reduced. Furthermore, when the network has a shared network before the splitting of the heads like how we implemented it in the Atari environment the speed is only a little slower than the original implementation of a DQN. The implementation K=10, p=1 ran with less than a 20% increase on wall-time versus DQN [17].

3.3.4 Integration

An important aspect for the business perspective besides the effectiveness of the different algorithms is how easy it is to implement our approach and our findings in their new and/or existing systems. If it wants to implement one of the different algorithms the start is to implement the normal DQN with an ϵ-greedy policy. So we can say that this is the most 'simple' algorithm. Immediately followed by the linear annealed variant of the ϵ-greedy policy. This just requires a couple of extra lines of code which is no effort in comparison with the whole system. The noisy networks come up next when it comes to complexity. An extra neural network layer has to be integrated and some parameter optimisation needs to be performed for new problems. This optimisation can be really important for the performance of the network. The most difficult algorithm to implement is the bootstrapped DQN. This requires to make decisions in the number of heads and the fraction that of information is shared between the heads. Next to that the most difficult integration part is to design and implement the network architecture. What part of the network should be shared and where should the network be split in the different heads.

Next to keeping in mind how the algorithms should be integrated, it is also important for the business whether an algorithm can be explained to stakeholders within a company. With bootstrapped DQN this is also the hardest after the noisy networks. The ϵ-greedy and linear annealed policy are relatively simple to explain compared to them.

3.3.5 Environment fit

Different algorithms can show different results in different environments. This is because there should be a fit for an algorithm depending on its environment. For some environments it is hard to find the optimal policy and there needs more exploration. ϵ-greedy has a constant level of exploration which does not change over time. This algorithm is the best for environments which do not need exploration. The linear annealed policy is for environments where there first needs to be obtained a lot of information to make the optimal decisions. Noisy networks show signs of little exploration. In the beginning, it is high as the weights are initialised with noise. After some time it does not show that much of exploration. bootstrapped DQN is specialised for environments where a lot of exploration is needed, but both can converge to an optimum after a lot of iterations.

4 Experimental setup

In this chapter, we will discuss how the different algorithms are exactly implemented in more detail and how it is made reproducible for future research.

In our setup, we have a DQN with 3 extra variations on it next to the original ϵ-greedy DQN. These variations are linear annealed, noisy networks and bootstrap exploration. Other than the described differences, the networks are exactly the same. All are run on the same type of machine on the AWS cloud. In all environments the simulation is run 3 times with different seeds[1]. The multiple runs are necessary because the initial values of the neural networks can be very dependent on the explored space. The weights bias the Q-function towards a set of actions and which might result in different future exploration. Also, the algorithm itself has stochasticity because it can perform action selection randomly. The final results are based on the maximum obtained scores of the different seeds.

4.1 Network architecture

In this research, 2 different neural network architectures are used. One is used for the Atari games and another model for the other environments.

4.1.1 Atari

The network used for the Atari environment is an exact copy of the network which is described in DQN [14]. There are some slight variations with the different algorithms which were explained in the methods section. The exact parameters used and implementations in this research are outlined in the next section.

The input of the DQN is a grey-scale image representation of 84 by 84 pixels. The 4 recent frames are feed into the network because

[1] 1,2,3 are the chosen seeds

this gives information in pictures which represents the direction of moving objects. This allows the network to observe the change in the pixels and so the movement of certain objects. The first hidden layer of the network is a convolutional layer of 32x8x8 filters with stride 4 with the input image and applies a rectifier non-linearity. The second hidden layer is another convolutional layer of 64x4x4 filters with stride 2, again followed by a rectifier non-linearity. The third hidden layer is another convolutional layer of 64x3x3 filters with stride 1, again followed by a rectifier non-linearity. The final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 based on the games we considered [13].

Now some parameters will be explained which are used in the network and their values of them in this experiment. In total there are 10^6 number of actions taken. The higher this number, the more information the network has and also the more likely the network will perform better. The learning rate, α, is set to 10^{-4}. As described at the beginning of the paper this rate determines how much of the new information is influencing the new target value. The discount factor, γ, is set to 0.99 which indicates how important future rewards are. Every 1000 steps the network is updated. The memory gets sampled to update the network every 4 steps with mini-batches of size 32.

Double DQN

This small variation in the DQN is implemented to boost the performance [28]. DQNs are known to overestimate the action values [25]. To avoid these two value functions are set instead of one. First there was a network for selecting and evaluating actions. In the Double DQN setting, there is a current network and an older one. The current network w, selects actions a and the older network w' is used for evaluation. Where I is the target update of the network.

$$I = [r + \gamma \max_{a'} Q(s', a', w)Q(s, a, w)]^2 \tag{4.1}$$

$$I = [r + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s', a', w), w') - Q(s, a, w)]^2 \tag{4.2}$$

Reward clipping

All Atari games have different reward functions. For some games, players can earn up to 10000 points and others only 10. Keeping these reward functions means training will be unstable. This is why all negative rewards are set to -1 and all positive rewards to 1. This is called reward clipping.

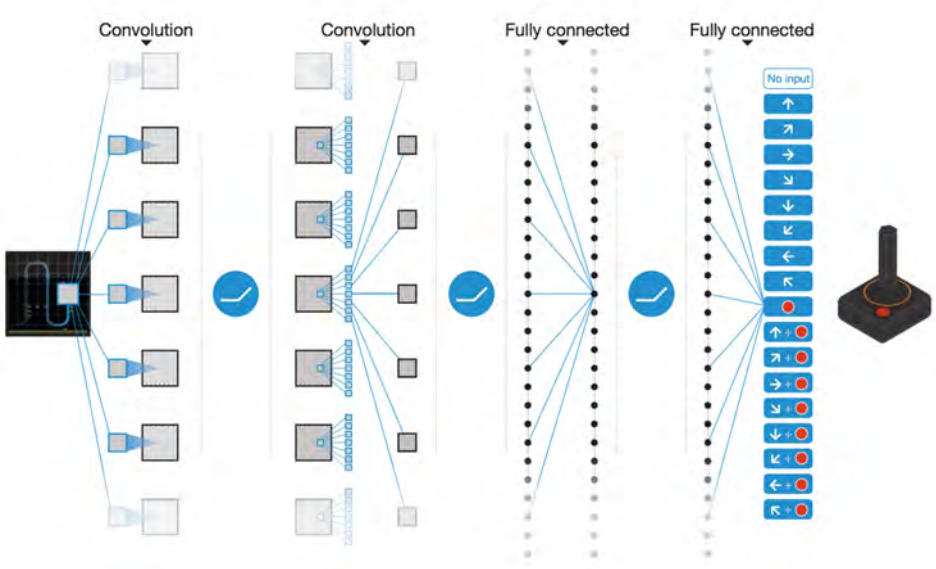


Figure 4.1: DQN network Atari [3]

4.1.2 Bootstrapped DQN

As already explained, for speed efficiency purposes instead of creating k different networks, part of the network is shared. After the final layer, the network is split into $K = 10$ distinct heads, each one is fully connected and identical to the single head of DQN. This includes a fully connected layer to 512 units. After that layer, it will split into the different heads of another fully connected layer. For every head, a Q-value for every action is generated. All the fully connected layers use Rectified Linear Units(ReLU) as a non-linearity. We normalise gradients $1/K$ that flow from each head.

4.1.3 Others

We have two different types of models. One is for the Chain and Mountain Car environments and the other is for the Atari games. The difference is in the number of layers, type of layers, size of layers and parameters of the agent.

In general all parameters are the same as in the Atari setup as long as we do not mention it. But the architecture of the networks is different. The network had a much simpler network of just one convolutional layer of 64 units with a ReLu and a fully connected layer with a linear activation function with the number of actions as a number of units. With bootstrapped DQN the first fully connected layer was replaced by $K = 10$ different networks. The learning rate was 10^{-3} and $\epsilon = 0.1$, which have proven to perform in online contests [16].

5 Results & Conclusion

In this chapter, we will discuss all the obtained observations and results. A framework will be provided to select the most appropriate algorithm for new problems.

5.1 Gym

As described, one of the tested problems is the Mountain Car. This problem needs a lot of exploration as it will only find a reward when going up the hill. This requires the algorithm to perform the tasks in a way that it will reach the top without knowing that that is the goal.

Figure 5.1 and table 5.1 show the results obtained after running the experiments. From these results we see that the bootstrap exploration strategy learns to climb uphill the fastest in comparison to the other exploration strategies. We can observe this during the first 150 iterations. Furthermore, we observe that the mean reward drops after iteration 150. A plausible explanation for this behaviour could be that the exploration rate is still too high after 150 iterations and as a result, the network keeps exploring which results in the policy changing over time. This also suggests that it might be beneficial to run the experiments for a larger number of iterations. Similar behaviour is seen in experiments with annealed and greedy exploration. Noisy exploration was not able to reach the top of the hill during our experiments. We suspect that this is due to the parameters and network architectures that were selected. These parameters and architectures were selected based on the original paper [13] that used this exploration method. This paper however tests noisy networks in an Atari environment while we test it in a different environment. Finally, we see that the linear annealed algorithm scores the best in both AUC and top score. This exploration strategy shows a clear exploration due to the fact that the average reward is fluctuating.

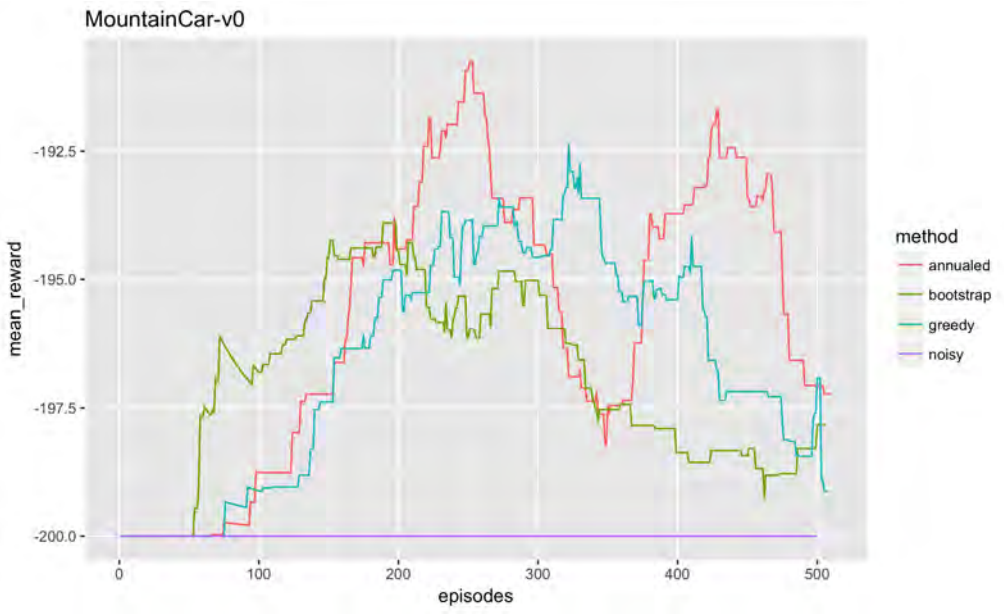


Figure 5.1: Results Mountain Car

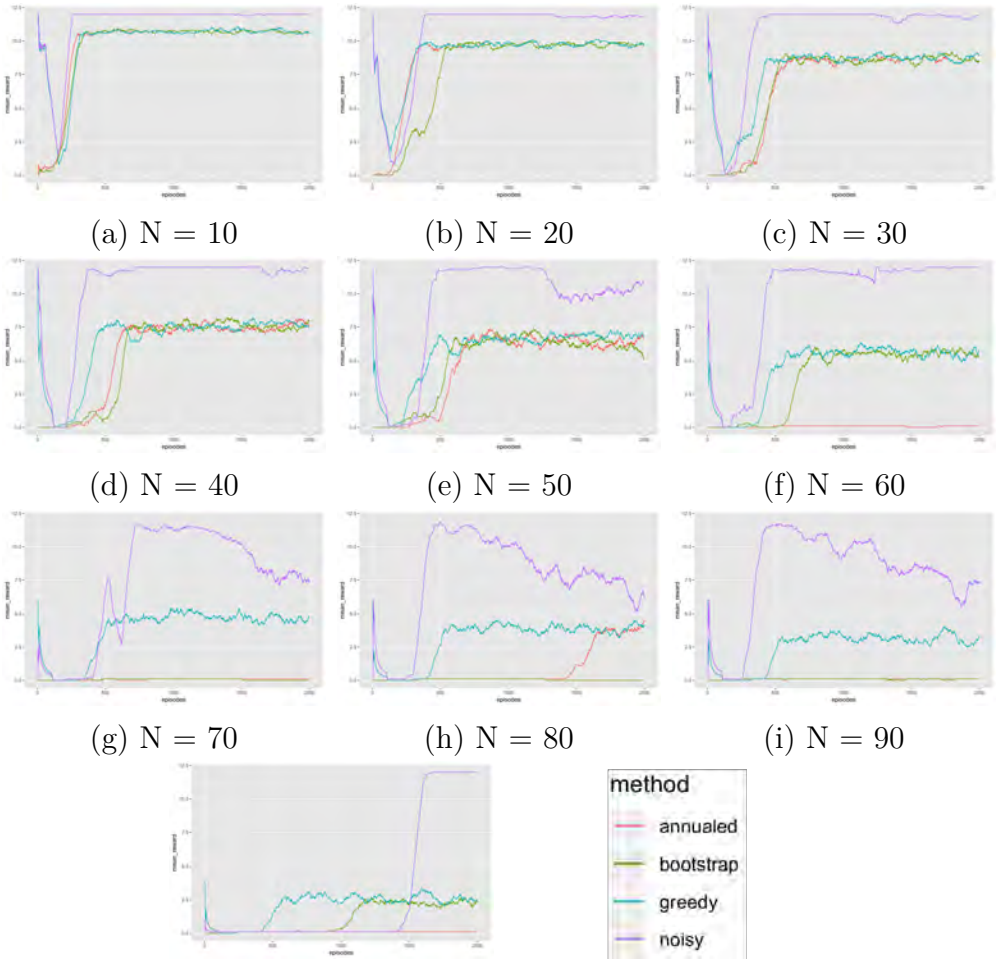
Method	AUC	Top score
Greedy	1719	-192
Annealed	2111	-191
Bootstrap	1540	-194
Noisy	0	-200

Table 5.1: Results Mountain Car

5.2 Chain

In figure 5.2 all the different results are plotted for the chain environment. We first see that the rewards are much higher when N is low. This is exactly what is to be expected as it is easier to reach a low N when doing for example just random moves. When the algorithm reached a certain state it is likely to keep going to that state as the

reward is much higher than in-state 1. We can clearly see this in figure 5.2j that when the noisy network found the optimal solution after 75% of the time and kept going to that state. In table 5.2 all the top scores and AUC's are presented. The greedy algorithm is performing well when N is low. We can see that the algorithm will reach a perfect store for $N < 50$. After that, the greedy algorithm does not perform enough exploration to obtain this optimum.



(a) N = 10 (b) N = 20 (c) N = 30

(d) N = 40 (e) N = 50 (f) N = 60

(g) N = 70 (h) N = 80 (i) N = 90

(j) N = 100 (k) Legend

Figure 5.2: Plots for different chain lengths

Method	N=10		N=20		N=30		N=40		N=50	
	AUC	Top	AUC	Top	AUC	Top	AUC	Top	AUC	Top
Greedy	19833	12	18071	12	15231	12	12676	12	11310	10
Annealed	19213	11.1	17224	10.1	13528	9.1	10975	8.1	9337	7.4
Bootstrap	18990	10.9	15610	10	13666	9.2	10770	8.2	9687	7.4
Noisy	22650	12	21206	12	21147	12	20861	12	18397	12
Method	N=60		N=70		N=80		N=90		N=100	
	AUC	Top	AUC	Top	AUC	Top	AUC	Top	AUC	Top
Greedy	8921	7.5	7531	6	6066	6	4932	6	3990	4
Annealed	128	0.1	131	0.1	1769	4.5	165	0.1	177	0.1
Bootstrap	7457	6	114	0.1	93.2	0.1	157	0.1	2203	2.7
Noisy	19447	12	14668	11.6	15130	11.9	15873	11.7	5645	12

Table 5.2: Results Chain

In table 5.3 we see that the noisy networks outperform the other algorithms, followed by the greedy algorithm. The linear annealed and bootstrap algorithms show similar results. Both are reaching the end, but cannot converge into the optimal solution. Even after they find a good solution they are exploring where they should have to exploit their knowledge.

Method	AUC	Top score
Greedy	10856.1	8.75
Annealed	7264.7	5.07
Bootstrap	7874.7	5.47
Noisy	17502.4	11.92

Table 5.3: Results the mean of the Chain

5.3 Atari

The results of Atari can be found in table 5.4. It includes the scores when playing using random policy [29]. This gives a good view on how the algorithms really improve and learn. In figure 5.3 and 5.4 the results are shown over time. We can see that in the Atari domain the bootstrapped DQN is outperforming all other algorithms. The performance of the other algorithms is similar across each other. In both environments we see that noisy networks are underperforming in the beginning, but they recover after a while. In both figures we can see that the angle of the slope is the highest in the end. We suspect that with more iterations it will outperform the linear annealed and greedy algorithm. This is also supported by the original paper of the noisy networks [6].

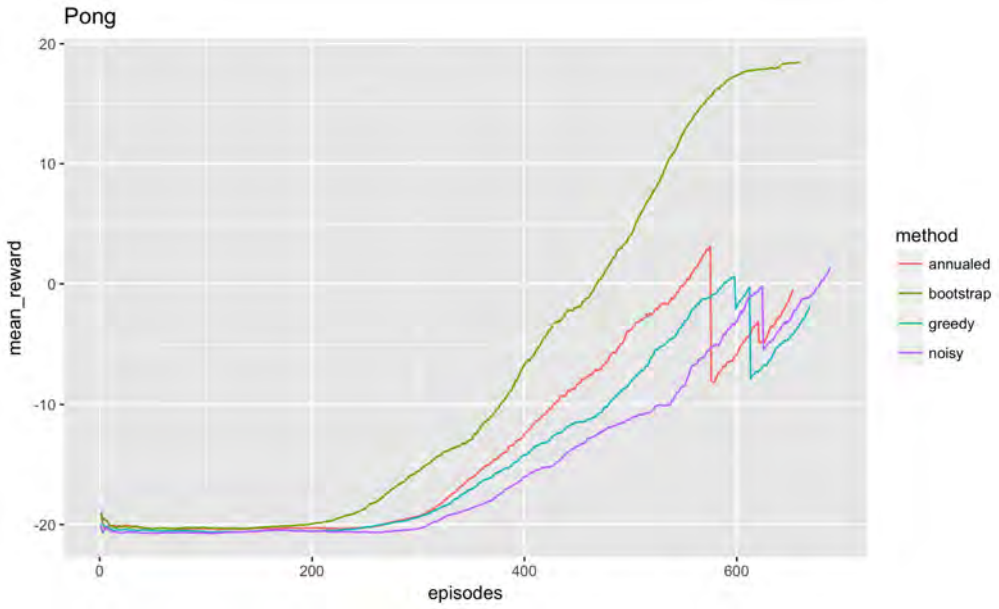


Figure 5.3: Results Pong

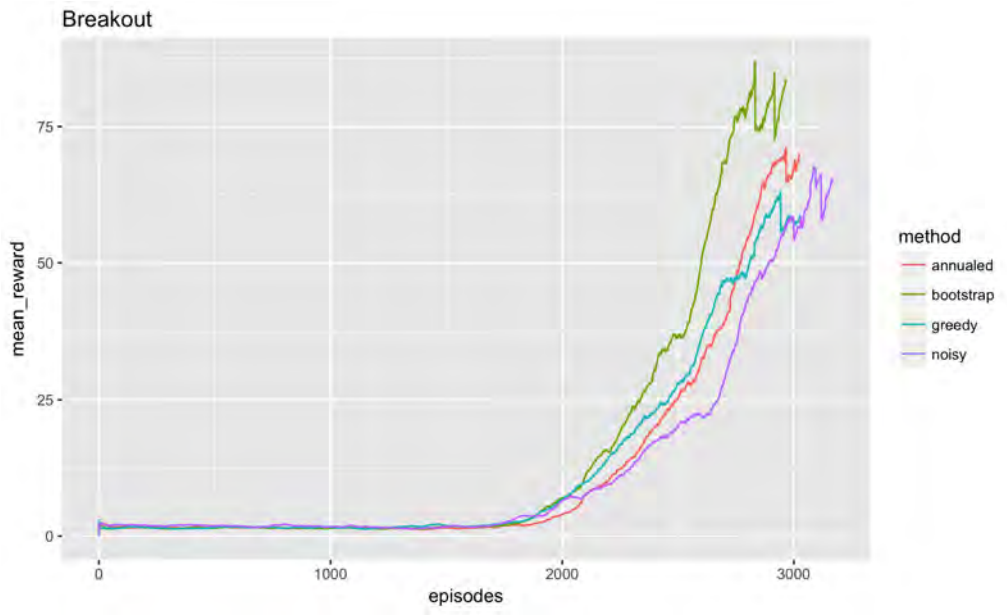


Figure 5.4: Results Breakout

Method	Pong		Breakout	
	AUC	Top	AUC	Top
Greedy	3783	0.62	37038	63.0
Annealed	4140	3.14	35678	71.2
Bootstrap	8222	18.4	44545	86.9
Noisy	3436	1.4	38978	67.7
Random		-20.7		1.7

Table 5.4: Results Atari

5.4 Framework

In table 5.5 we provided the final framework. This framework can be used by companies. It gives an overview of the difficulty of implementation, the speed, AUC, top score and what type of environment is suitable. The framework provides this information for every algorithm. Companies can check their problem based on their restrictions

and what they think is important. For example when there is no limitation on computational time a bootstrap algorithm can be the best option as it will converge faster. When a problem doesn't involve a lot of randomnesses and has to be solved in a shorter amount of time the annealed algorithm would be selected.

	Imp.	Speed	AUC	Top score	Env. type
Greedy	++	+	−	−	No exploration needed
Annealed	++	+	+	+	Explore over time
Bootstrap	− −	−	++	++	Keep exploring
Noisy	−	+	+	+	Little exploration

Table 5.5: Framework: How to choose which algorithm

In conclusion, when looking at the performance the bootstrapped DQN is doing the best and also learns faster. The bootstrapped DQN works better with more complex problems. Despite the fact of being the best performer it also has some downsides. As it is harder to implement and is slower. When these downsides are big enough that it is not a good fit for the problem, there should be looked at the different algorithms. As shown in the framework these differences are limited and mostly depend on the type of environment and the amount of exploration needed.

6 Discussion

More research has to be done to make stronger claims about the different algorithms and their performance. Also due to time limitations and cost restrictions we could not have enough timesteps to converge to an optimal solution for any of the different algorithms. This is also shown in the different papers which describe the algorithms used. Despite our limited simulation times, we can now see how the algorithms behave with limited data. This is also very useful especially for businesses as they can have an idea which algorithm to use with limited data.

Also we found out that results differed a bit from the original papers. This is because we did not use the exact same networks and parameters for all experiments. We see this for the performance of the chain environment with the bootstrapped DQN. The noisy networks do not perform in the Mountain Car environment. This suggests that the parameters of the noise that was added were not suitable for this problem. To compare the methods in a more equal way we can first optimise the individual algorithms. After the right networks are chosen and the corresponding parameters the final results can be generated and compared. The optimisation method can also be very helpful for a company which needs to implement the algorithms to boost their performance.

In this research, every simulation is done exactly 3 times with different seeds which leads to different results. It is done multiple times as all algorithms and some environments have random factors in them. To prove the significance the experiments have to be performed multiple times. A Wilcoxon signed-rank test [32] can be performed on the different metrics to conclude significance. To make bigger claims the number of different seeds has to be increased as it will help with this test. In this research, this could not be performed due to time limitations.

Next to improvements in the experimental setup, we can also look at different algorithms which are competitive with the current ones. One recent publication is "Rainbow: Combining Improvements in

Deep Reinforcement Learning" [9]. This DQN is a combination of a set of state of the art reinforcement algorithms. It can be interesting to see if other algorithms are performing better in the different environments. In the paper they outperformed almost all know algorithms at that point in time. This can be seen in figure 6.1

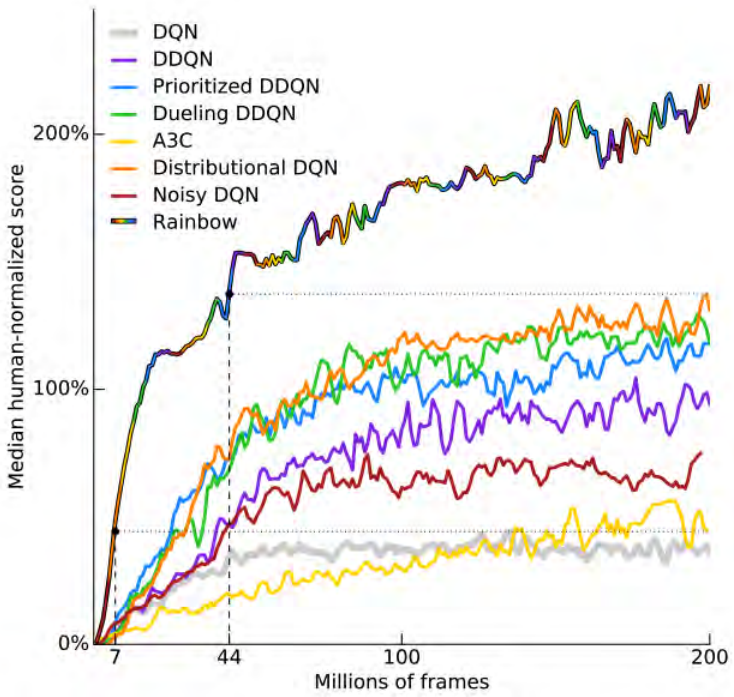


Figure 6.1: Results rainbow [9]

As shown in figure 6.1, there are a lot of different algorithms that are compared. If this research will be extended this will be one of the first to take a look at. As the time is progressing a lot of different algorithms are developed with improvements in performance. Our framework will also check how suitable these new algorithms will be in a business case.

Bibliography

[1] Stella: A multi-platform atari 2600 vcs emulator.

[2] RS Sutton AG Barto and CW Anderson. Neuronlike adaptive elements that can solve difficult learning control problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

[3] Arthur Juliani. Simple reinforcement learning with tensorflow part 4: Deep q-networks and beyond, 2016. [Online; accessed July 24, 2018].

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[5] Ronald Ortner Christos Dimitrakakis. *Decision Making Under Uncertainty and Reinforcement Learning*. 2018.

[6] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017.

[7] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[8] Matthew Hausknecht and Peter Stone. The impact of determinism on learning atari 2600 games. *AAAI*, 2015.

[9] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[10] Andrew Liao. Noisy net linear network layer using factorised gaussian noise. `https://github.com/andrewliao11/NoisyNet-DQN`.

[11] Tambet Matiisen. Demystifying deep reinforcement learning computational neuroscience lab. `http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/`, 2015.

[12] J Veness MG Bellemare, Y Naddaf and M Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[15] A Moore. Efficient memory-based learning for robot control. *PhD thesis, University of Cambridge*, 1990.

[16] Open AI. Leaderboard gym, 2018. [Online; accessed July 24, 2018].

[17] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. *CoRR*, abs/1602.04621, 2016.

[18] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[19] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray

Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016.

[20] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[21] Steeve Huang. Introduction to various reinforcement learning algorithms. part i (q-learning, sarsa, dqn, ddpg), 2018. [Online; accessed July 24, 2018].

[22] Steeve Huang. Introduction to various reinforcement learning algorithms. part i (q-learning, sarsa, dqn, ddpg), 2018. [Online; accessed July 24, 2018].

[23] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. January 1, 2018, 2014, 2015, 2016, 2017, 2018.

[24] Jakub Sygnowski and Henryk Michalewski. Learning from the memory of atari 2600. *CoRR*, abs/1605.01335, 2016.

[25] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In Michael Mozer, Paul Smolensky, David Touretzky, Jeffrey Elman, and Andreas Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, pages 255–263. Lawrence Erlbaum, 1993.

[26] Sebastian B Thrun. Efficient exploration in reinforcement learning. 1992.

[27] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

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

[29] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[30] Christopher Watkins. Learning from delayed rewards. 01 1989.

[31] M. Wiering. *Explorations in efficient reinforcement learning.* 1999.

[32] RF Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007.