Prioritized Experience Replay based on the Wasserstein Metric in Deep Reinforcement Learning

The regularizing effect of modelling return distributions

T. Greevink

Master of Science Thesis

TU Delft
Delft University of Technology

Prioritized Experience Replay based on the Wasserstein Metric in Deep Reinforcement Learning

The regularizing effect of modelling return distributions

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

T. Greevink

April 5, 2019

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Copyright ©
All rights reserved.

Abstract

This thesis tests the hypothesis that distributional deep Reinforcement Learning (RL) algorithms get an increased performance over expectation based deep RL because of the regularizing effect of fitting a more complex model. This hypothesis was tested by comparing two variations of the distributional QR-DQN algorithm [1] combined with Prioritized Experience Replay (PER) [2]. The first variation, called QR-W, prioritizes learning the return distributions. The second one, QR-TD, prioritizes learning the Q-Values. These algorithms were be tested with a range of network architectures. From too large architectures which are prone to overfitting, to smaller ones prone to underfitting. To verify the findings the experiment was done in two environments. As hypothesised, QR-DQN with Wasserstein metric based PER (QR-W) performed better on the networks prone to overfitting, and QR-DQN with TD based PER (QR-TD) performed better on networks prone to underfitting. This suggests that fitting distributions has a regularizing effect, which at least partially explains the performance of distributional algorithms. To compare QR-TD and QR-W to conventional benchmarks from literature they were tested in the Enduro environment from the Arcade Learning Environment (ALE). QR-W outperformed the state-of-the-art algorithms IQN and Rainbow [3, 4] in a quarter of the training time.

Table of Contents

Acknowledgements ix

1 Introduction 1

2 Preliminaries 5
- 2-1 Reinforcement Learning . . . 5
- 2-2 Deep Q-Network . . . 7
- 2-3 Prioritized Experience Replay . . . 10
- 2-4 Return Distributions with QR-DQN . . . 12
- 2-5 Discussion . . . 14

3 Methods 15

4 Experiments 19
- 4-1 Implementation . . . 19
- 4-2 Cart-pole . . . 20
- 4-3 Lunar lander . . . 20
- 4-4 Enduro . . . 22

5 Results and discussion 23
- 5-1 Cart-Pole . . . 23
- 5-2 Lunar lander . . . 26
- 5-3 Enduro . . . 32

6 Conclusion & Discussion 35

Bibliography 37

Glossary 41
- List of Acronyms . . . 41
- List of Symbols . . . 41

List of Figures

1-1 A depiction of two different distribution with the same mean 2

2-1 Fully connected neural network [5] 8
2-2 Filter in convolutional layer [6] 9
2-3 Example of Convolutional Neural Network (CNN) [7] 9
2-4 Convex asymmetric loss function for quantile $\tau \in [0, 1]$ [8] 13
2-5 Quantile Huber loss 13
2-6 Cumulative Density Function (CDF) of probability distribution $Z \in \mathcal{Z}$ and sample distribution $\Pi_{W_1} Z \in \mathcal{Z}_Q$, 1-Wasserstein error is shaded region. [1] 14

3-1 A depiction of two different distribution with the same mean 16

4-1 Cart-pole environment [9] 21
4-2 Render of lunar lander environment [10] 21
4-3 Enduro environment in the morning, noon, evening and night 22

5-1 Boxplot of max scores for 50 cart-pole simulations 24
5-2 Mean and median score vs steps for 50 cart-pole simulations 25
5-3 Boxplot of max scores for 10 lunar lander simulations 27
5-4 Mean and median score vs steps for 10 lunar lander simulations 27
5-5 Mean score vs steps and boxplot for 192-192 network, ... 29
5-6 Mean and median score vs steps for 10 lunar lander simulations with 192-192 network and $\alpha = 0.4$ 30
5-7 Boxplot of 300 evaluation episodes of Enduro 33

List of Tables

3-1 An overview of different algorithms and their methods 18

4-1 Deviating hyper-parameters 20
4-2 Structure of the CNN for the Enduro agent 22

5-1 Mean and median of max scores for 50 cart-pole simulations 24
5-2 Mean and median max scores of cart-pole vs networks 26
5-3 Mean and median of max scores for 10 lunar lander simulations 27
5-4 Mean and median max scores of 10 lunar lander simulations versus network sizes 28
5-5 Mean and median max scores of 50 lunar lander simulations for 192-192 network 29
5-6 Mean and median max scores of 10 lunar lander simulations versus alphas 29
5-7 Mean and median of max scores for 10 lunar lander simulations with 192-192 network and varying α 30
5-8 Mean and median of max scores for 10 lunar lander simulations with decaying alpha 31
5-9 Mean and median score of 300 evaluation episodes of Enduro 32
5-10 Mean Enduro scores found in literature. Reference values from [11, 4, 3]. 33

Acknowledgements

I would like to thank my supervisor ir. Tim de Bruin for guiding me through this thesis process, and all the hours of back-and-forth we have had. Thank you for the inspiration you provided, and the direction you gave me in the times I needed it. Furthermore I would like to thank dr.ing. Jens Kober for organising and overseeing this thesis project, and all the feedback you have given me throughout. Lastly, in advance, I thank prof.dr.ir. Hans Hellendoorn and the entire thesis committee for the time they took to read and judge my thesis.

Delft, University of Technology T. Greevink
April 5, 2019

“One day I will find the right words, and they will be simple.”
— *Jack Kerouac*

Chapter 1

Introduction

Systems and control engineers usually focus on model based control. However, as you might have guessed from the title, this thesis will focus on a different research area in control. Model based control works very well on a subset of problems. Namely, environments which can be approximated with mathematical models, especially linear models. This encompasses a lot of engineering problems but it has its limits. Even discrete environments which operate with simple rules can be very hard to capture in a mathematical model. As technology keeps getting closer to our daily lives there is an increasing demand for control in environments which are hard to model. In recent years however, there have been significant breakthroughs in the field of model free control through machine learning. Through the use of deep neural networks, algorithms are able to better generalize their discovered knowledge to unseen data. This has led to great results in many deep learning applications such as computer vision [12], speech recognition [13], natural language processing [14] and many others. One of the subfields which has gotten a lot of attention is deep Reinforcement Learning (RL). Deep RL can learn optimal control policies, model free, in complex environments with high dimensional observations. Since there is no preset policy, the algorithm can solve problems in ways the maker never thought of. As of the writing of this thesis however, the algorithm is still very sample inefficient. This means that, in practice, it only works well in simulated environments, with a lot of computational power available. Development of deep RL is going rapidly though, major accomplishments in the past years include beating the best humans and computers in: a host of Atari games [15], Go [16], Chess [17], Shogi (Japanese chess) [17], Dota 2 [18] and recently StarCraft II [19]. These are all environments where this level of performance by computers was unthinkable 5 years ago.

In deep RL an agent receives a feedback signal from its environment called the reward. The agent trains a neural network to estimate how much total reward it will get from choosing a state-action, valuating reward further in to the future less then immediate rewards. This evaluation is called the return. The agent chooses the action with the highest return estimate. This thesis will take a closer look at the domain of distributional deep RL. In particular the distributional algorithm proposed by Dabney et al. in [1], QR-DQN. Distributional RL algorithms achieve great performance by doing something seemingly counter-intuitive. Instead

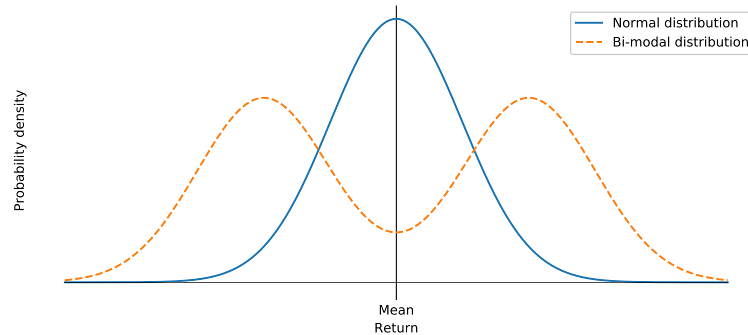

Figure 1-1: A depiction of two different distribution with the same mean

of estimating just the expected return for each state-action pair, they estimate an entire probability distribution for all possible returns for each state-action pair. The agent then takes the mean of each distribution and picks the action with the highest mean. But the mean of the distribution should be the same as the expected value. It seems pointless, and even unnecessarily difficult, to estimate an entire distribution of returns if we then take the mean anyway. This poses an interesting question, why does it work so well? This question does not have a decisive answer yet, but the authors of the paper say in multiple publications [1, 3] that the answer lies in the algorithm inherently trading of approximate solutions with likelihoods.

As was explained in the last paragraph our reinforcement learning agent will train a neural network to make an estimate of the return it will get from every state-action. We call the set of estimates for all state-actions the model. By tuning the parameters of the neural network the agent minimizes the error between the data it gathers from feedback of the environment and its estimates. This is called fitting the model to the data. By changing the prediction the model makes from the expected return to the return distribution the model complexity is increased. To better understand the difference between expectational and distributional models, let us take a look at the distributions shown in Figure 1-1. These two distributions are clearly not the same, but they do share the same mean. Because an expectational algorithm only tries to learn the expected value, it can not see the difference between these distributions. Therefore it cannot learn anything from this pair of distributions. A distributional algorithm learns to estimate the entire return distribution, so it will learn the difference. However, when the agent chooses an action it only compares the mean of each estimate distribution, so why can it still help? One explanation could be that solving a more difficult problem helps to regularize the model. Learning the best fitting distribution will automatically ensure that the correct expectation is learned, because this is just the mean of the distribution. But learning the distribution instead of the expectation is a harder model to fit to the data, because it contains more information. Learning a more complex model helps to prevent the model from overfitting on the data. Preventing overfitting will help the model to generalize its predictions better to unseen data. Another theory on why learning distributions could be important is that it helps to better differentiate between different types of environment mechanics with the same mean. For example, a state-action pair which gives a 50% chance on a return of 0 and a 50% chance on a return of 10 is probably driven by different environment mechanics then a state-action which gives a return of 5 100% of the time. Learning to differentiate between

these two can give the agent a deeper understanding of the environment mechanics. If the agents policy then changes and the first state-action pair now gives a 60% chance on a return of 10 the estimate of this environment mechanism can be changed without it bleeding over on to the evaluation of the other environment mechanism. If any of these theories are correct then learning the difference between the distributions shown in Figure 1-1 is important, even though the agent does not use the information in choosing an action.

The notion that an agent can learn more from some transitions then from others is already a well established theory in deep RL. As one would expect, the agent learns less from transitions which it already predicts very well than from the ones it predicts poorly. Schaul et al. [2] proposed an algorithm that prioritizes the replay of transitions which are predicted poorly. By prioritizing these transitions the agent was able to both learn faster and have a better final score in almost all of the environments in the Arcade Learning Environment (ALE) [20]. In this algorithm the prioritization of the transitions is based on the Temporal Difference (TD)-error, which is the error between the estimated return and the target return calculated with the reward from the transition. But a distributional algorithm has more information available. With the distributions available we can calculate the error between two entire return distributions using the Wasserstein distance. For example, take a transition which has the distributions from Figure 1-1 as its return distributions; one for its estimate return distribution and the other for its target return distribution. If one would prioritize transitions based on the Wasserstein distance this transition would be prioritized, whereas aTD-error based prioritization would not.

By prioritizing replay of transitions on different measures a direct comparison can be made on how important learning the correct distribution is, versus learning the best fitting mean. Given the fact that the agent will only base its choices on the expected return, one would say that prioritizing replay based on a distributional metric such as the Wasserstein distance would be less optimal then prioritizing based on the TD-error. However, taking the ideas discussed in the previous paragraphs into account one might argue that fitting the entire distribution might be more important then finding the best fitting means. By comparing these two methods of prioritization, the differentiating factor between these two ideas is isolated. This provides interesting insight in the learning process of distributional algorithms.

In this thesis it is hypothesised that prioritising on the Wasserstein distance instead of on the mean will result in a better performance, because of the regularizing effect of fitting a distributional model. To test this hypothesis the TD-error and Wasserstein error based distributional Prioritized Experience Replay (PER) algorithms were trained on a set of different network sizes. If this hypothesis is true one would expect that the Wasserstein distance based algorithm performs better in large networks prone to overfitting, and worse in small networks prone to underfitting. The results of the experiments done in this thesis matched this expectation. In the networks large enough to allow overfitting the Wasserstein distance based agents performed better. When the network became so small that the model was underfitting, the TD-error based algorithm performed better. In all three environments which were tested the Wasserstein distance based algorithm outperformed the TD-error based algorithm. One of the environments tested was the Atari game Enduro, form the ALE. In this environment the Wasserstein distance based algorithm outperformed all algorithms compared in the state-of-the-art IQN [3] and rainbow papers [4].

The structure of this thesis is as follows: In Chapter 2 the preliminaries for this thesis are

given, discussing the essential deep RL, PER and Quantile Regression DQN (QR-DQN) knowledge for this thesis. Chapter 3 goes in to the methods, discussing the hypothesis of the thesis, what exactly is being tested, and why. Chapter 4 covers how the hypothesis was tested, the exact implementation of the algorithms, and the test environments. In Chapter 5 the results for these experiments are presented and interpreted. And Chapter 6 gives a concise overview of the contributions in this work and the drawn conclusions.

Chapter 2

Preliminaries

To understand the research in this thesis some preliminary knowledge in the field of deep Reinforcement Learning (RL) is necessary. This chapter will briefly cover this necessary background knowledge and provide citations to more elaborate descriptions. The four topics which will be covered are RL, Deep Q-Network (DQN), Prioritized Experience Replay (PER) and Quantile Regression DQN (QR-DQN). A good understanding of the last two topics is essential to the further understanding of this thesis, as these two algorithms will be used extensively.

2-1 Reinforcement Learning

Let us start with the basics, RL. RL is a type of machine learning which is neither completely supervised nor unsupervised learning. The main advantage of RL over supervised learning is that it does not need labeled data or examples to learn from. With RL the agent learns from data which is produced by the agent itself. The only feedback the agent gets is through a reward which is given by the environment. The agent then tries to find the action for each state which approximately maximizes the sum of all the rewards it gets throughout the whole episode. The set containing each action an agent chooses for each state is called the **policy** π of the agent. Because the algorithm learns from rewards instead of from another policy it can find new and unexpected ways to solve a problem. Resulting policies have the potential to be much more effective then anything the maker of the algorithm could have thought up. Also, because the algorithm learns from rewards instead of examples, it is a very general algorithm. A single algorithm has the potential to solve many totally different problems, and it can solve problems in ways the maker of the algorithm might not have been able to think up.

However, the flip side of this freedom in policies is that for many problems the algorithm still has a hard time to converge to an effective policy at all. Shaping the reward structure to guide the algorithms to a specific policy can help but it gives up part of the advantages RL had in the first place.

2-1-1 Bellman update

First we will take a look at how the most basic version of the RL algorithm learns and introduce a few important terms. A more extensive overview of these concepts can be found in Sutton & Barto [21]. The **agent** is the entity which learns and acts out the policy. Everything the agent can detect and/or interact with is called the **environment**. All the information which describes the environment at a certain point in time is called the **state** ($s_t \in \mathbb{R}^n$). Every time step the agent interacts with the environment by picking an **action** ($a_t \in \mathbb{R}^m$). The agent then gets the new state, and a **reward** ($R_t \in \mathbb{R}$). This quadruplet of state, action, reward and new state is called an **experience**, and will often be referred to as a transition.

The set of experiences an agent makes from its start state until its terminal state is called an episode. The **Return** (G_t) is a discounted sum of all future rewards, where rewards further into the future are discounted more. In this thesis the return is defined by

$$G_t = \sum_{k=1}^{\infty} \gamma^{k-1} \cdot R_{t+k}, \tag{2-1}$$

where $\gamma \in [0, 1)$ is a discount factor. The return is very important as this is the 'approximate sum of all the rewards' the agent is trying to maximize. Because it is maximizing this total reward it will plan ahead and is able to pick actions with a delayed reward. The discount sum γ determines how far the agent will look into the future. At $\gamma = 0$ the return would be the same as the reward, at $\gamma = 1$ the return becomes the total sum of rewards in the episode. The expected return for a certain state is called the **Value** ($V^\pi(s) \in \mathbb{R}^n$) of that state and is dependent on the policy of the agent. Returns for state-action pairs are referred to as the **Q-Value** ($Q^\pi(s, a) \in \mathbb{R}^{n \cdot m}$) of that state-action, which is also dependent on the policy. If the state or action space is continuous, or there are simply to many states to store a function approximator has to be used to determine the Q-Value. In deep RL this function approximator will be a Deep Neural Network (DNN), which will be discussed further in Section 2-2.

If the correct Q-Values are available picking the policy is trivial; the agent simply picks the action with the highest Q-Value all the time. So if the agent can estimate these Q-Values we automatically get our policy. The Bellman equation offers a solution. By breaking up the Q-Value up into the reward of the next step, plus the discounted reward of all steps after that. The 'rewards of all the steps after that' is estimated by the Q-Value of the next state, so we get the elegant equation

$$Q(s_t, a_t) = \mathbb{E}[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})]. \tag{2-2}$$

Most likely the agent will have multiple Q-Values to choose from in state s_{t+1}. Most of the time the agent will have a greedy policy picking the action cooresponding to the highest Q-Value, but for reasons which will be discussed later this is not always the case. An agent could also be learning from the state transitions of a different agent. In these cases we have a choice to make, pick the Q-Value the agent took, or pick the Q-Value the agent thinks is highest at the moment of learning. In this thesis we will always take the Q-Values which the agent thinks is highest at the moment of learning.

Because the agent's policy will probably change as it is learning it is wise to keep a running mean to estimate the expectation. To keep this running mean first the error is calculated

between the current estimate of $Q(s_t, a_t)$ and the new target estimate $R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$. This error is called the **TD-error**. Then for the update TD-error multiplied by a fraction α is subtracted from the current estimate. The resulting update equation to estimate this expectation is

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)). \tag{2-3}$$

where $\alpha \in (0, 1)$ is the learning rate.

2-1-2 Exploration

In the previous section it was briefly mentioned that the agent might not always want to pick the highest Q-Value. For example, take a simple situation where an agent starts in a state with two actions. When it picks either one the agent gets a reward and the episode terminates. For the 1$^{\text{st}}$ action the agent always gets a reward of 1. For the 2$^{\text{nd}}$ action has a 25% chance of getting a reward of 8 and a 75% chance of getting a reward of 0. Now let us say the agent gets a reward of 0 the first time it chooses the second action. It now thinks the first action has a higher Q-Value, namely 1, and it will continue to pick this action. To mitigate this issue we will need some type of exploration to see whether the estimates of the Q-Values which are not chosen, are still right.

A simple but effective way of ensuring exploration is to have a chance of picking a random action every step. Usually this chance starts out at a 100% when the policy is still very unreliable and decreases as the agent has learned more. The downside of this type of exploration is that it is very inefficient to explore areas of the state space far away from the current policy. With an infinite number of steps the agent is assured to explore the whole state-space this way. However, since this is impossible to achieve the agent might end up getting stuck in a local optimum.

2-2 Deep Q-Network

The DQN algorithm [15] is the most influential RL paper in recent years. It has sparked a wave of research in the subfield, deep RL. Deep RL is mostly the same algorithm as normal RL, which was discussed in Section 2-1. The only difference is that a DNN, which will be discussed in Section 2-2-1, is used as a function approximator for the Q-Values. Also a few adjustments were made to make the algorithm more stable, which will be discussed in Section 2-2-2.

The advantage of using a DNN as a function approximator for the Q-values is that DNNs are good at generalizing over the state space. This means that policies which are successfully learned in one part of the state space, can also be applied in other parts which are similar. Then not all of the state space has to be explored and learned to find a policy which performs well everywhere. This opens up the possibility to learn much bigger state spaces, and therefore much more interesting problems.

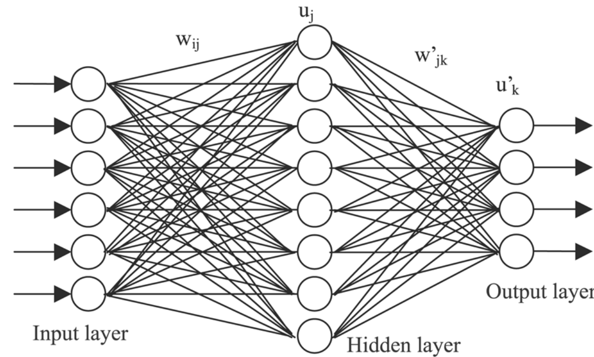

Figure 2-1: Fully connected neural network [5]

2-2-1 Deep Neural Networks

DNNs are a biologically inspired technology, loosely based on the nerve networks found in humans and animals. In 1943 Warren McCulloch and Walter Pitts [22] created the first computational model for a neural network. In 1958 Frank Rosenblatt, published the perceptron [23], an algorithm for pattern recognition. However, the use of DNNs was only popularized in the 2010's due to the heavy computational demands to train a DNN. A DNN is essentially a non-linear input-output mapping. The basic version of a DNN is a fully connected network. The DNN consists of a series of layers containing non-linear functions. These non-linear functions are called nodes. The output of each node is weighted, summed up with the other nodes and then fed into the input of one of the nodes in the next layer. This is done for every function in the next layer. Figure 2-1 gives a schematic overview of what this might look like, where w_{ij} is the weight for node i in the first layer to node j in the second layer. u_j is the output of node j, and so on for w_{jk} and u_k. As can be seen there are three types of layers in the network; an input layer, a hidden layer and an output layer. Each of these layers is a set of nonlinear functions, represented by the nodes in Figure 2-1. Common functions used are for example sigmoidal functions, hyperbolic tangents (Tanh) or rectified linear units (ReLU). The output of the first layer is then used as an input of the second layer, and so forth. This is represented by the lines between the nodes in Figure 2-1. Each of these lines has a weight by which the new input is multiplied, and optionally a bias which is added.

The objects to be classified are given at the input layer and the network is calculated through to the output layer to give a prediction. To train the network labeled or target data is used. The prediction is compared to the target which gives an error for that data. Then from the output layer to the input layer the weights are adjusted a little bit using stochastic gradient descent. This training of the DNN is called back propagation [24].

A variant of the DNN is the Convolutional Neural Network (CNN), proposed by Yann le Cun in [25], which is most commonly used in visual classification tasks. This adaptation makes better use of spacial information by preprocessing the input data with convolutional and pooling layers. A convolutional layer consists of a set of filters which are convoluted with an input image to form an output image. This means it scans over the input, multiplies it with a set of weights and outputs a linear combination of the input values. These weights are tuned with back propagation. This way the network will learn its own set of features. An example of such a filter is given in Figure 2-2. A convolutional layer has a specified number

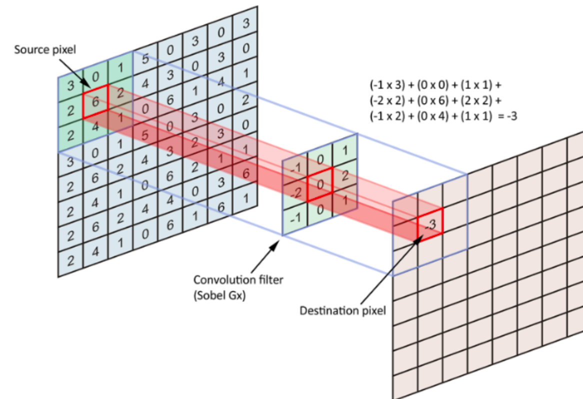

Figure 2-2: Filter in convolutional layer [6]

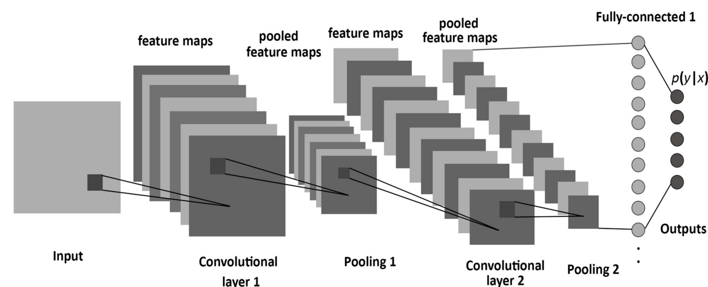

Figure 2-3: Example of CNN [7]

of filters, of a specified size which scan the input with a specified stride.

Then there typically follows a pooling layer to generalize the positions of features. However, in control settings this layer is usually skipped, as the exact position of features can be important information. Max pooling is the most common type of pooling layer, which selects the maximum value in a certain area. This reduces the size of the representation to reduce the number of parameters and amount of computation in the rest of the network. It also helps with generalization as it blurs out the exact location of features but retains a rough relative location to other features, which helps to control overfitting. Finally the CNN might look something like Figure 2-3.

2-2-2 Replay memory & Target network

As was mentioned in Section 2-2 a few key adjustments were proposed in [15] to get the Q-learning algorithm to work with a DNN. The first adjustment is that the agent learns from experience replay instead if directly from the state transitions. In experience replay the state transitions are stored in a replay memory. When the network is trained a batch of them is selected at random and with back propagation a stochastic gradient descent step is performed on the weights. By sampling batches at random like this there are no temporal correlations between the state transitions, which helps to stabilize the convergence of the network. This is because the neural network has to make a highly nonlinear mapping of the state space to the value functions. If it is trained for a certain part of the state space for too long all the

other parts will get messed up. By taking randomly sampled batches instead of single state transitions the accuracy of the gradient estimate is improved.

To further stabilize the convergence of the network a copy of the Q-Network is made every N steps which is used to compute the target for the stochastic Q-update. This is important because as we can see from equation 2-2 there is a feedback loop in the Q network from $Q(s_{t+1}, a_{t+1})$ used to estimate the target to $Q(s, a)$ which is updated. Feedback loops can get unstable, so to mitigate this problem the network used to estimate $Q(s_{t+1}, a_{t+1})$ is frozen, and only updated every N steps.

2-3 Prioritized Experience Replay

In the last section it was discussed that the state transitions are sampled from the replay memory with a uniform probability. This means that on average all transitions are revisited about the same number of times. However, some experiences might be very rare, whilst a lot can be learned from them. If these transitions are sampled just as often as all the other transitions the agent might not be able to learn everything it could from these transitions.

Schaul et al. [2] proposed an alternative to uniform sampling. Their PER algorithm attempts to sample according to how much we expect to learn from a transition. A measurement for how much we expect to learn from a transition could be its TD-error, given by

$$Q(s, a) - [R(s, a) + \gamma Q(s_{t+1}, a_{t+1})]. \tag{2-4}$$

Each transition gets a sample probability $P(i)$ which are used to sample a batch of transitions. To sample k transitions we take the range $[0, P_{total}]$ and divide it in k ranges. A value is uniformly sampled from each range and the transition i that corresponds to this value is retrieved. By changing the sample distribution a bias is introduced. This bias can be counteracted by importance sampling.

As was shown by Schaul et al. [2], applying these changes leads to faster learning and a better performance on many learning tasks. This concept of non uniform sampling is also used in this thesis. The prioritization of the transitions is based on either TD-error as in the work of Schaul et al., or the Wasserstein distance. By comparing these two we can see the influence putting the emphasis on learning the best fitting mean versus learning the correct distribution.

2-3-1 Priorities

To determine the desired priorities we are looking for a metric which estimates how much we can learn from a state transition. Unfortunately a direct measurement is not available, but the TD error from equation 2-4 should give a reasonable estimate proxy. This error represents how much the current network was wrong in predicting the outcome of a state transition. An additional advantage of the TD-error is that it is already calculated for each state transition every time it is used to update the network. However, as the network is updated this TD-error is immediately outdated. Because the network only changes a little at a time though, it should be reasonably accurate for a while.

Calculating the TD-error for every transition in the replay memory at every learning step is computationally unfeasible, so using the last known TD-error is a decent alternative. However, using the last known TD-error as priority has some shortcomings. First off, if the first TD-error is very small the transition might never be sampled again. Secondly the algorithm will play back stochastic transitions more often, because they will always seem 'surprising' with the TD-error. To mitigate these shortcomings as much as possible the sample probability is determined by

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{2-5}$$

where $p_i > 0$ is the priority of transition i. The parameter α will determine how much the prioritization is used, with $\alpha = 0$ being the uniform sampling case and $\alpha = 1$ being sampling proportional to the priority. The priority p_i can be determined in multiple ways but here the proportional variant will be discussed. In this case the priority is simply the last known TD-error plus a small positive constant which ensures the transition can still be sampled if the error is zero.

2-3-2 Importance Sampling

By changing the sampling rate from uniform to a prioritized distribution an unwanted effect is introduced. Namely, because the occurrence rate of certain events is synthetically amplified or decreased the expected values of the returns estimated by the stochastic updates become biased. This bias can be corrected by using weighted importance sampling. In weighted importance sampling the TD-error is weighted with

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{2-6}$$

where N is the batch size, and β is a parameter between 0 and 1 which determines the amount of importance sampling. At $\beta = 1$ the weights fully compensate for the non-uniform distribution. For stability reasons all weights are normalized so they only scale downwards.

What happens in weighted importance sampling is that as samples become more likely to be sampled the step which is taken in the direction of its gradient is scaled down. So with $\beta = 1$ a transition which would be sampled four times as often, would have one fourth the step size. This effectively cancels out the effect the prioritized sampling gave us. RL has a highly non-stationary learning process because both the policy and the state distribution constantly change. Therefore it is assumed that at the start of learning, when the policy is still very volatile, a small bias can be ignored for the sake of a prioritized distribution. As the algorithm converges the parameter β is scheduled to converge to 1 which eliminates the bias. The importance sampling does have two other unintended effects which can also play a role. First off an advantage could be that as the algorithms gets closer to a solution the steps get smaller. In highly non-linear environments smaller learning steps can have an advantage as only the first gradient is approximated. Smaller steps can work better in such an environment as non-linear functions can be approximated with linear functions on small increments. The second effect is less desirable. As some of the transitions step sizes shrink more then others, the size of the mini batch is artificially shrunk as well. A smaller batch size means that there is more variance in the gradient, which can have a negative effect on convergence.

2-4 Return Distributions with QR-DQN

In the deep RL algorithms described so far the DNN tries to estimate the expectation of the discounted return. The correct expectation of the return for a certain policy is all the information an agent needs to determine the optimal policy. However, it is very tricky to get a good estimation of the expectation. Bellemare, Dabney and Munos [26] put forth an algorithm which estimates the entire return distribution, instead of just the expectation. The agent then takes the mean of this distribution, which should be the same as estimating the expectation directly. Still, this algorithm reaches state-of-the-art performance on most of the games in the Arcade Learning Environment (ALE) proposed by Bellemare [20]. Bellemare et al. argue [26] that estimating distributions has a fundamental importance in the learning process of a reinforcement agent. They argue that in deep RL an underlying similarity in outcome is more important than exactly matching means.

Bellemare, Dabney and Munos opt for a distributional approach, instead of estimating the means by optimizing for the maximum likelihood, they minimize the Wasserstein distance between estimates of distributions. Bellemare et al. [26] prove that the Wasserstein metric cannot, in general, be minimized by taking the Wasserstein distance as a loss using stochastic gradient methods. The algorithm they used in that paper was based on a workaround, which approximated minimizing the Wasserstein distance. Later, in [1] this same group of researchers went on to create an algorithm which uses quantile regression [27] to stochastically estimate return distributions for the state-action pairs. With this new algorithm they close the theoretical gap left by the workaround used in the previous paper by proving the algorithm minimizes the Wasserstein metric. This algorithm is named QR-DQN.

2-4-1 Quantile Regression

To explain quantile regression let us start with what a quantile is. The most well known example of a quantile is the median. The value which divides a data set in two equal parts. Half of the samples in the set are above it, and half are below. With quantiles, this principle is generalized for any number of parts. E.g. with five parts the first quantile would have 20% of the samples below it and 80% above, and so on. The groups that the quantiles separate have names like halves, thirds, quarters and so on but are also often referred to as quantiles. Finally, the term quantile mid-point refers to the value exactly in between two quantiles.

Quantile regression is a method to stochastically approximate the quantile functions of distributions. For a quantile $\tau \in [0, 1]$, the quantile regression loss is an asymmetric convex loss function that penalizes overestimation errors with weight τ and underestimations with weight $1 - \tau$. This results in the loss function

$$\mathcal{L} = \sum_{j=1}^{N} \sum_{i=1}^{N} \rho_\tau(Z_j - \theta_i) = \sum_{j=1}^{N} \sum_{i=1}^{N} \rho_\tau(u_{ji}) \tag{2-7}$$

where Z_j is one of the target quantiles, and θ_i is one of the current estimate quantiles. The ρ_τ function is given by

$$\rho_\tau(u) = (\tau - \delta_{u<1})u. \tag{2-8}$$

and shown in Figure 2-4, where u is the error between the target and the estimate, and $\tau \in [0, 1]$ specifies the estimate quantile i.

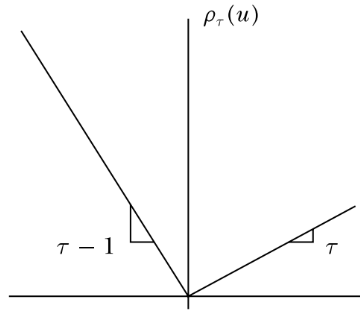

Figure 2-4: Convex asymmetric loss function for quantile $\tau \in [0, 1]$ [8]

2-4-2 Quantile Huber Loss

As can be seen in Figure 2-4 the loss function is not smooth at zero. Huber [28] argues that this can give problems when estimating parameters. This is because the gradient of the loss function never gets smaller, even as the loss function goes to zero. The estimate of the parameter will always keep overshooting its correct value. Therefore Dabney et al. also included a version of the QR-DQN algorithm with an adjusted asymmetric Huber loss on the interval $[-\kappa, \kappa]$, which they named the quantile Huber Loss. The Huber loss [28] is given by

$$\mathcal{L}_\kappa = \begin{cases} \frac{1}{2}u^2, & \text{if } |u| \leq \kappa \\ \kappa(|u| - \frac{1}{2}\kappa), & \text{otherwise} \end{cases} . \tag{2-9}$$

The quantile Huber is then simply the asymmetric veriant of the Huber loss,

$$\rho_\tau^\kappa(u) = (\tau - \delta_{u<1})\mathcal{L}_\kappa(u). \tag{2-10}$$

The resulting loss function is shown in Figure 2-5. In [1] the QR-DQN algorithm was tested for both $\kappa = 1$ and $\kappa = 0$, called QR-DQN-1 and QR-DQN-0 respectively. Both algorithms produced a similar mean and median performance. In the rest of this thesis the QR-DQN-0 algorithm will be used.

2-4-3 Minimizing the 1-Wasserstein distance

The p-Wasserstein metric W_p for $p \in [1, \infty]$ has been discovered several times from different perspectives in the 20$^{\text{th}}$ century. A more correct name would therefore be the Gini-Dall'Aglio-

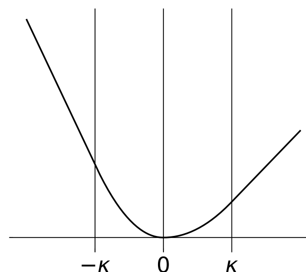

Figure 2-5: Quantile Huber loss

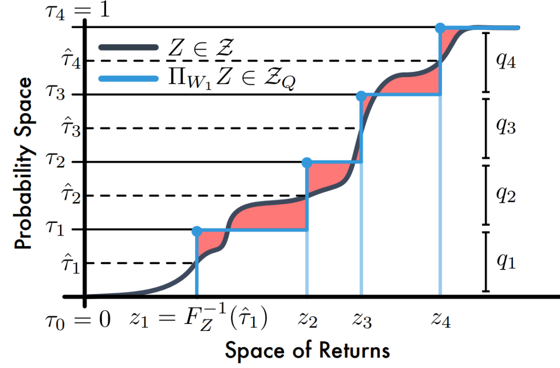

Figure 2-6: CDF of probability distribution $Z \in \mathcal{Z}$ and sample distribution $\Pi_{W_1} Z \in \mathcal{Z}_Q$, 1-Wasserstein error is shaded region. [1]

Kantorovich-Vasershtein-Mallows metric, but for the sake of simplicity it will be referred to as the p-Wasserstein metric W_p in this thesis. The p-Wasserstein metric $W_p(U, Y)$ between two distributions U and Y is given by

$$W_p(U, Y) = (\int_0^1 |F_Y^{-1}(\omega) - F_U^{-1}(\omega)|^p d\omega)^{1/p} \tag{2-11}$$

where F_Y^{-1} is the inverse Cumulative Density Function (CDF) [29] of distribution Y. To better understand the Wasserstein metric lets take a look at an example of two CDFs of a probability distribution $Z \in \mathcal{Z}$ and sample distribution $\Pi_{W_1} Z \in \mathcal{Z}_Q$ in Figure 2-6. The shaded areas represent the 1-Wasserstein distance between the two distributions. To minimize the 1-Wasserstein distance between the two distributions the supports of the sample distribution Z should be at the z values which minimize this area. At the vertical axis four quantiles are indicated with τ_1 to τ_4, each containing 25% of the distribution. In [1] Dabney et al. proved that for any distribution, the quantile mid-point $\hat{\tau}_i$ always corresponds to the value z_i which minimizes the shaded area. We then take these quantile mid-points $\hat{\tau}_i$ as the tau parameter in the quantile regression algorithm which gives us the set of supports which minimizes the 1-Wasserstein distance.

2-5 Discussion

In this chapter we covered the basics of RL, deep RL and two algorithms which build on these methods; PER and the return distribution based QR-DQN. Learning the return distributions with QR-DQN has a great performance, even though the distributions themselves are not used directly by the agent. This is slightly counter intuitive because one would think that if decisions are being made based solely on the mean return, that this is the only value we care about estimating. In this thesis the PER algorithm is utilised to see whether learning the approximate return distribution is actually more important then learning the best fitting mean return. This gives insight in the successful performance of distribution based algorithms like QR-DQN and also tests a novel way of prioritizing transitions in distribution based algorithms. The next chapter will cover the methods used to prioritize the transitions in the new algorithm, and how it differs from existing algorithms.

Chapter 3

Methods

Distributional Reinforcement Learning (RL) was not a new concept when the C51 algorithm was published by Bellemare et al. [26]. As they state themselves in their paper, a distributional perspective on RL goes back almost as far as the Bellman equation itself [30]. However in these instances the distributions had been used for specific purposes such as; to model parametric uncertainty, design risk-sensitive algorithms, or for theoretical analysis. The research on these topics dates back to before deep reinforcement learning was popularized by Mnih et al. [15]. It was not until the release of much more recent work that the interest of the deep learning community was drawn to distributional learning again. Arjovsky et al. published [31] a Generative Adversarial Network (GAN) which minimized the Wasserstein distance instead of the Kullback-Leibler divergence. A GAN is a different form of unsupervised deep learning, which can generate data which is superficially indistinguishable from the data which the GAN is trained on. Mostly it is used to generate images. This paper got a lot of traction as this different optimization metric significantly improved the stability of learning. How GANS work exactly is not in the scope of this thesis, but it is interesting to note that optimizing for the 1-Wasserstein distance is not just limited to deep RL.

At the same time Arjovsky et al. published Wasserstein GAN, Bellemare et al. published their first version of their distributional RL algorithm [26], C51. C51 was the state-of-the-art deep RL algorithm at the time of its release, with a generous margin. But there was no understanding of why the algorithm worked so much better. C51 did not even formally minimize the Wasserstein metric, but used a heuristic approach which approximated it. QR-DQN algorithms closed the theoretical gap left by the heuristic approach to minimizing the Wasserstein distance, and it took the place of C51 as state-of-the-art single-algorithm. However, it gave no further explanation as to why minimizing the Wasserstein distance over the maximum likelihood has such a positive effect on performance. Since the initial success of C51 there has been research [32, 33] on convergence guarantees of distributional deep RL. But why it performs better then expectation based RL is still unknown. As Bellemare et al. state in a publication [33] from Feb. 2019: *'Despite many algorithmic advances, our theoretical understanding of practical distributional reinforcement learning methods remains limited.'*.

So far, Lyle et al. found evidence [34] that the benefits of distributional learning lie in func-

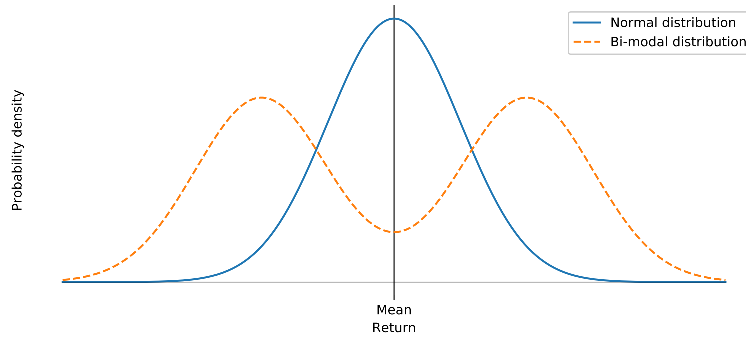

Figure 3-1: A depiction of two different distribution with the same mean

tion approximation. Bellemare et al. [26] discuss several possible explanations for the success of distributional deep RL. Two of these explanations were: state aliasing and a richer set of predictions. State aliasing was already briefly discussed in the introduction, but let us reiterate. Imagine two different states with the different return distributions given in Figure 3-1. Since the return distributions have the same mean, an expectation based algorithm will make the same prediction for these states. A distributional algorithm however makes different predictions. Bellermare et al. say that these aliasing predictions may result in effective stochasticity, as this prediction will not specify which of the two states is being observed. The second idea is that having a richer set of predictions can be beneficial for learning. This is a reoccurring research subject [35, 36, 37] in reinforcement learning. Bellemare et al. theorize that predicting the return distribution is an auxiliary task which is very closely related to the performance of the agent, which is why it is so successful.

In this thesis I theorize that these two explanations are actually the same phenomenon, which helps to regularize the model. Take two states with the distributions in Figure 3-1. For an expectation based model these states will have aliasing return estimates. Because the return estimates of some states will be aliasing the model will be easier to fit to the data, but later on the agent will not be able to predict the difference between these states. A model which is easier to fit to the data can be a beneficial in a situation prone to underfitting. This can occur when either the model does not get enough training time, or the network architecture is too small to represent a more complex model. However, the downside is that with enough training time and a sufficiently large network the simpler model will more easily overfit. This will hurt the agents ability to generalize its knowledge to unseen states. Having a more complex model results in there being less aliasing return estimates. Because the model makes a more complex prediction the return estimates will be harder to fit to the data. Because the model will be harder to fit to the data it will also be harder to overfit. In other words using a more complex prediction will help to regularize the model. This theory leads to the following hypothesis: 'Fitting distributions instead of expectations regularizes the model, which can lead to an increase in performance in situations prone to overfitting'.

The hypothesis makes a clear prediction; agents who fit distributions will have a better performance in situations prone to overfitting. However, to properly test this prediction is less straightforward. If we just compare a distributional and an expectation based model there are many factors which we are not controlling for. For example, the two algorithms will in-

trinsically have a different network architecture. Bellemare et al. [26] highlighted a significant instability in the bellman operator, which in combination with function approximation might prevent the policy from converging. Bellemare et al. believe that distributional algorithms are able to mitigate these effects by the averaging of the distributions to get the expected values. Furthermore, the two algorithms minimize a completely different loss function, of which any side effects are hard to determine. So by comparing these two algorithms it is not possible to attribute any results strictly to the fitting of distributions. In this thesis a different approach is taken. Instead of comparing a distributional algorithm to a expectation based algorithm directly, a distributional algorithm is combined with Prioritized Experience Replay (PER). This PER algorithm bases its priorities on different error metrics. Two versions of this combined algorithm are made. One version prioritizes based on the TD-error, the other one prioritizes based on the 1-Wasserstein error. The TD based version will only prioritize transitions based on the the estimated expectation. These transitions will be learned from more often which should give the model a preference for fitting these expectations. This algorithm will be referred to as QR-DQN with TD based PER (QR-TD) in this thesis. The priority for this algorithm is given by

$$\frac{|\sum_{i=1}^{N} Z_i - \theta_i|}{N} \propto \left| \sum_{i=1}^{N} Z_i - \theta_i \right| = p$$

with N quantiles in the estimate distribution θ_i and in the target distribution Z_i. Since the priorities p are only used proportionally we do not have to divide by N. Although technically this combined algorithm has not been tested in literature, this is just the straight forward combination of two algorithms which at their time of release both produced state-of-the-art results.

The Wasserstein based version will prioritize transitions based on the 1-Wasserstein error of the entire distribution. This will give the model a preference to fit the entire distribution instead of the expectation. This algorithm will be referred to as QR-DQN with Wasserstein metric based PER (QR-W) in this thesis. The priority for this algorithm is given by

$$p = \sum_{i=1}^{N} |Z_i - \theta_i|$$

with N quantiles in the estimate distribution θ_i and in the target distribution Z_i. Note that this is not exactly the same as the 1-Wasserstein distance between two sample distributions θ and Z. The difference is that for the formal 1-Wasserstein distance the samples should be sorted, to calculate the minimum distance. However, since our samples are not random but represent the quantiles, it does not make sense to sort them. To illustrate the difference between QR-TD and QR-W lets take another look at the distributions in Figure 3-1. In this case the QR-TD algorithm would give a priority of zero, and not replay the transition. The QR-W algorithm however, would give a non-zero priority and replay the transition to better fit the model to the data. The pseudo code for QR-W is given in algorithm 1.

To test the hypothesis QR-TD and QR-W will be compared using a range of network architectures. Some will allow for overfitting, others will force underfitting. The hypothesis predicts that in the overfitting scenarios the QR-W algorithm will perform better. It is also likely that if the network is not large enough to estimate a distributional model, and the model will

Algorithm 1 QR-W

Require: quantiles N, step size η, κ, γ, train freq. F, minibatch size k, α, β, steps T
  Initialize replay memory $\mathcal{H} = \emptyset$, $p_1 = 1$
  for t = 0 **to** T **do**
    Observe S_t and choose $A_t \sim \pi_\theta(S_t)$
    Observe R_t, S_{t+1}
    Store transition (S_t, A_t, R_t, S_{t+1}) in $\mathcal{H}$ with maximal priority $p_t = \max_i p_i$
    if t mod F = 0 **then**
      for j = 1 **to** k **do**
        Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
        Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
        Define distributional target for quantile i as $Z_i = R_t + \gamma\theta_{target/i}(S_{t+1}, A^*_{t+1})$
        Update priority with 1-Wasserstein error $p_j = \sum_{i=1}^N |Z_i - \theta_i(S_t, A_t)|$
        Compute quantile regression loss $\mathcal{L}_j = \sum_{i=1}^N |\sum_{k=1}^N \rho_{\hat{\tau}_i}^\kappa (Z_k - \theta_i(S_t, A_t))|$
      end for
      Calculate weighted loss $\mathcal{L} = \sum_{j=1}^k w_j \cdot \mathcal{L}_j$
      Update network weights with Adam Optimizer with loss $\mathcal{L}$
      Periodically update target network weights $\theta_{target} \leftarrow \theta$
    end if
  end for

Table 3-1: An overview of different algorithms and their methods

Return\Prioritization	Uniform	TD	Wasserstein
Expectation	DQN	PER	-
Distribution	QR-DQN	QR-TD	QR-W

be underfitting, the QR-TD algorithm will perform better because it is only focused on the TD-error. The environments which will be used for testing will be discussed in Chapter 4.

So far five deep RL algorithms have been discussed; Deep Q-Network (DQN), PER, Quantile Regression DQN (QR-DQN) (described in Sections 2-2, 2-3 and 2-4 respectively), QR-TD and QR-W. The difference between these algorithms lies in two factors; whether the return estimates are expectations or quantile distributions, and how the transitions are sampled from the replay memory. An overview of these five algorithms is given in Table 3-1.

Chapter 4

Experiments

The QR-DQN with Wasserstein metric based PER (QR-W) algorithm was tested in three different environments. In these environments the main metric of comparison is the maximum rewards achieved for every seed. The learning speed for these algorithms is also taken into account. The QR-W algorithm will be compared to Deep Q-Network (DQN), Prioritized Experience Replay (PER), Quantile Regression DQN (QR-DQN) (described in Sections 2-2, 2-3 and 2-4 respectively) and most interestingly QR-DQN with TD based PER (QR-TD). The only difference between the QR-TD and the QR-W algorithm is how the transitions are sampled form the replay memory. This comparison isolates the difference between how much is learned from trying to fit just the expected value versus the entire distribution.

The first environment is a cart-pole balancing task, a simple reinforcement learning problem with a four dimensional continuous state space and a discrete action space. All algorithms should be able to solve this problem most of the time. The second environment is a much tougher challenge called the lunar lander. The environment has an eight dimensional continuous state space with four discrete actions, which will need a much more complicated policy to solve it. Because this environment is a tougher challenge it is expected to show more separation between the performance of the algorithms. The third and last environment is by far the most complex. It is Enduro, a racing game released in 1983 designed for the Atari 2600 game console. This is one of the games from the Arcade Learning Environment (ALE) published in [20]. The observation of Atari 2600 games is an RGB image, which technically means the state space is discrete. However, the state space is so large that most states will never be visited. This is where deep Reinforcement Learning (RL) algorithms truly separate themselves from conventional RL algorithms, due to their ability to generalize their policy to unseen states. In this environment only QR-TD and QR-W will be tested, and compared to the published results for DQN, PER and QR-DQN [15, 2, 1].

4-1 Implementation

For the implementation of QR-W the same implementation as the QR-DQN-0 algorithm [1] was taken. Then the proportional prioritized sampling and the importance sampling weights

Parameter	Value
Exploration fraction	0.1
Final exploration ϵ	0.02
Learning starts	1000 steps
Target network update	1000 steps
Learning rate	10^{-3}
$\hat{\epsilon}$	10^{-8}

Table 4-1: Deviating hyper-parameters

from PER [2] were added in. The priority p_i of transition i as mentioned in [2] was based on the 1-Wasserstein distance between the estimate and the target, instead of the TD-error. The estimate and target distributions were not sorted before taking the 1-Wasserstein distance. Unless specified in Table 4-1 the hyper-parameters were kept the same as in the original papers. Except for the Enduro environment, where all parameters were kept the same as the original papers. The exploration fraction is the fraction of steps in which the algorithm goes from exploration $\epsilon = 1$ to its final exploration. The Adam optimizer [38] was used with parameter $\hat{\epsilon}$.

4-2 Cart-pole

The description of the cart-pole environment given by Open AI Gym is: 'A pole is attached by an unactuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time step that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.' [9]

This implementation from OpenAI Gym [39] corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson [40]. A visual representation of the environment is given in Figure 4-1. Cart-pole has a continuous state space with four states $\mathcal{S} \in \mathbb{R}^4$. These states are the carts position, the carts velocity, the poles angle and the poles angular velocity. The Deep Neural Network (DNN) used for the agent has four input nodes, a hidden layer of 128 nodes, a second hidden layer of 64 nodes and then the output layer of two nodes. The simulation is run for 40K steps, and the replay memory contains all the transitions.

4-3 Lunar lander

The second environment is also from the OpenAI Gym [39], called lunar lander. Lunar lander is an environment where the agents goal is to land a spacecraft on a landing pad without crashing. The state space is continuous and has eight dimensions $\mathcal{S} \in \mathbb{R}^8$. The states are the two Euclidean coordinates and velocities, the rotation and rotational velocity of the spacecraft and two booleans indicating for each of the legs if it touches the ground.

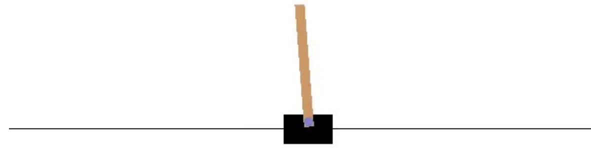

Figure 4-1: Cart-pole environment [9]

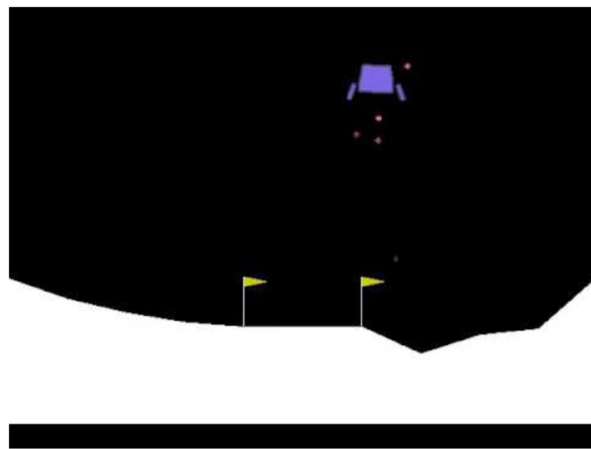

Figure 4-2: Render of lunar lander environment [10]

The ship has four discrete actions available: do nothing, fire left orientation engine, fire right orientation engine and fire main engine.

The landing pad is always at coordinates (0,0). The ship gets reward for moving closer to that landing pad, for having no velocity and for staying level. However, it loses reward for moving away, having a velocity and tilting. The reward for moving from the top of the screen to the landing pad and having zero speed is approximately 100 to 140 points, depending on the initial conditions. The spaceship also gets 10 reward for each leg which is on the ground. An episode finishes if the lander crashes or comes to rest at which point it receives an additional -100 if it crashed or left the screen and +100 points if it landed. Firing main engine costs -0.3 points each frame, and firing the orientation engines -0.03. The problem is considered solved at 200 points but higher scores are possible. Landing outside the landing pad is possible and fuel is infinite.

A visual representation of the environment is given in Figure 4-2. The DNN used for the agent has eight input nodes, a hidden layer of 128 nodes, a second hidden layer of 64 nodes and then the output layer of four nodes. The simulation is run for 250K steps, and the replay memory contains 100K transitions.

Table 4-2: Structure of the CNN for the Enduro agent

Layer	Input	Filter	Stride	Num Filters	Activation	Output
Conv1	84x84x4	8x8	4	32	ReLu	20x20x32
Conv2	20x20x32	4x4	2	64	ReLu	9x9x64
Conv3	9x9x64	3x3	1	64	ReLu	7x7x64
Flatten	7x7x64	-	-	-	-	3136
FullCon	3136	-	-	512	ReLu	512
FullCon	512	-	-	200x9	Linear	200x9

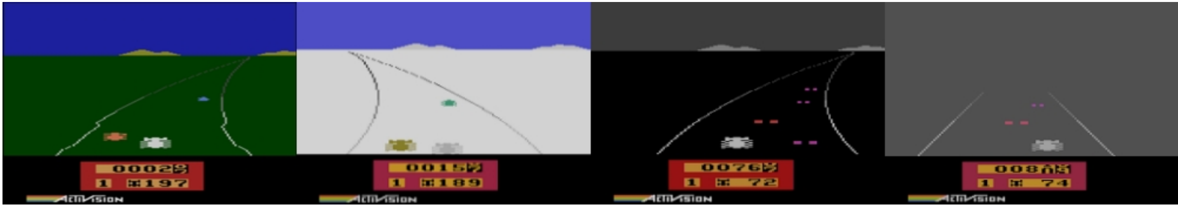

Figure 4-3: Enduro environment in the morning, noon, evening and night

4-4 Enduro

Enduro is a racing game made for the Atari 2600 in 1983. The goal of the game is to pass a set number of cars each day. As each day progresses the colour scheme of the environment changes to represent the time of day. Examples of how the environment might look are given in Figure 4-3. As can be seen during the evening and night only the brake lights of the cars are visible, and during night there is a reduced line of sight. In the bottom there is a red square with three counters, the upper value is the distance travelled (score), in the bottom left the current day is displayed, and in the bottom right the number of cars which need to be passed before that day is over. The first day 200 cars need to be passed and every day after that 300 cars. The agent has a discrete set of nine actions; do nothing, left, right, brake, accelerate, left+brake, left+accelerate, right+brake and right+accelerate. If the agent crashes into the car its speed is reduced to zero, often resulting in being overtaken by other cars.

The setup of the experiment is kept exactly the same as the set up used in the QR-DQN paper [1]. The architecture of the Convolutional Neural Network (CNN) is given in Table 4-2. The only difference is that in this environment the experiment is run for the first 50 million steps instead of the full 200 million steps. This is done to reduce the computational capacity needed to run the experiment. This leads to a small change in the algorithm, as the β needed for the PER algorithms no longer converges to 1, but to $\beta_0 + (1 - \beta_0) \cdot \frac{30}{200}$.

Chapter 5

Results and discussion

In this chapter the hypothesis formed in Chapter 3: 'Fitting distributions instead of expectations regularizes the model, which can lead to an increase in performance in situations prone to overfitting' is empirically tested. This is done in two environments, cart-pole and lunar lander, in Sections 5-1 and 5-2 respectively. For these tests the scores are a rolling mean over ten consecutive episodes during training. After this hypothesis is tested, the QR-TD and QR-W algorithms are compared to state-of-the-art algorithms on the Enduro environment from the Arcade Learning Environment (ALE). In this comparison the agents performance was periodically evaluated during training. A snapshot of the best evaluated agent was then run for 300 test episodes to determine the performance of the algorithm. This comparison can be found in Section 5-3

5-1 Cart-Pole

This section the results for the tests in the cart-pole environment are presented. First a comparison is made between QR-TD, QR-W and three algorithms from the literature they are based on. Then in the next subsection the hypothesis from Chapter 3 is tested by comparing QR-TD and QR-W for different network sizes.

5-1-1 Performance comparison

For the initial performance comparison all five algorithms (Deep Q-Network (DQN), Prioritized Experience Replay (PER), Quantile Regression DQN (QR-DQN), QR-DQN with TD based PER (QR-TD) and QR-DQN with Wasserstein metric based PER (QR-W)) are run 50 times for 40K steps. The mean and and median of the maximum scores are given in Table 5-1. Boxplots of the maximum scores for each run are given in Figure 5-1. As we would expect, the distributional algorithms perform best. Looking at the three distributional algorithms there is not very much separating them. However, QR-W does pull ahead a little bit and comes out as the best performer. Since the maximum score is 200 the top scoring algorithms

Table 5-1: Mean and median of max scores for 50 cart-pole simulations

	DQN	PER	QR-DQN	QR-TD	QR-W
Mean	186.47	193.68	196.55	196.30	**197.21**
Median	195.7	**200**	198.2	**200**	**200**

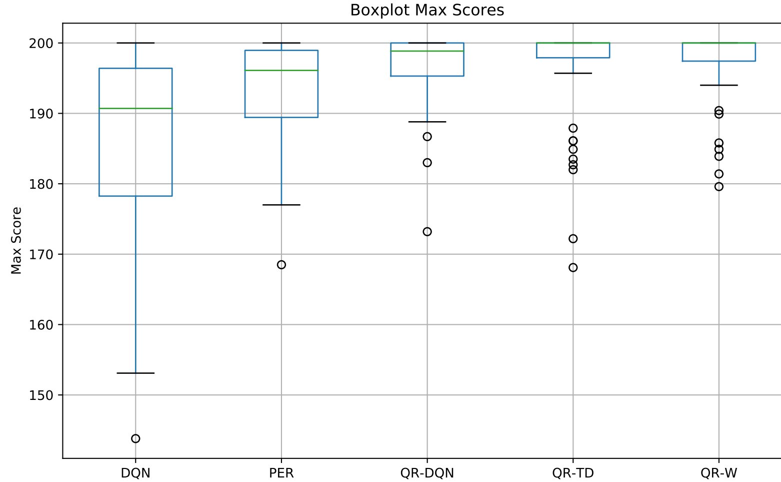

Figure 5-1: Boxplot of max scores for 50 cart-pole simulations

are all compressed near 200. The 0.66 points between QR-TD and QR-W might not look like much, but they represent a 19% error reduction.

To see whether a difference between two distributions is significant, an Analysis of Variance (ANOVA) test is often performed. However, looking at Figure 5-1 we can see that the scores are compressed at the top end, by the maximum score of 200. Because of this compression the scores can not be represented by normal distributions. Therefore an ANOVA test is not valid on these scores, since it assumes normally distributed groups. To better analyse these results the Mann-Whitney U test was performed. This test could be described as a non-parametric version of the ANOVA test, it does not assume a normal distribution. The Mann-Whitney U test gave $p = 0.23$. This means that there is a 23% chance that this result is by coincidence, which is not very convincing on its own. To draw conclusions the results will have to show consistent patterns over multiple environments and test parameters.

In the Methods chapter it was theorized that prioritizing to fit the distributions over the means would help to regularize the overfitting of the model. This is the reason we theorized that QR-W would perform better than QR-TD. We have found that QR-W performs better than QR-TD, now let us try to analyse whether this is due to the regularization like we theorized. To analyse this, tests were run for multiple smaller network sizes. Here we can look at how both QR-TD and QR-W perform with over/underfitting network architectures. This will give insight into the amount of regularization by both algorithms. The results of these tests are discussed in the next section.

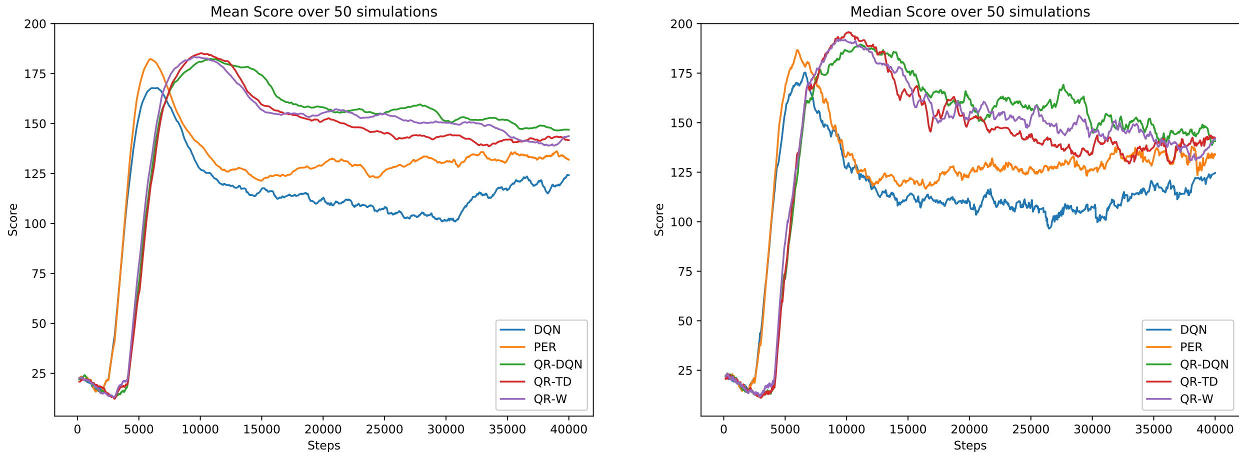

Figure 5-2: Mean and median score vs steps for 50 cart-pole simulations

5-1-2 Regularization with over and underfitting network architectures

In this section we try to analyse whether the increase in performance of QR-W can be attributed to better regularization in situations prone to overfitting. In Chapter 3 it was theorized that prioritizing to fit distributions instead of expected values will fit a more complex model, which helps to regularize in situations prone to overfitting. It was theorized that this regularization would be beneficial to the performance of the algorithm and therefore QR-W would outperform QR-TD. In the previous section we found that QR-W had an increased performance, now we will try to analyse whether this was due to better regularization. We will try to analyze this by seeing how both QR-TD and QR-W perform with over and underfitting network architectures. Therefore tests were run with a set of different network architectures. The tests were run for a network with 32-64, 16-32 and 8-16 nodes in the hidden layers, representing $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{8}$ of the networks hidden nodes of the initial performance test. Each network and algorithm was run for 10 simulations. After running the initial 10 simulations the best performing network was run an additional 10 times for verification. The mean and median of the max scores of all these runs are given in Table 5-2, where the best scores for each algorithm were highlighted.

Upon analyzing the results we can see that the model was doing some sort of overfitting in the initial performance test. As can be seen in Table 5-2 the scores increases as the network size decreases. The performance reaches a peak at a network structure of 16-32 nodes, after which it starts to underfit.

The results concur with our prediction; QR-W performs better then QR-TD with the larger networks. When the network becomes too small, and the performance declines the network is underfitting. With the underfitting model QR-TD performs better. These observations support the theory that fitting the entire distribution helps to regularize the model when it is overfitting. To better assert this hypothesis it is important to repeat this experiment in the lunar lander environment to establish that it is not an environment specific effect. The Mann-Whitney U test on the 16-32 network for 20 seeds gave $p = 0.20$ which, again, is not very convincing on its own. However, the consistent performance increase of QR-W with

Table 5-2: Mean and median max scores of cart-pole vs networks

	Mean		Median	
Net size	**QR-TD**	**QR-W**	**QR-TD**	**QR-W**
8-16	190.92	188.70	199.95	196.95
16-32	**198.23**	**199.53**	**200**	**200**
32-64	197.58	198.39	199	**200**
64-128	196.30	197.21	**200**	**200**

multiple network architectures does add to the credibility of a qualitative difference between these algorithms. It will be very interesting to see whether these results can be duplicated in a different environment.

5-2 Lunar lander

This section the results for the tests in the lunar lander environment are presented. Like in the previous section, a comparison is made between QR-TD, QR-W and three algorithms from literature they are based on. Then in the next subsection the hypothesis from Chapter 3 is tested by comparing QR-TD and QR-W for different network sizes. It was found that PER is not beneficial in this environment. Therefore, in Subsection 5-2-3 the influence of the alpha parameter from the PER algorithm (equation 2-5) is analyzed. Finally in Subsection 5-2-4, the PER part of the algorithms is adjusted to analyze the effect of importance sampling on the performance.

5-2-1 Performance comparison

Just like with cart-pole a performance comparison is done for all five algorithms. By analyzing these results we get a feel for how the algorithms perform in this environment, and what further tests might be interesting. The performance comparison run was done with a 128-64 network and the implementation described in Section 4-1. The results are given in Table 5-3 and Figures 5-3 and 5-4.

In Table 5-3 we see that QR-DQN performs best, better even than both QR-TD and QR-W. It is a common phenomenon in Deep Reinforcement Learning (RL) that an algorithms performance differs between environments. Although PER will have a positive effect on performance in most cases, in this specific environment QR-DQN performs better on its own. However, in this thesis the main focus lies on analysing the differences between QR-TD and QR-W. Tests on different network architectures will be done in Section 5-2-2 to see if the results found in Section 5-1-2 can be duplicated. Since QR-DQN has the best performance the α parameter will be varied in Section 5-2-3, to see whats influence it has in this environment. After this the prioritization algorithm in QR-TD and QR-W will be slightly adjusted in Section 5-2-4, to see whether the importance sampling is causing the performance to be lower than QR-DQN.

Table 5-3: Mean and median of max scores for 10 lunar lander simulations

	DQN	PER	QR-DQN	QR-TD	QR-W
Mean	54.59	121.99	**226.89**	202.81	216.32
Median	0.07	165.68	**228.53**	213.70	219.87

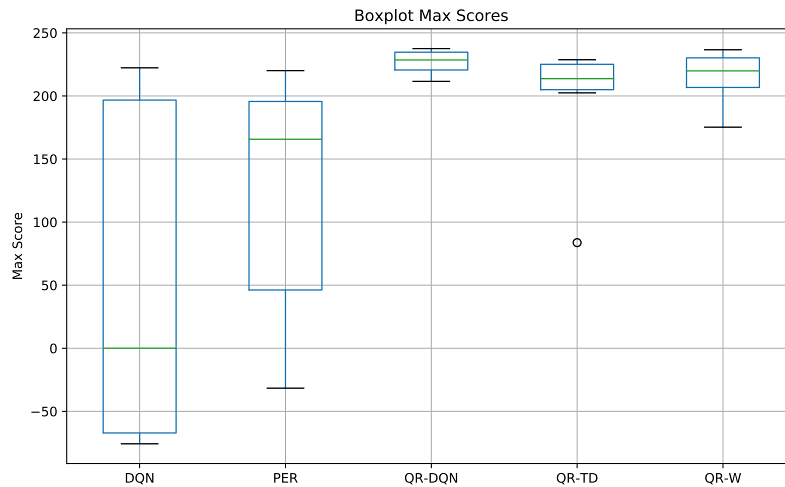

Figure 5-3: Boxplot of max scores for 10 lunar lander simulations

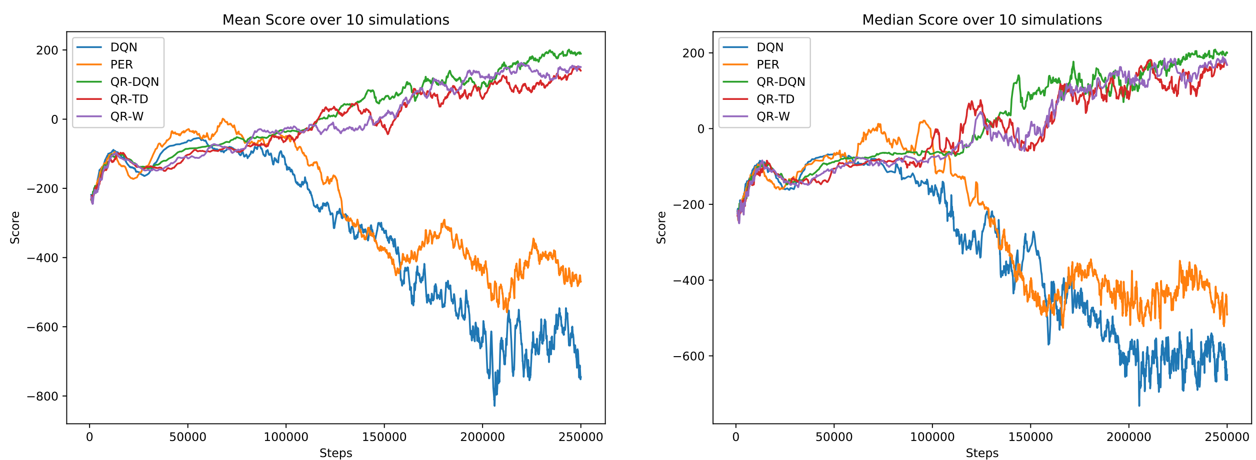

Figure 5-4: Mean and median score vs steps for 10 lunar lander simulations

Table 5-4: Mean and median max scores of 10 lunar lander simulations versus network sizes

	Mean		Median	
Net size	**QR-TD**	**QR-W**	**QR-TD**	**QR-W**
32-32	180.38	145.29	195.83	178.85
64-64	195.38	189.61	216.81	219.96
128-64	202.81	216.32	213.70	219.87
128-128	**221.83**	223.39	224.38	223.88
192-192	210.11	**226.71**	**228.80**	**226.11**
256-256	164.92	186.25	217.14	223.89

5-2-2 Regularization with over and underfitting network architectures

In Chapter 3 it was hypothesised that QR-W will perform better then QR-TD because of the regularizing effect of fitting distributions instead of expected values. In Section 5-1-2 this hypothesis was tested by varying the network size to see how QR-W and QR-TD performed in under and overfitting scenarios. If the hypothesis is true we would expect that with larger network sizes, where the model is prone to overfit, QR-W performs better then QR-TD. If the network size gets too small and the model is underfitting the regularization will yield no benefit. Therefore the prioritization of the best fitting means in QR-TD should give the best performance. The results found in Section 5-1-2 concurred with the hypothesis. To see if these results are environment specific or algorithm specific the experiment is replicated on the lunar lander environment. The lunar lander agent was tested with a 32-32, 64-64, 128-64, 128-128, 192-192 and 256-256 network. The results are shown in Table 5-4, where the best scores for each algorithm were highlighted.

Looking at the results, exactly the same pattern emerges as was the case with the cart-pole environment. The QR-W algorithm achieves a better performance then its QR-TD counterpart across a wide range of network architectures. Only when the agents are clearly underfitting the QR-TD starts to perform better then QR-W. This result supports the hypothesis that fitting distributions instead of expected values performs better because the more complex distributional model helps to regularize the training process if it is overfitting. To increase the validity of the measurement another 40 runs are done for the best performing network, 192-192, which are shown in Table 5-5.

As can be seen in Table 5-5 the mean scores of both algorithms are significantly lower then what we found in the initial tests. The median values did not suffer as much though. This can be explained by the skewed nature of the distribution seen in the boxplot in Figure 5-5. The scores can much more easily have outliers at the bottom end of the distribution then at the top end, since getting scores at the top end of the distribution is suppressed by the dynamics of the environment. Apart form this observation we see that with a bigger sample size the QR-W algorithm still performs better then QR-TD. Doing a Mann-Whitney U test on these results gives $p = 0.09$.

Table 5-5: Mean and median max scores of 50 lunar lander simulations for 192-192 network

	Mean		Median	
Net size	**QR-TD**	**QR-W**	**QR-TD**	**QR-W**
192-192	193.31	**203.91**	222.77	**225.44**

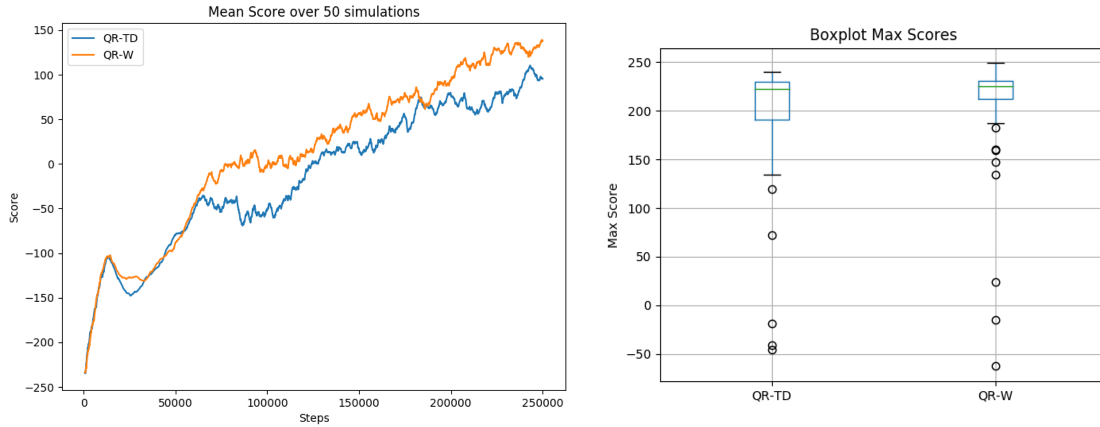

Figure 5-5: Mean score vs steps and boxplot for 192-192 network,

5-2-3 Prioritization alpha

The results from the initial test showed that uniformly sampled QR-DQN had better results then prioritized QR-DQN. However, the prioritized sampling algorithm, which was based on [2], lets one tune the amount of prioritization with a parameter α. In the original paper this parameter was set at 0.6, but as prioritized sampling does not seem to work very well it might be interesting to experiment with different values. All three prioritizing algorithms PER, QR-TD and QR-W were tested for α values 0.6, 0.4 and 0.2. In the results $\alpha = 0$ was also included, at which point PER reverts to DQN, and both QR-TD and QR-W revert to QR-DQN. The results of these experiments are given in Table 5-6. As can be seen the QR-TD algorithm improves as α is decreased. However, QR-W has its optimal performance at $\alpha = 0.4$, only just being beaten by uniform sampling. At this alpha the Mann-Whitney U test gives $p = 0.14$.

Doing a grid search for optimal parameters for the QR-W algorithm is too computationally

Table 5-6: Mean and median max scores of 10 lunar lander simulations versus alphas

	Mean			Median		
α	**PER**	**QR-TD**	**QR-W**	**PER**	**QR-TD**	**QR-W**
0.6	121.99	202.81	216.32	**165.68**	213.70	219.87
0.4	**141.77**	216.06	225.78	164.16	222.20	226.74
0.2	76.8	223.35	217.40	67.75	221.85	220.33
0	54.59	**226.89**	**226.89**	0.07	**228.53**	**228.53**

Table 5-7: Mean and median of max scores for 10 lunar lander simulations with 192-192 network and varying α

	$\alpha = 0$	$\alpha = 0.4$		$\alpha = 0.6$	
	QR-DQN	**QR-TD**	**QR-W**	**QR-TD**	**QR-W**
Mean	**236.91**	229.24	230.47	202.81	216.32
Median	**236.77**	232.91	232.28	213.70	219.87

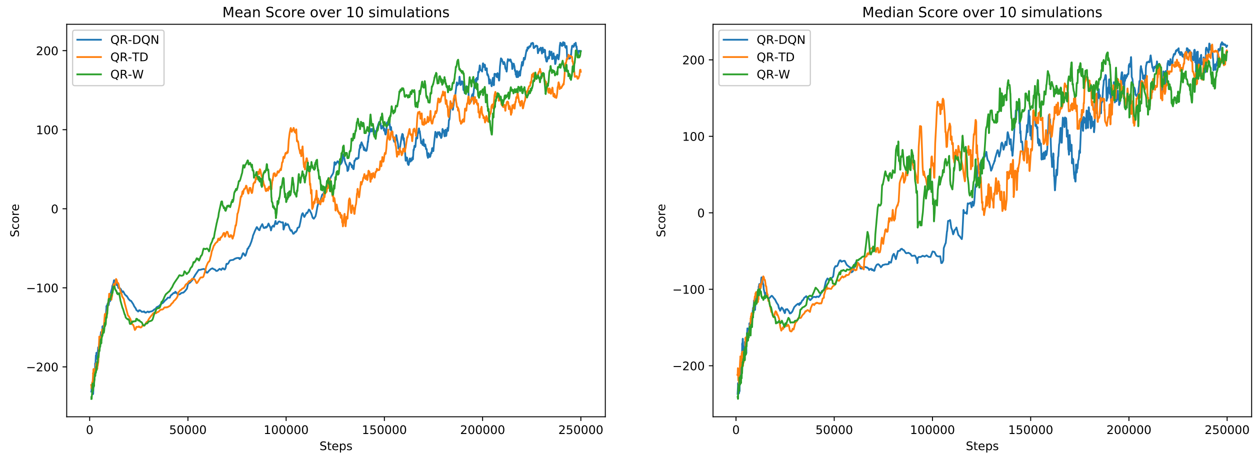

Figure 5-6: Mean and median score vs steps for 10 lunar lander simulations with 192-192 network and $\alpha = 0.4$

intensive. However combining the best performing network architecture with $\alpha = 0.4$, even though these two parameters are not independent, might still deliver something close to an optimal performance for the QR-W algorithm. The results of this test is shown in Table 5-7. This table also contains the earlier test results for $\alpha = 0.6$ and new results for the QR-DQN algorithm with the 192-192 network structure. As can be seen both QR-TD and QR-W have big jumps in performance, giving them an almost identical performance. This nearly identical performance means the Mann-Whitney U test over these 10 seeds gives $p = 0.43$ QR-DQN has an even bigger performance increase, leaving the other two algorithms behind it with a large margin. By analyzing the learning process shown in Figure 5-6 we can hypothesise why the QR-DQN algorithm does so well, and why the other two QR algorithms end up so similar.

The strength of PER lies in promoting sparse transitions so the algorithm can more quickly adjust its policy to it, instead of it being buried in the replay memory. With this in mind, lets look at the learning behaviour of QR-TD, QR-W and QR-DQN in Figure 5-6. In the learning process a clear pattern can be seen of the three stages of learning to fly the lander. Stage one is learning how to stay upright, and not immediately crash-land. This happens approximately simultaneous in all three algorithms from step 0 to 20K. After it has learned not to crash the next step is to learn to hover to the landing zone. This initially causes the total average score to drop because hovering takes actions such as firing the main engine, which cost points. As the agent learns to control the lander the score gradually increases. At a certain point all three algorithms make a large jump in score at which point the score

Table 5-8: Mean and median of max scores for 10 lunar lander simulations with decaying alpha

		Mean		Median	
α_0	β	**QR-TD**	**QR-W**	**QR-TD**	**QR-W**
0.4	**0.4**	222.8	**231.4**	228.5	**232.6**
0.6	**0.6**	199.0	**224.3**	228.7	**229.4**

becomes very volatile. This radical change in both the score and volatility of the score indicate that the agent has learned that for the best score it has to land in the landing zone, so it gets the bonus. In Figure 5-6 it can clearly be seen that both QR-TD and QR-W learn this behaviour significantly earlier then QR-DQN. This gives the PER algorithms a leg up for most of the training period. However, at the end of the learning trajectory at around 190K steps QR-TD and QR-W converge to very similar scores. This can be explained by the importance sampling starting to play a bigger role. The importance sampling removes the bias introduced by the PER. It also artificially reduces the minibatch size as the step size of some transitions is reduced. Reducing the minibatch size in turn leads to a greater variance in the gradient, which slows down convergence. At this point QR-DQN overtakes both QR-TD and QR-W in score. An interesting follow up experiment will be to see whether the QR-TD and QR-W algorithms perform better with a decaying α instead of importance sampling. This the results of this experiment are discussed in the next section.

5-2-4 Decaying alpha

In the previous section the influence of alpha on the algorithms performance was tested. From the results it seems that prioritization has a positive influence on performance at first, but no prioritization wins out in the end. We theorized that this could be the consequence of the importance sampling used in the algorithm. When the importance sampling parameter β anneals to one, surprising transitions which are sampled more often also get a smaller and smaller step size. This means that these transitions have a much smaller contribution to the gradient direction. A side effect of that is that the minibatch size is artificially been made smaller, since the gradient direction will be determined with fewer transitions. In this section the hypothesis that importance sampling is the reason QR-DQN performs better then QR-TD and QR-W in the lunar lander environment is tested. To test this a modified algorithm is used which decays α to zero instead of annealing β to one. First the algorithm was tried with $\beta = 0$, but it became clear very quickly that this only worked with an $\alpha_0 \leq 0.1$, at which point it is almost similar to the original algorithm. The algorithm was then tested with parameters $\alpha_0 = 0.4, \beta = 0.4$ and $\alpha_0 = 0.6, \beta = 0.6$. The results are shown in Table 5-8. The decaying alpha algorithm has a slightly better performance than the regular algorithm. However it still does not come close to the performance of QR-DQN. When comparing QR-TD and QR-W we see that QR-W still has the best performance. The Mann-Whitney U test for parameters $\alpha_0 = 0.4, \beta = 0.4$ gives $p = 0.15$. This, again, conforms to the pattern of not being a very significant difference on its own, but adding to the overall likelihood of a qualitative difference.

Table 5-9: Mean and median score of 300 evaluation episodes of Enduro

	QR-TD	QR-W
Mean	2305.50	**2546.17**
Median	2287.0	**2574.0**

5-3 Enduro

So far the results seem promising, and consistent over the two tested environments. These results beg the question whether this performance will hold up in the ALE [20]. In the ALE computational power starts to play a more prevalent role. Because of this there is only enough computational power available to perform one run for both QR-TD and QR-W. This means that the results of this test in isolation are statistically unreliable. However, in conjunction with the results found in the other environments, this test can still provide additional insight in the performance of this algorithm.

It is also worth noting that because there will only be one run per algorithm no parameter tuning was done. All the parameters were taken from the original papers [2, 1]. The runs were shortened to 50 million frames, instead of the conventional 200 million. To still be able to compare the results of this run to the existing literature the β parameter was still annealed to 1 as if there were 200 million frames. This recreates exactly similar test conditions as the first 50 million frames test done on the ALE in most literature. The Enduro environment was, in part, selected because it seems to have a very fast learning curve. In both the QR-DQN and the PER paper the algorithm got close to its final performance within the first 50 million frames. However, QR-TD and QR-W will still be at an significant disadvantage compared to a full 200 million frames test. This is because the algorithm is evaluated every million frames. A snapshot of the best performing evaluation is then further evaluated for the final result. So in a full 200 million frame test there are four times more evaluation points, of which the best performing one is chosen. Then we also have to take in to account that the agent will take a while to get close to its final performance.

Both QR-TD and QR-W were trained for 50 million steps, being evaluated every million steps. A snapshot of the best performing agent was then further evaluated for 300 episodes without exploration or learning. The mean and median performance of the two algorithms is given in Table 5-9. Also, Figure 5-7 gives a boxplot of the scores in these test episodes.

As can be seen in the results there is a significant gap in performance between the two algorithms. QR-W has a 10% increase in mean reward, and a 12.5% increase in median reward. Table 5-10 shows how QR-TD and QR-W stack up against the current state-of-the-art algorithms trained for 200 million frames. Notably, QR-DQN and IQN were run for three and five seeds respectively. The reported values are from one of those seeds, which seed was used is not specified in [3]. As can be seen QR-TD conforms with the scores found in the state-of-the-art literature. QR-W surpasses these scores with a significant margin, setting a new record on this particular environment. Since this is only one of the 57 games which is normally tested in the ALE, conclusions can not be drawn based on this data alone. However, since no parameter tuning was done whatsoever, it can safely be said that this is a promising result.

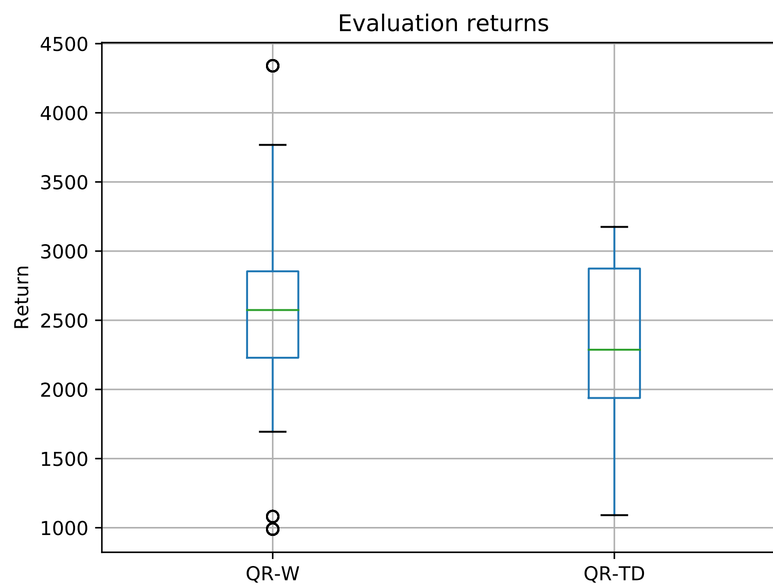

Figure 5-7: Boxplot of 300 evaluation episodes of Enduro

Table 5-10: Mean Enduro scores found in literature. Reference values from [11, 4, 3].

Human	DQN	PER	Rainbow	QR-DQN	IQN	QR-TD	QR-W
860.5	729.0	2093.0	2125.9	2355	2359	2305.50	**2546.17**

Chapter 6

Conclusion & Discussion

In this thesis it was hypothesised that distributional deep Reinforcement Learning (RL) gets performance improvements over expectation based deep RL from the regularizing effect of fitting a more complex model. This hypothesis was tested by comparing two variations of combining the Quantile Regression DQN (QR-DQN) (Section 2-4) and Prioritized Experience Replay (PER) (Section 2-3) algorithms. One variation, named QR-TD, prioritized based on the error of the expectation. The other one, named QR-W, based its prioritization on the Wasserstein distance between the return distributions. The only difference between these two algorithms is that QR-TD promotes learning a return distribution function for which the mean is correct, whereas QR-W promotes return distribution function for which the entire distribution is correct. Our hypothesis predicts that QR-W will perform better than QR-TD when the network is sufficiently or excessively large. It also predicts that when the network architecture becomes too small, and the model is forced to underfit, the QR-TD algorithm should perform better.

An experiment was carried out for these two algorithms with a range of different network sizes in two simulated environments; a cart-pole balancing task (Section 4-2), and the control of a lunar lander (Section 4-3). The results of these experiments concurred with the predictions for the hypothesis. These results provide new insight in to the mechanisms behind the performance increase of distributional RL algorithms.

Since the novel QR-DQN with Wasserstein metric based PER (QR-W) algorithm performed well it was run in the Enduro environment (Section 4-4) from the Arcade Learning Environment (ALE) [20] to benchmark its performance. In this test no parameter tuning was done, to ensure there was no tailoring to this specific environment. Also the test was run for 50M steps instead of 200M because of limited computational resources. Still, the QR-W algorithm outperformed all algorithms compared in the state-of-the-art IQN [3] and rainbow [4] papers.

The findings in this thesis provide fundamental insight in to the advantages of distributional algorithms over expectational ones. This can help to identify situations where distributional RL will give an increased performance over expectation based RL. Also by shedding light on the underlying mechanisms which drive performance, the creation of new algorithms might be inspired which make even better use of these mechanisms.

Further research on this topic could consist of additional tests in the ALE environment to verify the increased performance in more environments. Also, in this thesis it was shown that the regularizing effect is responsible for at least part of the performance increase of distributional RL. An additional interesting experiment would be to compare DQN and QR-DQN directly with different network sizes, to see whether the regularizing effect accounts for all of the performance increase.

Bibliography

[1] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos, "Distributional reinforcement learning with quantile regression," *arXiv preprint arXiv:1710.10044*, 2017.

[2] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[3] W. Dabney, G. Ostrovski, D. Silver, and R. Munos, "Implicit quantile networks for distributional reinforcement learning," *arXiv preprint arXiv:1806.06923*, 2018.

[4] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *arXiv preprint arXiv:1710.02298*, 2017.

[5] "Fully connected neural network." https://www.extremetech.com/wp-content/uploads/2015/07/NeuralNetwork.png. Accessed: 2018-10-11.

[6] "Convolutional filters." https://cdn-images-1.medium.com/max/1600/1*EuSjHyyDRPAQUdKCKLTgIQ.png. Accessed: 2018-10-11.

[7] "Convolutional neural network." https://www.mdpi.com/entropy/entropy-19-00242/article_deploy/html/images/entropy-19-00242-g001.png. Accessed: 2018-10-11.

[8] "Quantile loss." https://i.stack.imgur.com/DmKq7.png. Accessed: 2019-03-13.

[9] "Cart-pole environment open ai gym." https://gym.openai.com/envs/CartPole-v1/. Accessed: 2018-11-05.

[10] "Lunar lander." https://gym.openai.com/envs/LunarLander-v2/. Accessed: 2018-11-05.

[11] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[12] R. K. Sinha, R. Pandey, and R. Pattnaik, "Deep learning for computer vision tasks: A review," *arXiv preprint arXiv:1804.03928*, 2018.

[13] C. R. Rubi, "A review: Speech recognition with deep learning methods," *International Journal of Computer Science and Mobile Computing*, vol. 4, no. 5, 2015.

[14] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Computational Intelligence Magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, p. 484, 2016.

[17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[18] OpenAI, "Openai five." https://blog.openai.com/openai-five/, 2018.

[19] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II." https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[20] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction.* Adaptive computation and machine learning, MIT Press, 1998.

[22] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[23] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, p. 386, 1958.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by backpropagating errors," *Nature*, vol. 323, no. 6088, p. 533, 1986.

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[26] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," *arXiv preprint arXiv:1707.06887*, 2017.

[27] R. Koenker and K. F. Hallock, "Quantile regression," *Journal of Economic Perspectives*, vol. 15, no. 4, pp. 143–156, 2001.

[28] P. J. Huber *et al.*, "Robust estimation of a location parameter," *The Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73–101, 1964.

[29] A. Müller, "Integral probability metrics and their generating classes of functions," *Advances in Applied Probability*, vol. 29, no. 2, pp. 429–443, 1997.

[30] D. White, "Mean, variance, and probabilistic criteria in finite markov decision processes: a review," *Journal of Optimization Theory and Applications*, vol. 56, no. 1, pp. 1–29, 1988.

[31] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," *arXiv preprint arXiv:1701.07875*, 2017.

[32] M. Rowland, M. G. Bellemare, W. Dabney, R. Munos, and Y. W. Teh, "An analysis of categorical distributional reinforcement learning," *arXiv preprint arXiv:1802.08163*, 2018.

[33] M. G. Bellemare, N. L. Roux, P. S. Castro, and S. Moitra, "Distributional reinforcement learning with linear function approximation," *arXiv preprint arXiv:1902.03149*, 2019.

[34] C. Lyle, P. S. Castro, and M. G. Bellemare, "A comparative analysis of expected and distributional reinforcement learning," *arXiv preprint arXiv:1901.11084*, 2019.

[35] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[36] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," *arXiv preprint arXiv:1611.05397*, 2016.

[37] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, "Integrating state representation learning into deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1394–1401, 2018.

[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[39] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[40] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, no. 5, pp. 834–846, 1983.

Glossary

List of Acronyms

RL Reinforcement Learning

DNN Deep Neural Network

CNN Convolutional Neural Network

DQN Deep Q-Network

PER Prioritized Experience Replay

QR-DQN Quantile Regression DQN

QR-W QR-DQN with Wasserstein metric based PER

CDF Cumulative Density Function

ALE Arcade Learning Environment

QR-TD QR-DQN with TD based PER

TD Temporal Difference

GAN Generative Adversarial Network

ANOVA Analysis of Variance