Prioritization Experience Replay and Distribution Reinforcement Learning using Quantile Regression

by

Manisha Gupta and Dhruv Parikh

EE-556 Stochastic Systems and Reinforcement Learning
Ming Hsieh Department of Electrical and Computer Engineering
Viterbi School of Engineering

Instructor:
Prof. Rahul Jain

Ming Hsieh Department of Electrical and Computer Engineering
Viterbi School of Engineering

May, 2021

Contents

Part A : Implementation of Deep Reinforcement Learning with Prioritization Experience Replay

Chapter 1

Introduction and Motivation

A rich literature in the Machine Learning community focus explicitly on the methods that deals with the issues regarding limited datasets [10]. Methods like resampling are used in both supervised and unsupervised learning. Such strategies manage various issues, for example, class imbalance, data scarcity, parameter estimation, etc.

Experience replay [8] takes a note of such resampling procedures and applies them to the space of Reinforcement Learning (RL). In a classical reinforcement learning setting, wherein the model learns on-line by interacting with the environment in real time, as soon as the algorithm 'sees an experience' (which could be state transition and reward), it disposes it away after updating the relevant model parameters. The essential thinking behind utilizing Experience Replay strategies is that, with the ability to re-utilize this experience, the RL model can learn considerably more rapidly with just the expense of memory stacking. The issue of computational cost associated with the memory storage, while significant, but is not a major problem in the Reinforcement Learning. Since being able to learn more quickly is a huge relative gain, especially in the situations where the environment-based interactivity of the model is in environments that are expensive/sensitive/critical. Additionally, RL models are known to be highly correlated, the inherent sequential nature of the RL problem makes the parameter updates also correlated to each other. The ER has been successfully applied in deep RL models, for example, Deep Q-Networks (DQN) [10], Double Deep Q-Networks (DDQN) [13]. In this work, we have focused our attention on deep RL models, to be specific, DQN and DDQN.

Stochastic Gradient Descent (SGD) is the most popular technique used in training DQN models[11] to solve for the relevant optimal quantities, and most SGD based methods assume independent and identically distributed (i.i.d.) nature of observations. This is a major issue which can be eased with the experience replay. But, even with all such advantages by Experience replay, there is a possibility that the DQN model disposes a 'rare experience' after using it to update its parameters just once, missing out on an underscored learning opportunity. To resolve this, we have used Prioritization Experiment replay technique [12].

Chapter 2

Background and Problem Statement

The motivation behind using Prioritized Experience Replay (PER) is through neuro-scientific evidence [2, 6], that have distinguished proof of sequential memory replay in the hippocampus of rodents, suggesting that sequences of prior experiences are replayed. The experiences which are related with high rewards seem, by all accounts, to be replayed much more frequently.

We have implemented various versions of Deep Q Learning [10, 13] to improve agent performance and tried to replicate state-of-the-art results highlighted in the papers referred for this project. The core of prioritization experience replay is to replay experiences which are causing significant learning to agent. The work in [12] advises to sample memory experiences which cause high loss after forward pass from neural network, also known as temporal difference (**TD**) error.

Temporal difference error acts as an indicator of how instructive the transition could be. We want to keep the experiences which led to an important difference between expected reward and the reward we actually get. TD error provides a mechanism to find the significance of a experience for the agent. In our approach we are using stochastic prioritization which is a robust method to learn a function approximator from samples.

2.1 Prioritization Experience Replay

Replay Memory is a collection or we can say buffer of all transitions from which agent is sampling randomly. The whole interaction of acquiring experience and examining from replay memory is called experience replay. Utilizing a replay memory prompts plan decisions at two levels : which experience to store and which experience to replay. PER delivers to later: replaying the critical experiences.

2.2 Prioritization using TD Error

The center of prioritization experience replay is the basis by which the significance of each experience is estimated. One idealized criterion would be the amount the RL agent can learn from a transition in its current state (expected learning progress). But as we can not quantify it straightforwardly, we measure the magnitude of signification of a transition with its TD error, which indicates how 'surprising' or unexpected the transition is: specifically, how far the value is from its next-step bootstrap estimate[1].

$$\delta_j = R_j + \gamma_j Q_{target}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1}). \tag{2.1}$$

In greedy TD error prioritization algorithm, it stores the last encountered TD error along with each transition in replay memory. A Q-learning update is applied to this transition, which updates the weights in proportion to the TD error. New transitions arrive without a known TD-error, so we put them at maximal priority in order to guarantee that all experience is seen at least once.

For implementing greedy TD error, we have used a binary heap data structure for the priority queue, for which finding the maximum priority transition when sampling is O(1) and updating priorities (with the new TD-error after a learning step) is O(logN).

2.3 Stochastic Prioritization

However, the greedy TD error prioritization has some major issues. It may possible that transition with low TD error on first encounter may not be replayed for a long time. Greedy prioritization focuses on a small subset of the experience: errors shrink slowly, especially when using function approximation,meaning that the initially high error transitions get replayed frequently. This lack of diversity that makes the system prone to over-fitting.

To overcome these issues, we have used stochastic sampling method. We interpolate between pure greedy prioritization and uniform random sampling. It ensures that the probability of being sampled is monotonic in a transition's priority, while guaranteeing a non-zero probability even for the lowest-priority transition.

We define the probability of sampling transition i as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{2.2}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. We consider a direct, proportional prioritization where $p_i = |\delta_i| + \epsilon$, with ϵ taken as a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero. The second variant is an indirect, rank-based prioritization where

$$p_i = \frac{1}{\text{rank}(i)},$$

where rank(i) is the rank of transition i when the replay memory is sorted according to $|\delta_i|$. In this case, P becomes a power-law distribution with exponent α. Both distributions are monotonic in $|\delta|$, but the latter is likely to be more robust, as it is insensitive to outliers.

2.4 Important Sampling Weights

The calculation of the expected value using stochastic updates is based on updates that have the same distribution as the expected value. Prioritized replay introduces bias because it adjusts the distribution in an uncontrolled manner, changing the solution to which the estimates will be applied to converge on (even if the policy and state distribution are fixed). To correct this bias, we have used Importance sampling weights as follows.

$$w_i = \left(\frac{1}{N}.\frac{1}{P(i)}\right)^{\beta}. \tag{2.3}$$

The weights fully compensate for non-uniform probabilities $P(i)$ if $\beta = 1$. These weights are folded into Q-Learning update by using $w_i\delta_i$ instead of δ_i (weighted Importance Sampling) [9]. We normalize weights by $\frac{1}{\max_i w_i}$ to scale the update downwards.
In Reinforcement learning, unbiased nature of updates are required for convergence at the end of training. To get this, by defining a schedule on the exponent β that reaches 1 only at the end of training, we take the advantage of annealing the sum of importance-sampling correction over time.

Algorithm 1: Double DQN Implementation with Proportionate Prioritization

```
1: **Input**: minibatch k, step-size $\eta$, replay period K and size N, exponents $\alpha$ and $\beta$, budget T
2: Initialize replay memory D =$\Phi$, $\Delta = 0$, $p_1 = 1$
3: Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
4: **for** $t = 1, 2, \ldots, T$ **do**
5:    Observe $S_t$;$R_t$;$\gamma_t$;
6:    Store transition $(S_{t-1}; A_{t-1}; R_t; \gamma_t; S_t)$ in D with maximal priority $p_t = max_{i<t}p_i$
7:    **if** t = 0 mod K **then**
8:       **for** $j = 1, 2, \ldots, k$ **do**
9:          Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
10:         Compute importance-sampling weight $w_j = (N.P(j))^{-\beta}/max_i w_i$
11:         Compute TD-error $\delta_j = R_j + \gamma_j Q_{target}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
12:         Update transition priority $p_j \leftarrow |\delta_j|$
13:         Accumulate weight-change $\Delta \leftarrow \Delta + w_j.\delta_j.\nabla_\theta Q(S_{j-1}, A_{j-1})$
14:      **end for**
15:      Update weights $\theta \leftarrow \theta + \eta.\Delta$, reset $\Delta = 0$
16:      From time to time copy weights into target network $\theta_target \leftarrow \theta$
17:   **end if**
18:   Choose action $A_t \sim \pi_\theta(S_t)$
19: **end for**
```

Chapter 3

System Study, Analysis and Design

We combine our prioritized replay algorithm with the state-of-the-art Double DQN algorithm to create a full-scale reinforcement learning agent. Our main change is to replace Double DQN's uniform random sampling with our stochastic prioritization and significance sampling methods. With every one of these ideas set up, we presently examine how much replay with such focused on inspecting can improve execution in sensible issue spaces.

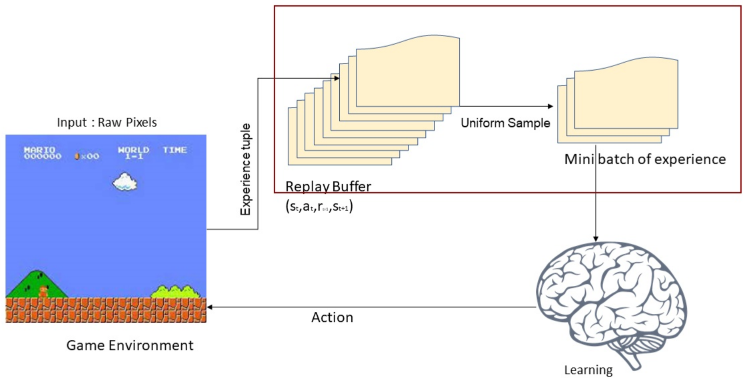


Figure 3.1: System Design

3.1 Deep Q Network

A deep Q network (DQN) is a multi-layered convolutional neural network that outputs a vector of action values given state s and network parameters θ. It is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$, where n is the dimension of the state space and m is the dimension of the action space. Three key elements of the Deep Q-network algorithm are experience replay, fixed target Q-networks, and limiting the range of rewards. Experience replay addresses the previously stated problem that rewards are often time-delayed. It helps break correlations in data and learn from all past policies. A bank of the most recent transitions are stored for some predetermined steps and sampled uniformly at random to update the network. In our implementation, we ran chosen environments with both prioritized and unpriortized experience replay to observe the results and learning rate.

$$y_t^{DQN} = r_{t+1} + \gamma max_a Q(s_{t+1}, a, \theta_t) \tag{3.1}$$

3.2 Double Deep Q Network

In both traditional and deep Q-learning algorithm, the max operator uses the same values to choose and evaluate an action, which can lead to greater estimation error, and, as a result, overconfidence. To mitigate this, we follow an approach proposed by van Hasselt, by assigning experiences randomly to update on of two value functions, which results in two sets of weights, θ and $\theta^{'}$. Each update, one is used to determine the greedy policy while the other determines its value. The target network in the deep Q-network model provides a a second value function without having to create another network. We evaluate the greedy policy with the online network, but then estimate its value with the target network. Thus our target becomes

$$y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t).\theta_t^{'}) \tag{3.2}$$

3.3 Prioritized Experience Replay

We have performed experiments using both Prioritized and unpriortized experience replay. The replay buffer used during experiment is of size 10000 from which a mini batch of size 32 is sampled (using stochastic prioritization). We are using $\alpha_o = 0.9$ and gradually decreasing it and $\beta_o = 0.4$ and gradually increasing it to 1(at the end of the training). For this project, we are doing proportionate prioritization and computing TD error using:

$$\delta_j = R_j + \gamma_j Q_{target}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1}) \tag{3.3}$$

3.4 Exploration vs. Exploitation

When training our agent, we used a linearly decaying ϵ-greedy approach. That is our agent selects the action a_{opt} which maximizes our estimated future value with probability $(1 - \epsilon)$ and uniformly selects from the valid actions otherwise.Each training epoch, we decreased ϵ by a fixed value. During training ϵ was initialized such that $\epsilon_i = 1$, as our agent had no knowledge of the world. Our target value was set to $\epsilon_f = .01$. During evaluation an testing, we set $\epsilon_{eval} = .01$ to limit exploration.

3.5 Convolution Neural Network Model

The model used worked directly with in-game raw pixel frames as input, so some image preprocessing was done in order to reduce dimensionality and lessen the computational load. Each frame was downsampled from the original $256x240$ pixels to an $84x84$ grayscaled image. A square input image was needed to use GPU-based 2D convolution. We denote this process for an in game frame j as ϕ_j

The convolutional neural network architecture is described below. It mirrors the network used in referred paper the Atari Learning Environment, however, with a couple modifications to the size of each layer just as minor changes of the filter itself.

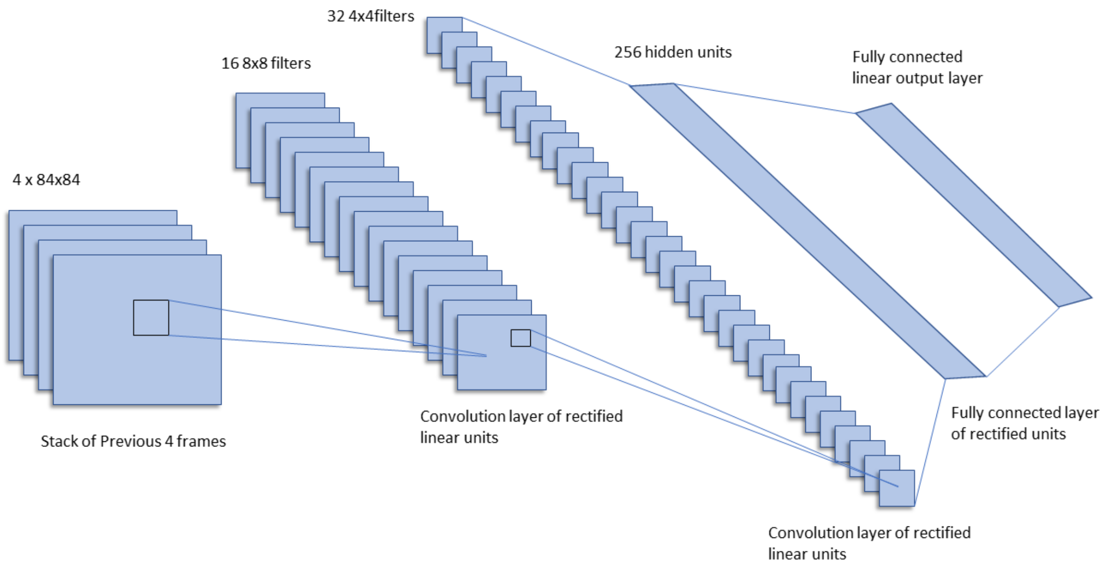


Figure 3.2: Convolution Neural network used as a functional approximation for the state to action map.

1. Input: Four grayscale frames with a resolution of 84 $\times$ 84 pixels
2. Hidden Layer: Convolves 16 8 $\times$ 8 filters of stride 4 with the input image and applies a rectifier nonlinearity
3. Hidden Layer: Convolves 32 4 $\times$ 4 filters of stride 2 and applies a rectifier nonlinearity
4. Hidden Layer: Fully connected layer that consists of 256 rectifier units
5. Output: Fully connected linear layer which outputs Q-values of each valid action

Chapter 4

Experiments

For this project, we have performed reinforcement learning experiments on 3 open AI gym [4] environments: Lunar Lander, Cart Pole and Super Mario Bros [7]. Please find code at following link : `https://github.com/Manisha2612/EE-556-Final-Project`. We now describe each of the experiment and discuss the results.

4.1 Lunar Lander

OpenAI Gym's Lunar Lander is an environment that takes in one of 4 discrete actions at each time step returns a state in an 8-dimensional continuous state space along with a reward. The environment simulates the situation where a lander needs to land at a specific location under low-gravity conditions, and has a well-defined physics engine implemented. The main goal of the game is to direct the agent to the landing pad with as softly and fuel-efficiently as possible. The state space is continuous as in real physics, but the action space is discrete. There are four discrete actions available: do nothing, fire left orientation engine, fire right orientation engine, and fire main engine. Firing the left and right engines introduces a torque on the lander, which causes it to rotate, and makes stabilizing difficult.

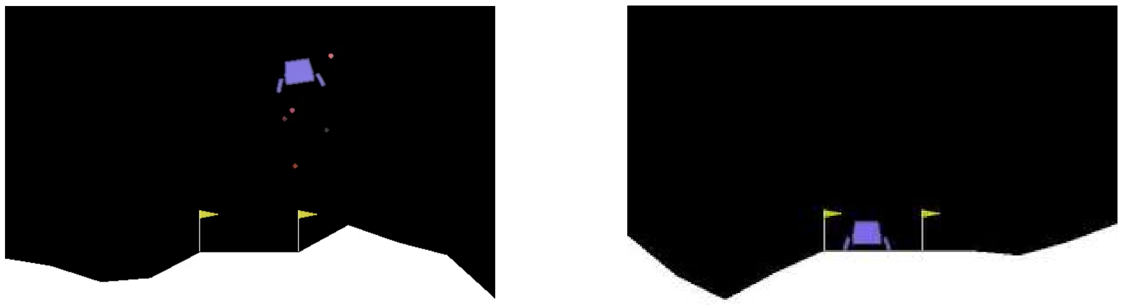

(a) Moving Lander (b) Successfully Landed

Figure 4.1: Lunar Lander environment

The goal of the problem is to direct the lander to the landing pad between two flag poles as softly and fuel efficiently as possible. Both legs of the lander should be in contact with the pad while landing. The lander should reach the landing pad as quickly as possible, while maintaining a stable posture and minimum angular speed. Also, once landed, the lander should not take off again. In the original problem, uncertainty is added by applying a randomized force to the center of the lander at the beginning of each iteration. This causes

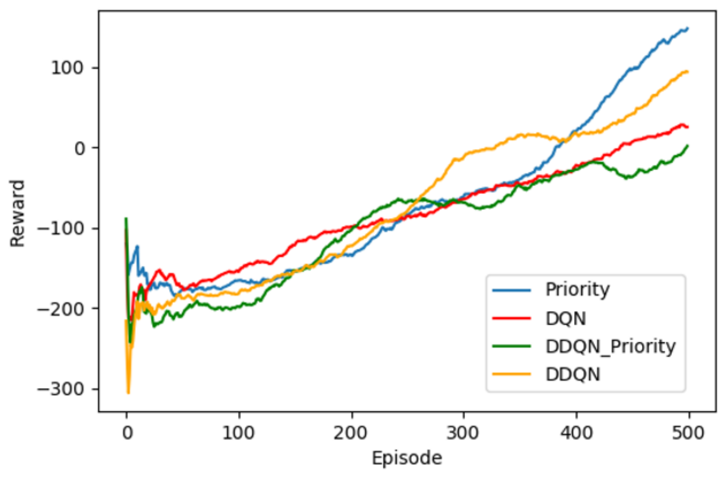


Figure 4.2: The resulting reward with various deep RL models for LunarLander-v2 with a total of 500 episodes.

the lander to be pushed in a random direction. The lander must recover from this force and head to the landing pad. Training was done for 4000 episodes with and without PER using DQN and Double DQN.

Above figure shows the average reward obtained (over the previous 500 episodes) by the DQN and DDQN agents with and without priority experience replay. At the start, the average reward is highly negative since the agent is essentially exploring all the time and collecting more transitions for storing in memory D. As the agent learns, the Q-values start to converge and the agent is able to get positive average reward. The PER implementation performs well and results in good scores. It surpasses DQN and DDQN to converge and get high score quick in comparison two other 2 agents.

4.2 Cart Pole

We have implemented another environment using OpenAI Gym to simulate the Cart-Pole system. Few snapshots of Cart-Pole states are shown in below figure. The left image shows the balanced state while the right image shows an imbalanced state. It consists of a cart (shown in black color) and a vertical bar attached to the cart using passive pivot joint. The cart can move left or right. The problem is to prevent the vertical bar from falling by moving the car left or right. The state vector for this system x is a four dimensional vector having components (x, x', θ, θ').The action has two states: left (0) and right (1).

It is observed that PER is slightly faster compared to the standard Deep Reinforcement algorithms and allows the use of continuous state values. However, for CartPole, it doesn't provide any significance improvement over DDQN and DQN architectures.

4.3 Super Mario Bros

Super Mario Brothers. is a side-looking over Nintendo game comprising two-dimensional levels where the player assumes responsibility for Mario and endeavors to explore towards the flagpole situated at the extreme right of each level, denoting the objective of the level. Each

(a) Balanced CartPole (b) Unbalanced CartPole

Figure 4.3: Cartpole environment with the objective of balancing the stick on a moving platform.

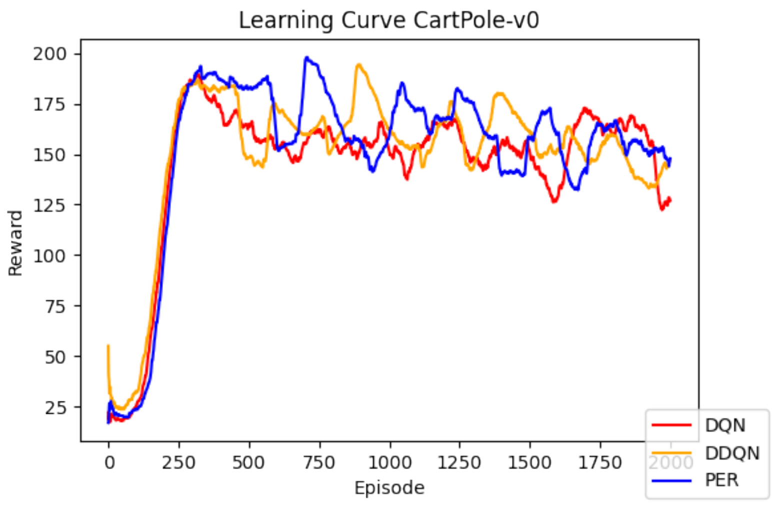
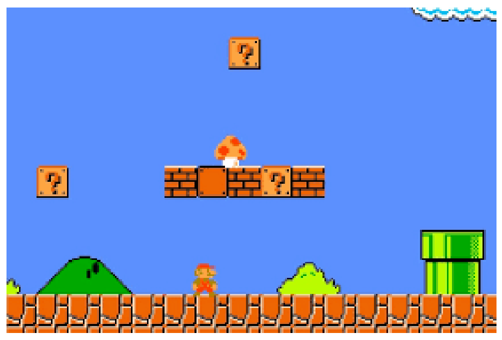


Figure 4.4: The resulting reward with various deep RL models for CartPole-V0 with a total of 2000 episodes.

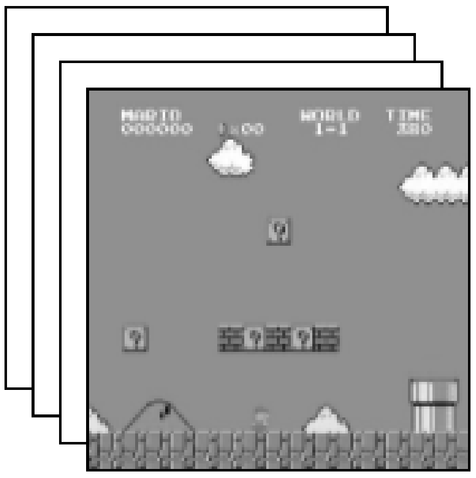
level comprises different tiles, which are their essential structure blocks. During this project, we studied to construct an RL Mario controller agent using Deep Reinforcement learning, which can learn from the game environment.

The essential goal of the game is to arrive at the flag post toward the finish of each stage without Mario losing the entirety of his lives. The optional goal is to acquire the most noteworthy score conceivable, which is done through gathering things, murdering adversaries, and finishing a level rapidly. Our agent will interact with game environment E with a sequence of actions, observations and rewards. At each time-step the agent selects an action at from the set of legal game actions. The action is passed to the emulator, which modifies its own internal state. Agent will observe the vector of raw pixels that represents the current on-screen frame. Agent will receive a reward determined by change in game score and distance to target destination.

We consider tasks in which our agent interacts with an environment E, in this case the NES emulator, in a sequence of actions, observations, and rewards. At each time-step the agent selects an action at from the set of legal game actions, A = 1, ..., K. The action is passed



(a) Gym Super Mario Original Frame



(b) Preprocessed Frame Stack

Figure 4.5: Super Mario Bros environment.

to the emulator, which modifies its own internal state and the game score. This internal state is not observed by the agent, which, rather observes a vector of raw pixels that represents the current on-screen frame. Additionally, the agent receives a reward r_t which is determined by a linear combination of the change in the total game score and the distance the agent moved to the right. The primary objective of the game is to reach the flag post at the end of each stage without Mario losing all of his lives. The secondary objective is to obtain the highest score possible, which is done through collecting items, killing enemies, and completing a level quickly

Evaluations were run at the end of every 5 episodes, which consisted of 10,000 time steps each Unfortunately, the results, while acceptable, were a bit below expectations. However, training was stopped at 6000 episodes with 2000 steps each time, but the agent was still learning and improving. A larger network and more training steps may have yielded more desirable results.

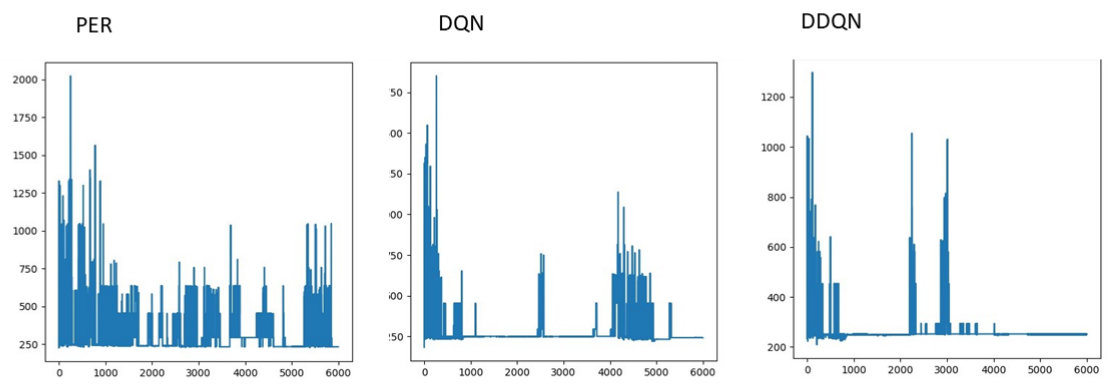


Figure 4.6: The resulting reward with using various techniques ranging from PER+DQN, DQN, DDQN from left to right for Super Mario Bros. A total of 6000 episodes were simulated.

We found that, while the agent showed moments of high-level play, where it would put together an excellent sequence of moves, it would just as often make ”unforced errors”, such as falling into a pit. The most pronounced roadblock occurs on the second stage, which is significantly more difficult than the first and requires a precise sequence of moves, including standing still, in order to effectively navigate it. With above diagram we can see, in PER agent is learning more frequently and producing higher rewards in comparison to DQN and DDQN. In the first figure above, which shows the average total reward using PER over each evaluation episode, the reward fluctuates greatly, as expected, but does not seem to trend positively as much as we had hoped. It is possible more training was simply needed in order for it to stabilize around a higher value.

Chapter 5

Conclusion

In this part we have implemented prioritized replay, a method that can make learning from experience replay more efficient. We studied a couple of variants, devised implementations that scale to large replay memories, and found that prioritized replay can speed up learning, leading to a new state-of-the-art of performance. We designed an automatic agent using Deep and Double Deep Q-learning based on the Atari Learning Environment in order to play Lunar Lander, CartPole and Super Mario Bros. With several run of episodes on different environments we observed that, while Prioritization Experience Replay provides faster learning to agent but it is also prone to noise and over-fitting. Hyper parameters and buffer size play an important role in the PER implementation.

While the agent was able to show successfully converge for Lunar Lander and CartPole environments, it was struggling with the challenging Super Mario. We observed that Deep Q-learning is an extremely effective technique when playing quick-moving, complex, short-horizon games with fairly immediate rewards, but does not perform as well in the long-horizon games, for example, Super Mario. Part of the difficulty of Mario and other long-horizon platform games is that, it requires precise timing and sequencing of actions in order to perform well. For future work on this project, we would first run a much longer training session to see the performance. We are also planning to explore other techniques like **A2C, A3C, Imitation Learning** methods which have shown improved results for complicated environments.

Part B : Distribution RL using Quantile Regression

Chapter 6

Introduction and Motivation

Distributional reinforcement learning attempts to learn reward function distributions, instead of simply the expectations. The most straightforward motivation to do so is the fact that knowing a distribution is always much better than knowing only the expectation of that distribution.

Distributional RL particularly has been able to reach benchmark results in the online learning of various games in the Atari suite [3]. The basic approach and algorithm utilized by us in the current project closely follows that presented in the same paper [5]. An initial version of the paper [5] utilized an algorithm that attempted to minimize the KL distance (Kullback – Leibler) between the target and the current estimate, and the loss function was also correspondingly defined.

A major issue with the initial paper was the fact that there wasn't any theoretical verification regarding the performance of the algorithm, despite its favorable performance. The loss function defined via KL distance was not the most suitable one to work with, for multiple reasons, one being the fact that the sample gradient estimates so provided via KL loss were biased, and thus SGD (Stochastic Gradient Descent) couldn't be performed to optimize such a loss.

In addition to the above issue with the sample gradients being biased, KL distance projection didn't conserve the contractive property of the distributional Bellman operator. The distributional Bellman equation being contractive with respect to a special type of metric (Wasserstein metric) was an important result, shown in the initial paper.

A resolution of these issues was given in the [3] paper, whereby the parametrization and projection of the reward function distributions were modified so as to ensure that the post-projection operator stayed contractive.

With the resolution provided, the paper presented an approach, with theoretical verification, to perform online off-policy reinforcement learning, further motivating us to implement the algorithm on the Super Mario Bros. environment available as a sub-class of the standard Gym module.

Chapter 7

Background and Problem Statement

We will begin by first discussing the distributional Bellman operator, and its characteristics.

7.1 The Distributional Bellman Operator

We consider the following problem statement, A Markov Decision Process (infinite horizon) with the following parameters,

X, A, R, P, γ

for the state space, action space, reward function, probability transition matrix and the discount factor γ belonging to $[0, 1)$.

First, let us consider the following function,

$$Z^{\pi}(x, a) = \sum_{t=0}^{\infty} \gamma^t R(x_t, a_t) \quad \textit{with } x_0 = x \textit{ and } a_0 = a \tag{7.1}$$

Note that the reward function is time invariant. $Z^{\pi}(x, a)$ is a random variable with its own distribution. In effect, we have 'initialized' it as $x_0 = x$ and $a_0 = a$, meaning that we start with some state and action **fixed** and then we follow our policy π which may not necessarily be deterministic.
An important observation here is that for each state and action that we presume to begin the time horizon with, we still get a random variable for each such fixed initial state and action. Thus, in effect we have a matrix of distributions that we need to 'estimate', with the matrix represented as,

$$\mathbf{Z} = [Z^{\pi}(x, a)]_{(|X||A|)} \tag{7.2}$$

Z is a matrix of random variables of dimension |X||A|, where the bars represent the space cardinality. Consider an entry (i,j) of Z, we have then,

$$\mathbf{Z}(i, j) = [Z^{\pi}(x, a)]_{(x_j, a_j)} \tag{7.3}$$

Where x_i and a_j are the i^{th} state and the j^{th} action respectively. A recursive relation, much like the one we write for action value and value functions, can also, then, be written for the distributional reward function, as follows,

$$Z^{\pi}(x, a) = R(x, a) + \gamma \sum_{t=0}^{\infty} \gamma^t R(x_{t+1}, a_{t+1}) \tag{7.4}$$

Giving us,

$$Z^{\pi}(x, a) = R(x, a) + \gamma Z^{\pi}(x', a') \overset{\Delta}{=} \textit{Distributional Bellman} \tag{7.5}$$

A few noteworthy points of the Distributional Bellman equation are, that, x' and a' are random next state and next action, additionally, once these random next quantities are realized, the function $Z^\pi(x', a')$ is again a random quantity (a distribution).
Finally, we define the distributional Bellman operator, using the same equation,

$$\mathbb{T}^\pi Z^\pi(x, a) = R(x, a) + \gamma Z^\pi(x', a') \triangleq \textit{Distributional Bellman Operator} \tag{7.6}$$

Where, x' $\sim$ P and a' $\sim \pi$

The distribution Bellman operator, $\mathbb{T}^\pi$ takes in an entire matrix of distributions (Z, the current one), updates the distribution corresponding to x,a, using the corresponding (current) reward and the distribution corresponding to x',a'. Thus, the operator is itself a distributional operator, and estimates distributions.

Similar to issues faced with generic Q Learning, whereby the tabular method becomes infeasible for situations where state and action spaces are huge, a 'tabular distributional' method becomes infeasible for learning the distributions.

To overcome this issue, we use Deep Neural Networks, which parametrizes the distributional reward function, and that learns the distributions for each state and action pair. The main problem statement we deal with here, is that of creating an algorithm that suitably updates the Deep Neural Network parameters, such that, we can use online off-policy techniques, to learn the distributions.

Before we end this initial discussion, a final note is that of the relationship between the $Q^\pi(x, a)$ action value function and the $Z^\pi(x, a)$ distributional reward function, which goes thus,

$$Q^\pi(x, a) = \mathbb{E}Z^\pi(x, a) \tag{7.7}$$

That is, that the Q value is the expectation of the Z distribution, which is not that difficult to see. In fact, the motivation in defining the Z distribution was the form of Q value function definition.

Chapter 8

System Study, Analysis and Design

Having defined the preliminaries, we will now discuss the basic system study and analysis points.

8.1 System Study and Analysis

Our first topic here will deal with understanding the metric associated closely with the distributional Bellman operator.

8.1.1 Wasserstein Metric

Having defined and deliberated upon the distributional Bellman operator, a natural question to ask is if the Bellman operator is contractive. If so, via the Banach Fixed Point Theorem, eventual convergence is guaranteed asymptotically.
As we will see, the distributional Bellman operator is indeed contractive, however not with any traditional metrics, but with a metric known as the Wasserstein Metric.
Again, our goal is to 'measure the gap' between two distributions, since **Z** is a matrix of distributions. We will first begin by defining the Wasserstein Metric, which 'measures the gap' between two CDF's. The p – Wasserstein metric (p –**W**) is defined as follows,

$$W_p(X,Y) = \left[\int_0^1 |F_X^{-1}(t) - F_Y^{-1}(t)|^p dt\right]^{1/p} \tag{8.1}$$

Where F^{-1} are the inverse CDF's, defined as,

$$F_X^{-1}(t) = inf\{x \in R : F_X(x) \geq t\} \tag{8.2}$$

Quite simply put, for input t in (0,1] the F^{-1} finds the smallest realization of the random variable for which its CDF at that point is at least t.
Specifically, for t=∞, we have,

$$W_\infty(X,Y) = sup\left[|F_X^{-1}(t) - F_Y^{-1}(t)|\right] \tag{8.3}$$

Thus, the ∞- **W** metric measures the magnitude of the largest gap between the two inverse CDF's, at the same point t. Intuitively, the following diagram shows the 1 – **W** loss between two CDF's, X and Y.

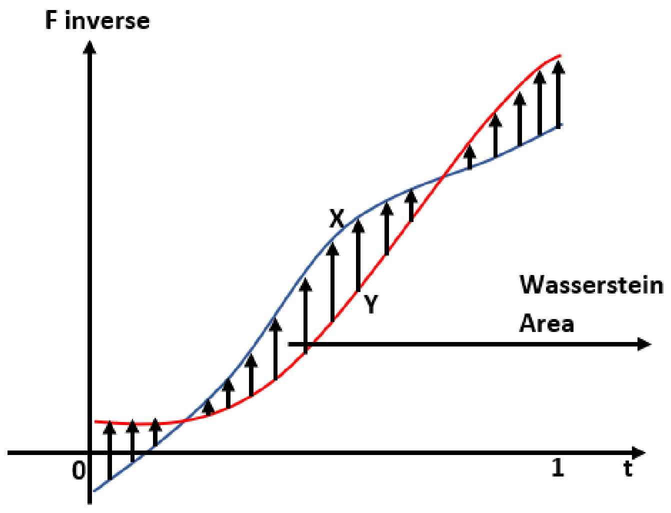


Figure 8.1: Wasserstein Area (p = 1)

As seen in the above diagram, the Wasserstein Area is shown in the blue arrowed lines. X and Y represent the inverse CDF's. It is clear from the figure that Wasserstein metric is an effective way to measure the disparity between two distributions.
Next, we discuss the way to measure the disparity between two matrices of distributions, which is what we really need, since Z is a matrix of distributions.

8.1.2 The d Metric

In order to evaluate the behavior of the T^{π} operator, when it is iteratively applied to an initial matrix of distributions, we define the following d Metric, between matrices of distributions/random variables,

$$d_p(Z_1, Z_2) = sup\ W_p\left(Z_1(x,a), Z_2(x,a)\right)\ \ over\ x, a \in X \times A \tag{8.4}$$

Where,

$$Z_1, Z_2 \in \{Z : X \times A \rightarrow Distribution,\ \ such\ that\ \mathbb{E}|Z(x,a)|^p < \infty, \forall\ x, a\ and\ p \geq 1\} \tag{8.5}$$

Thus, we only consider this metric to be defined for those matrix distributions Z which have all moments finite, for all the distributions within.
The above metric will be shorthanded p − d and we additionally note that this metric calculates the largest p − **W** metric, for corresponding x,a pairs, in two random distribution matrices. Thus, this metric is nothing but the **W** metric, with the only change being, it is the largest such **W** gap, that we can find in corresponding positions, between two **Z** matrix containing distributions within.

8.1.3 Contraction of the Distributional Bellman Operator

As has been hinted a priori, the Distributional Bellman operator is a contractive operator, with the metric of contraction being the p − d metric, for all p ≥ 1.
Thus, we have,

$$d_p(\mathbb{T}^{\pi} Z_1, \mathbb{T}^{\pi} Z_2) \leq \gamma d_p(Z_1, Z_2) \tag{8.6}$$

Where, γ is in $[0, 1)$.

We can now see the reasoning behind defining the Wasserstein metric, with this metric, the Bellman operator for distribution is contractive. Thus, using Banach FPT, convergence is asymptotically guaranteed.

Again, as much as being a contractive operator eases our convergence understanding of the Bellman equation, it does not give us a practical way of obtaining the distributions. Iteratively utilizing the equation for all states and actions, in a state and action space that is extremely large, is completely inconsequential.

An additional note that we will end this portion with is that despite being the metric for contraction, the p $-$ **W** metric cannot be used to define a loss function, that might attempt to optimize a neural network. Such a loss function (p $-$ **d** or p $-$ **W**) does not give unbiased sample gradient estimates, as has been teased before. Thus, we need some approach to define a loss function that has unbiased gradient estimates, along with minimizing the p $-$ **d** or p $-$ **W** gap.

8.1.4 Parametrizing Reward Distribution

Our parametrization of the Reward Distribution CDF (Z(x,a)) will be in the form of quantiles. To see how we parametrize the distribution, we consider the following steps:

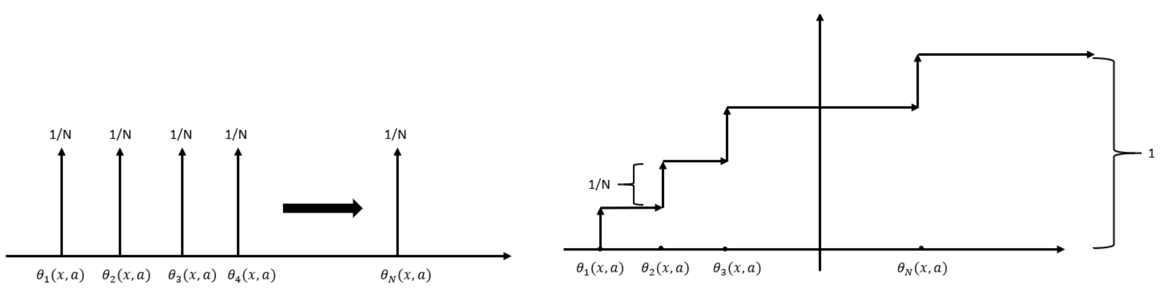
1. Place N weights, each weighing 1/N, at N points
2. These points will be estimated (parametrized) by the Deep Neural Network
3. Using the PDF obtained in step 1, we will obtain the CDF
4. We will use a target, given by Distributional Bellman operator, and compare it with our current estimate
5. Post comparison via a suitable metric (not the p $-$ **d** as mentioned earlier), we will then update the network parameters

To visualize the parametrization of Z(x,a), we redefine it as,

$$Z(x, a) = Z_{\boldsymbol{\theta}}(x, a) = Z(x, a, \boldsymbol{\theta}) \tag{8.7}$$

Which basically means that Z(x,a) is now being viewed as a function of the current network parameters. Additionally, to visualize the PDF and the CDF which we will be estimating, we look at the below figures. The quantile parametrization of the CDF and PDF for $Z_{\boldsymbol{\theta}}(x, a)$ is quite apparent.

Thus, rather than estimating/parametrizing the PDF/CDF values, we instead parametrize the locations with the PDF weights fixed. An important note here is the difference between θ and $\boldsymbol{\theta}$. The bold $\boldsymbol{\theta}$ represents the network parameters, while the normal θ represents the location of the weights as outputted by the neural network (on the basis of $\boldsymbol{\theta}$, state input, action input)
Finally, we ask an important question: How do we estimate the $\{\theta_i(x, a)\}_{i=1,2,\ldots,N}$ such that our Wasserstein loss is minimized?



(a) PDF (b) CDF

8.1.5 Minimizing the Wasserstein Loss for Quantiles

We consider two random variables, Z and $Z_{\boldsymbol{\theta}}$ where the latter represents the output of the neural network and the former represents a random variable whose CDF and PDF is known (known distribution).
We now need a technique via which the $\{\theta_i\}$ are estimated, such that we Wasserstein loss between Z and $Z_{\boldsymbol{\theta}}$ is minimized. Not only that, we need a loss function, other than the Wasserstein loss, that ensures this minimization. Had the Wasserstein loss been able to provide us with unbiased sample gradient estimates, this 'search' for a similar loss wouldn't be needed; we'd simply calculate the target via Bellman, obtain the current output from the network, and use the Wasserstein loss directly between target and current output to minimize it. Since Wasserstein gives biased gradient estimates, we need a different loss that works in parallel with Wasserstein.

First, we define the CDF points,

$$\tau_i = i/N \ for \ i = 0, 1, 2, \ldots, N$$

τ_i is simply the CDF height from θ_i to θ_{i+1}

Next, we state the following result,

Wasserstein Loss Minimizer: The Wasserstein loss,

$$W_1(Z, Z_{\boldsymbol{\theta}})$$

Is minimized when,

$$\left\{\theta_i : F_z(\theta_i) = \frac{\tau_{i-1} + \tau_i}{2} \ for \ i = 0, 1, 2, \ldots, N\right\} \tag{8.8}$$

Thus, the $1 - \mathbf{W}$ loss is minimized when θ are estimated according to the above rule. Or, When θ for $Z_{\boldsymbol{\theta}}$ are estimated, such that,

$$\left\{\theta_i : \theta_i = F_z^{-1}\left(\frac{\tau_{i-1} + \tau_i}{2}\right) \ for \ i = 0, 1, 2, \ldots, N\right\} \tag{8.9}$$

Then, the $1 - \mathbf{W}$ loss is minimized between Z and $Z_{\boldsymbol{\theta}}$ distributions, where Z is an RV whose distribution $F_Z(.)$ is known and $Z_{\boldsymbol{\theta}}$ is its N quantile estimate. Our goal is to estimate θ_i for each distribution in the above minimizing fashion. Additionally, we define,

$$\tau_i' = \frac{\tau_{i-1} + \tau_i}{2} \ for \ i = 0, 1, 2, \ldots, N\} \tag{8.10}$$

Normally, at this point, if the gradient estimates were unbiased, we'd be done with our discussion with an algorithm at hand.

Targets would be obtained via Bellman updates, network would provide us with the network estimate, and finally we'd use the $1 - \mathbf{W}$ loss function to minimize the loss between the Bellman updates and the network estimates, using the same loss to update the network parameters. In essence, this section would also be not required. However, optimizing via $1 - \mathbf{W}$ loss is not possible, as has been elucidated before. We therefore need a new loss function, a loss function which is minimized with the same criterion that the $1 - \mathbf{W}$ loss function is minimized, and in order to 'construct' such a loss we first needed to check where the $1 - \mathbf{W}$ loss is minimized so that our new loss can replicate it, legitimizing the necessity of this section.

Before we define such a loss, we diverge to discuss an additional important element: We really don't have the actual distribution as our target, we only have the estimate. Bellman updates don't give the actual distribution but the estimate of the distribution.

8.1.6 The Estimate Issue

We have already discussed why this issue creeps up. Instead of having Z, we actually only have its estimate. Thus, we note the following:

$$argmin\ \mathbb{E}W_p(\hat{Z}, Z_{\boldsymbol{\theta}}) \neq argmin\ W_p(Z, Z_{\boldsymbol{\theta}})\ (in\ general) \tag{8.11}$$

The E appears due to 'batch optimization'. Note, by argument, we mean the network parameters. Effectively, this inequality says that it is always possible that the network parameters updated via the estimate are not the same as those that may be updated via the actual distribution. As obvious as this fact seems, it presents an unavoidable issue in our algorithm since our minimization efforts are on distribution estimates as opposed to the actual distribution. Again, this issue is unavoidable, and can only be mitigated by selecting large batch sizes for optimization.

8.1.7 Quantile Regression Loss and Quantile Huber Loss

We finally discuss the penultimate topic: The QR loss function. Again, as discussed before, we need a loss function that minimizes 'similarly' to how our $1 - \mathbf{W}$ loss gets minimized, that is at estimates of $\{\theta_i\}$ around the points $\{F_Z^{-1}(\tau_i')\}$.
The Quantile Regression loss does just that. Additionally, it is an asymmetric convex loss function whose sample gradients are unbiased, and thus allows us to use SGD based optimization strategies (Munos, 2017). Consider $\hat{Z}$, τ_i' and θ_i as defined earlier.

Consider $\hat{Z}, \tau_i'$ and θ_i and θ_i as defined earlier, the QR loss is then defined as,

$$L_Q R^{\tau_i'}(\theta_i) = \mathbb{E}\rho_{\tau_i'}(\hat{Z} - \theta_i)$$

$$\rho_(\tau_i')(u) = u\left(\tau_i' - \delta_{\{u<0\}}\right)\ for\ all\ u \in R$$

The above loss function is automatically minimized when,

$$\theta_i = F_Z^{-1}(\tau_i')$$

Which is precisely when the 1 – W loss is minimized. The overall loss function is obtained as,

$$L_{QR}(\tau', \theta) = \sum_{i=1}^{N} L_{QR}^{\tau_i'}(\theta_i)$$

Which, again, is minimized when,

$$\{\theta_i\} = \{F_Z^{-1}(\tau_i')\} \ for \ all \ i$$

Which is the same criterion for the 1 – **W** loss minimization. Note, finally, that this loss function estimates the sample gradients in an unbiased fashion, thus, solving the issue that we faced for the 1 – **W** loss. Additionally, the 1 – **W** loss and the above loss are minimized equivalently.

The Huber loss is a slightly modified version of the above QR loss, defined as,

$$L_k(u) = \begin{cases} \frac{1}{2}u^2, & \text{if } |u| \leq k \\ k\left(|u| - \frac{1}{2}k\right), & \text{otherwise} \end{cases} \tag{8.12}$$

And, the final functional is defined using,

$$\rho_{\tau_i'}^{k}(u) = |\tau_i' - \delta_{u<0}| L_k(u) \tag{8.13}$$

The L_{HQR} will use the above function inside the $\mathbb{E}$. Now, after having defined our loss function parallel to how 1 – W loss is minimized, we can now define the following projection,

$$\pi_{W_1} Z = arg \ min_{Z_{\boldsymbol{\theta}}} W_1(Z, Z_{\boldsymbol{\theta}}) \tag{8.14}$$

Which gives us the $Z_{\boldsymbol{\theta}}$ that minimizes the 1 – W loss, or, the QR loss. Note that this search for $Z_{\boldsymbol{\theta}}$ will be over a set of quantile distributions (as we've parametrized them so).

8.1.8 Overall Contraction

Again, we note that in our actual algorithm we first use Bellman to obtain an update, then, we use the quantile regression loss between the estimate (of the network) and the update, and update our network parameters by minimizing this QR loss or the 1 – **W** loss equivalently. The QR loss, once minimized, gives us the projection $\pi_{W_1} Z$ from the Bellman update Z that we currently have. In effect, we are not just using Bellman, but also using the projection in our algorithm. Thus, we ask the question, is our overall operator contractive? As it turns out, it actually is, and we state the following result (Munos, 2017),

$$d_\infty\left(\pi_{W_1}\mathbb{T}^\pi \boldsymbol{Z}_1, \pi_{W_1}\mathbb{T}^\pi \boldsymbol{Z}_2\right) \leq \gamma d_\infty(\boldsymbol{Z}_1, \boldsymbol{Z}_2) \tag{8.15}$$

Which states that our combined operation of finding a target update using Bellman, and then 'projecting' onto our final quantile set by minimizing the 1 – **W** loss (or the QR loss), is a contractive operator. Note, that Bellman operator $\mathbb{T}^\pi \boldsymbol{Z}_1$ updates a matrix of distributions, which is then projected individual distribution wise onto the quantile set that minimizes the loss. Thus, $\mathbb{T}^\pi \boldsymbol{Z}_1$ operates on the entire matrix, and, $\pi_{W_1}\mathbb{T}^\pi \boldsymbol{Z}_1$operates also on the entire matrix of distributions, but, individual entry wise (minimizing distributions over all distributions in the matrix).
This result thus guarantees convergence of our algorithm, that we discuss next.

8.1.9 Final Algorithm

Based on our above exhaustive discussion, we can now provide a final algorithm that can be used to learn an RL problem, using Quantile Regression. We first consider the generic DQN algorithm (Mnih, 2015), and then update it minimally, so that it can perform the RL task via QR theory described above. The basic strategy is, that, we learn online, and use Temporal Difference techniques to learn online, as opposed to learning exhaustively.

The Algorithm 1 gives the algorithm to perform DQN via Raw Image pixels obtained. The algorithm has been adopted from (Mnih, 2015).

Algorithm 1: DQN for Raw Image inputs

```
 1: Initialize Replay Buffer = $\mathcal{D}$, Initialize Buffer Capacity
 2: Initialize $Q$ with random weights
 3: for $episodes = 1, 2, \ldots, M$ do
 4:     Capture initial image sequence and process the sequence
 5:     $s_1$=$\{x_1\}$, $\phi_1 = \phi(s_1)$
 6:     $s_1 \to$ image sequences, $x_1 \to$ individual image, $\phi \to$ preprocessor, $\phi_1 \to$ final state
 7:     for $time = 1, 2, \ldots, T$ do
 8:         Use $\epsilon$ greedy policy to select next action (with $\epsilon$ select $a_t$ randomly, $1 - \epsilon$ select
            $a_t = \arg\max Q(\phi_t, a; \boldsymbol{\theta})$
 9:         Get next image, update state sequence, process state sequence
10:         Get reward
11:         Store experience $\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$ Buffer
12:         Sample from $\mathcal{D}$ randomly, a batch of experience
13:         A sampled batch = $(\phi_j, a_j, r_j, \phi_{j+1})$
14:         for for all experiences sampled do
15:             Get target
                $\text{target} = y_j = \begin{cases} r_j & \phi_j\text{terminal} \\ r_j + \gamma \arg\max_{a'} Q(\phi_{j+1}, a'; \boldsymbol{\theta}) & \end{cases}$
16:             Get current value
                Current value = $val_j = Q(\phi_j, a_j; \boldsymbol{\theta})$
17:             Append both to their vectors
18:         end for
19:         Optimize on (target, current value) vector via some loss and some optimizer Update
            $\boldsymbol{\theta}$
20:     end for
21: end for
```

The above Algorithm 1, is then modified so that we can use our Quantile Regression algorithm, which is seen in Algorithm 2. The main three modifications are:

1. In normal DQN, an input state gives a vector of Q values. For us, we need, for an input state, a set of distributions for all actions corresponding to that state. Each action for a state has an N sized vector for its quantile distribution, and thus, the output layer for us gives a matrix of dimension N x $|\mathbb{A}|$, whereby for Mnih DQN it gives a vector of length $|\mathbb{A}|$

2. We optimize via the Quantile Regression loss function

3. Optimizer in Mnih DQN was RMSProp, the current paper suggests using Adam.

For our QR algorithm, everything else stays pretty much same.

Algorithm 2: QR using Raw Images (Modification to Algorithm 1)

```
1: # Take Algorithm 1 and modify it as thus
2: Inputs: $N$ (the quantile number), $k$ if Huber Loss is used, $x, a, r, x', \gamma$, the quantile
   weights at the $N$ locations are $\{q_i\}$ ($1/N$ weights, generally)
3: $a$ is chosen greedily as in Algorithm 1, $x, x'$ are processed versions ($\phi$ in Algorithm 1)
4: Get Bellman Target
5: Compute $Q(x', a) = \mathbb{E}Z(x', a') = \sum q_i\theta_i(x', a')$, $Z(x', a') = Z_\theta(x', a')$ outputted from the
   Neural Network
   (Note that the NN outputs all distributions for all actions for $x'$)
6: Next, calculate optimal action $a^*$
7: Bellman Update,
   $\mathbb{T}\theta_j = r + \gamma\theta_j(x', a^*)$ (for all j) (updates distributions) (is our target)
   (in reality, only greedily updates $\theta_j(x, a)$ distribution for all j)
8: Calculate NN prediction,
   $\theta_i(x, a)$(current value) (for all i)
9: Calculate QR/HQR loss as $\sum_{i=1}^{N} \mathbb{E}_j[\rho^k_{\tau'_i}(\mathbb{T}\theta_j - \theta_i(x, a))]$
10: Using loss, update $\boldsymbol{\theta}$, the NN network parameters (use Adam optimizer)
```

The Algorithm 2 still selects actions greedily, however, it then calculates distributional Bellman update based on the action that maximizes the next state Q value function, obtained by taking expectation over the distributions obtained corresponding to it. It then calculates the current value obtained from the network (current distribution), and since we only update via Bellman, the distribution corresponding to current state and current action, we obtain and minimize the corresponding state, action pair quantile loss, since rest all distributions stay the same.

In effect, the basic architecture of DQN is still valid, the basic pre-processing of images and state representation also stays same, only how the updates are obtained and evaluated, along with network output layer output dimension, is modified.

This minimally modified DQN Algorithm (Algorithm 2), is the one we have used for our experiments.

Chapter 9

Experiments

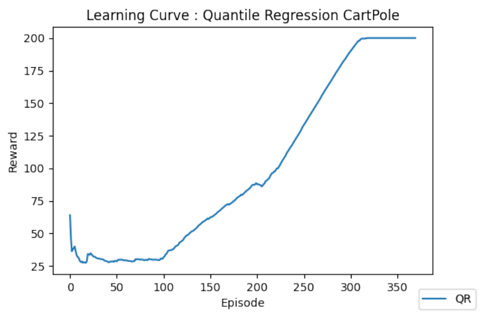
We have already discussed the Convolutional Neural Network Architecture that we have implemented in the previous section of the report (Part 1). Thus, those details will be skipped here to avoid redundancy. Additionally, we note the following few points:

1. Our implementation of Algorithm 2, utilized a hindsight experience storage approach whereby highly negative reward experiences (such as death, something that 'resets' the environment) were also stored, despite being 'sequentially out of order'. For an illustration, each episode for our Super Mario Bros. implementation, lasted for 100 – 500 Mario deaths, and each death did not reset the image buffer, but added the death image and the experience into the buffer. Image buffer was only reset in the beginning of each episode.

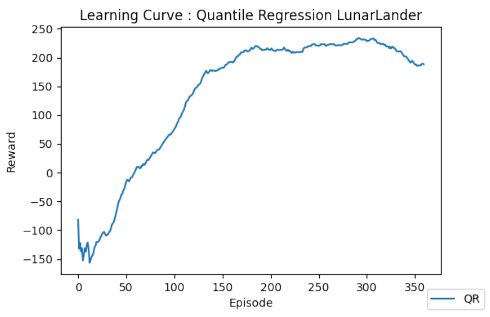
2. Image buffer retained the last 4 – 6 images for Mario, specifically. Thus, a state comprised of these last images, which were then processed, and sent to the NN as input.

3. The buffer for the replay (experience storage buffer) was around 5k – 10k.

The results we obtained are presented in the following figures.



(a) QR for CartPole



(b) QR for Lunar Lander

Figure 9.1: CartPole-V0 and LunarLander-V2 Environment.

The results we obtained for Lunar Lander and Cartpole environments with the QR algorithm (Algorithm 2) were pretty good, and the reward continuously increased with the episodes
However, QR for Mario did not converge with the episodes that we were able to run. Our computational capacity as restricted to 100 – 200 episodes, and Mario as an environment

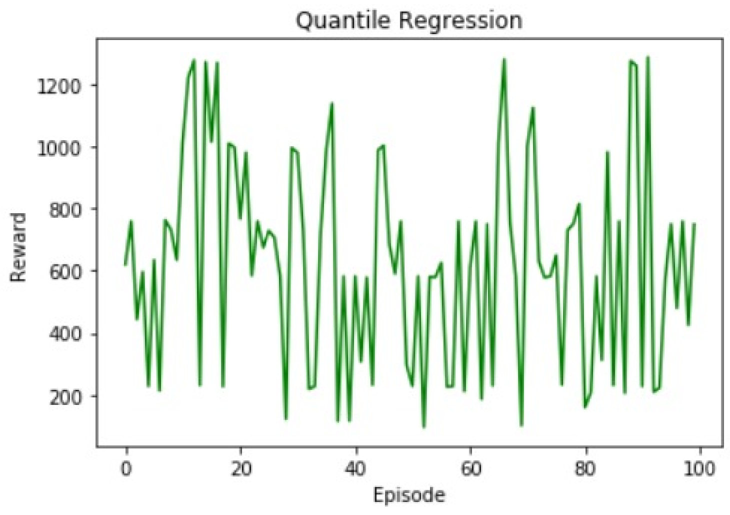
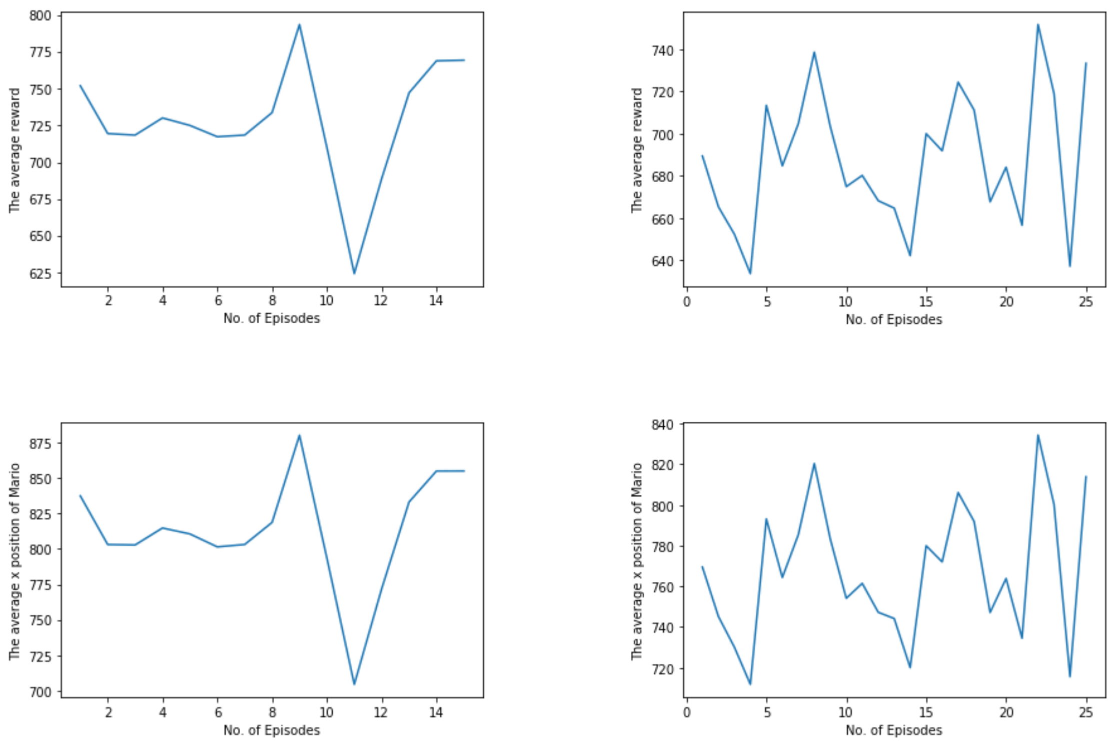
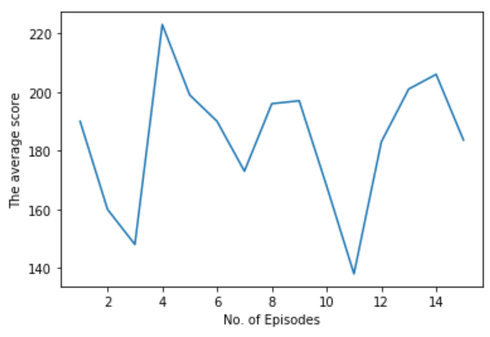


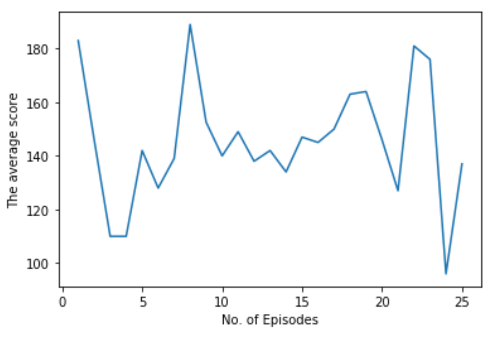
Figure 9.2: Super Mario bros

is pretty complicated, needing at least around a million episodes to learn the environment fully.

Additionally, with the Mario environment, we noticed that tuning the hyper – parameters was extremely important, since changes in them vastly affected the outcome.







The above two set of figures correspond to small runs, and represent the modification in the QR output for the Mario environment for minor change in hyper – parameters (specifically we changed the learning rate and the epsilon for the epsilon greedy policy, along with a few other minor changes in the hyper – parameters). Thus, we remark that the Mario environment was sensitive to:

1. Hyper – parameters (epsilon greedy policy)
2. The initial start

As such, we do note, that since we were only able to run the environment for a comparatively miniscule number of episodes due to computational limitations, these remarks may be biased and could become redundant if Mario was allowed to fully explore the environment for a large number of episodes.
We also remark that as seen in the Cartpole and the Lunar Lander results, the QR algorithm does work, and the implementation is able to optimize the policy selected using raw input image frames.

Chapter 10

Conclusion

In this joint project effort, in effect, we covered: DQN, DDQN, Distributional QR, Prioritization Experience Replay, and we ran our implementations of these algorithms on Lunar Lander, Cartpole and Super Mario Bros. environment. In overall conclusion, the results for the Lunar Lander and the Cartpole environment were quite favorable since these environments are much simpler to work with as opposed to Mario.
For Super Mario Bros. the main limitations were the computational restrictions and the deadlines we had. Further project exploration to improve on the Distributional QR algorithm, such that it can learn better from distribution estimates, is possible, along with analyzing the Prioritization Experience Replay framework much more in depth theoretically, since both these techniques have reached benchmark results on a multitude of game suites.

References

[1] David Andre, Nir Friedman, and Ronald Parr. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*. MIT Press, 1998.

[2] Laura A. Atherton, David Dupret, and Jack R. Mellor. Memory trace replay: the shaping of memory consolidation by neuromodulation. *Trends Neurosci.*, 38(9):560, Sep 2015. doi: 10.1016/j.tins.2015.07.004.

[3] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning, 2017.

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

[5] Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression, 2017.

[6] David J. Foster and Matthew A. Wilson. Reverse replay of behavioural sequences in hippocampal place cells during the awake state. *Nature*, 440:680–683, Mar 2006. ISSN 1476-4687. doi: 10.1038/nature04587.

[7] Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. URL `https://github.com/Kautenja/gym-super-mario-bros`.

[8] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8(3):293–321, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992699.

[9] A. Mahmood, H. V. Hasselt, and R. Sutton. Weighted importance sampling for off-policy learning with linear function approximation. In *NIPS*, 2014.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, Feb 2015. ISSN 1476-4687. doi: 10.1038/nature14236.

[12] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[13] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL `http://arxiv.org/abs/1509.06461`.