Project 1 (Navigation) Report

Overview

This submission implements three different Reinforcement Learning algorithms in order to examine primarily how quickly it solves the environment. The Deep Q-Learning algorithm (DQN) serves as a baseline. The second algorithm is the Double DQN algorithm (Double DQN). The final algorithm implemented is the Prioritized Experience Replay (PER), which is extended from Double DQN.

Learning Algorithm

The model for all the algorithms was kept fixed:

- Input layer: 37 (This corresponds to the total number of states)
- Hidden Layer 1: 64, with ReLU activation
- Hidden Layer 2: 64, with ReLU activation
- Output Layer: 4 (This corresponds to the number of actions)

The hyper-parameters were also kept constant across implementations:

- Batch size: 64:
- Discount Factor, γ: 0.99
- Optimizer uses Adam with a Learning Rate of 0.0005

Deep Q Network

In order to compute the Temporal Difference (TD) error, we take the difference between:

$$Q_{\text{target}} - \hat{Q}$$

However, we never really know what the real value of Q_{target} is, and therefore have to provide an estimate for it using the Bellman equation, which is given by the following equation:

$$Q_{\text{target}}(s,a) = R + \gamma max_a \hat{Q}(s',a)$$

However, there is a problem with this formulation: The weights used are essentially *the same* because they are applied to both Q_{target} and $\hat{Q}$. So each time we get closer to the target, the target would shift too. This phenomenon would potentially lead to training to become very unstable.

Enter DQN with *fixed targets*. Instead of simply using one network, we use two identical networks, one called Q_{target} and the Q_{local}.

This allows two separate parameters. The parameters used in Q_{target}, denoted as w¯, are used to estimate the TD target, and are only updated every τ steps (this is specified as a hyper-parameter), where the parameters of Q_{target} are now updated to use the current parameters of Q_{local}, denoted as w.

Double Deep Q Network

Recall how Q_{target} is calculated:

$$Q_{\text{target}}(s,a) = R + \gamma max_a \hat{Q}(s',a)$$

The second term is the discounted maximum Q-value of all the possible actions starting from the next state. The main problem is that during the *beginning* of training, the agent has little experience, and so whatever Q values that result at the beginning of learning are going to be very noisy and therefore not reliable.

The same strategy of decoupling is once again exploited:

$$Q_{\text{target}}(s,a) = R + \gamma Q_{\text{target}}(s', argmax_a Q_{\text{local}}(s',a))$$

Here, we use Q_{local} to pick the best action for the next state but we use Q_{target} to calculate the Q value of taking that action (i.e. chosen by Q_{local}) at that state.

This alleviates the problem of overestimation of the Q values, consequently leading to fewer fluctuations during learning.

Prioritized Experience Replay

The two above implementations use a Replay Buffer that simply stores a fixed number of experiences. If the buffer is full, the earliest experiences are discarded in favor of the new one.

During each learning step, the experiences from the buffer are sampled uniformly.

This random way of selecting experiences isn't optimal since experiences are not created equal. In particular, there are some experiences that lead to higher learning (i.e. larger TD error), which others might have little value.

Therefore Prioritized Experience Replay (PER) aims to solve this problem by selecting the best experiences to replay to the agent, which also making sure new experiences get to be chosen while also making sure that often seen samples are not seen as often.

Broadly, for each experience, the *priority* is computed as:

$$p_t = |\delta_t| + \epsilon$$

Here, the priority is the magnitude of the TD error plus some small value ϵ. This is to ensure that every p_t is larger than zero and therefore have some probability of being picked.

The next step is a way to prioritize each experience. However, we cannot be too greedy and simply select the experiences with the largest priorities. This idea is similar to the epsilon-greedy policy of choosing actions.

Hence, with another hyper-parameter α, the probability of an experience being chosen for replay is:

$$P(i) = p_i^a / \sum_k p_k^a$$

Having a value of $\alpha = 0$ is the same as choosing any experience with uniform randomness while $\alpha = 1$ means greedily selecting experiences with the highest priorities.

The final thing that needs to be accounted for is during weight updates. The problem stems from the *distribution* of where the sampled experiences come from.

Before, the experiences are uniformly sampled, but now because we are selecting the experiences based on priority (and sometimes randomly), some bias has been introduced towards selecting higher priority experiences.

In order to correct for this bias, *importance sampling* weights are used. The purpose of this is to make higher priority samples have lower importance since they will be seen more often.

The other thing to consider is that the learning from high priority experiences are more important towards to beginning of learning, and less so at the end.

Towards the end of learning, we want the samples to be as unbiased as possible.

To that end, another hyper-parameter, β, is introduced. The formula to compute the importance sampling weights is:

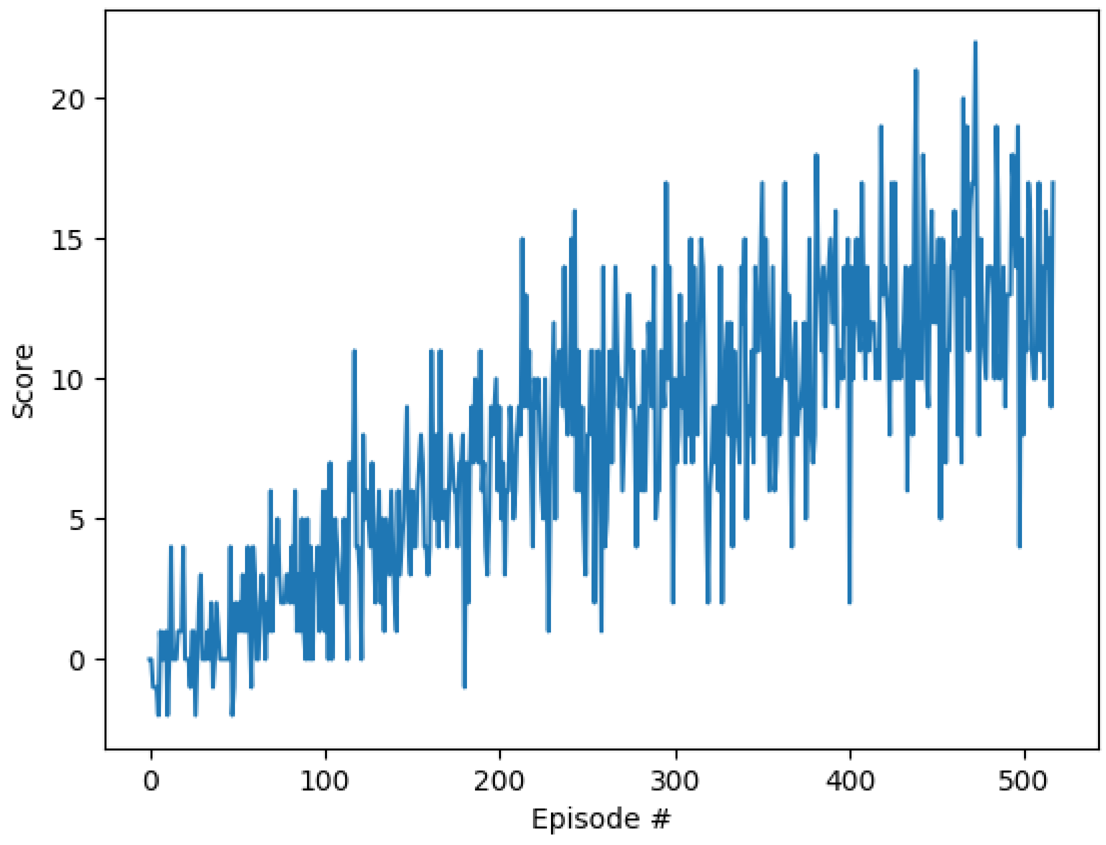
$$(N \times P(i))^{-\beta}$$

β is annealed up to 1 during training so that towards the end of training (hopefully during convergence), the weight updates become less biased.

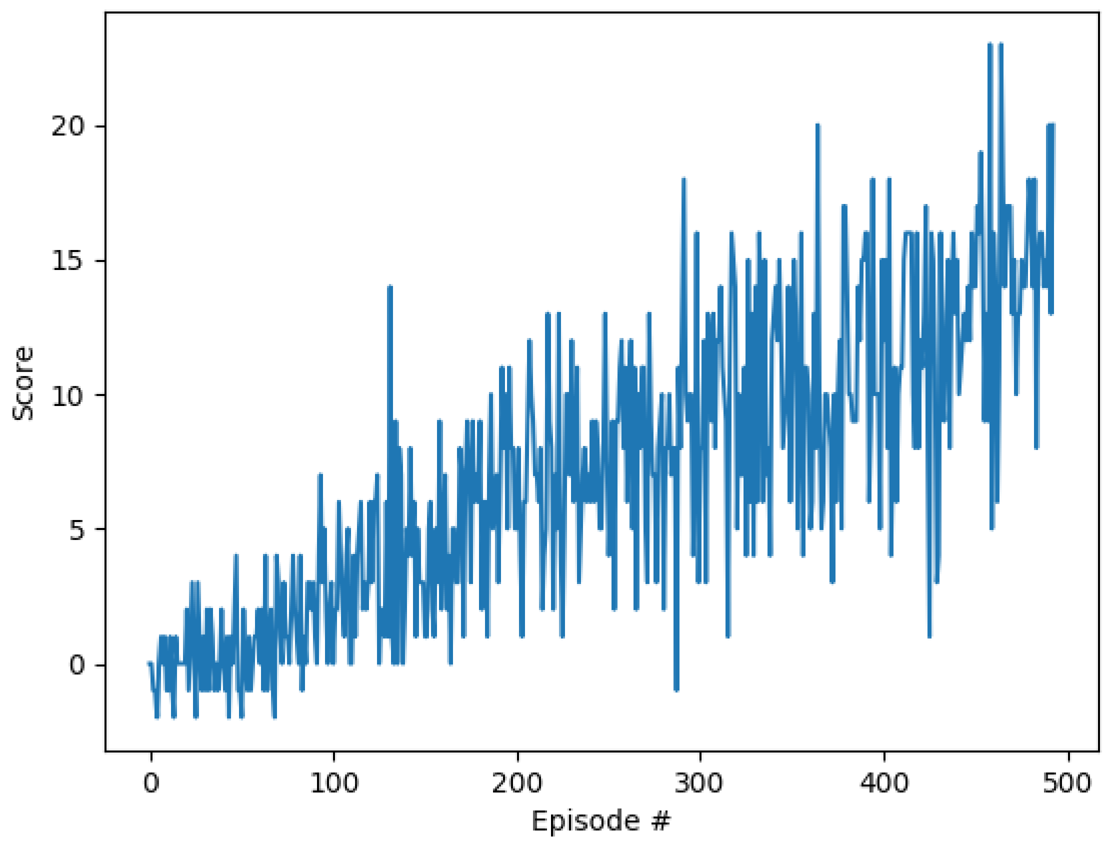
Plots of Rewards

The plots of rewards per episode for all three algorithms are included to illustrate that the agent is able to receive an average reward (over 100 episodes) of at least +13.

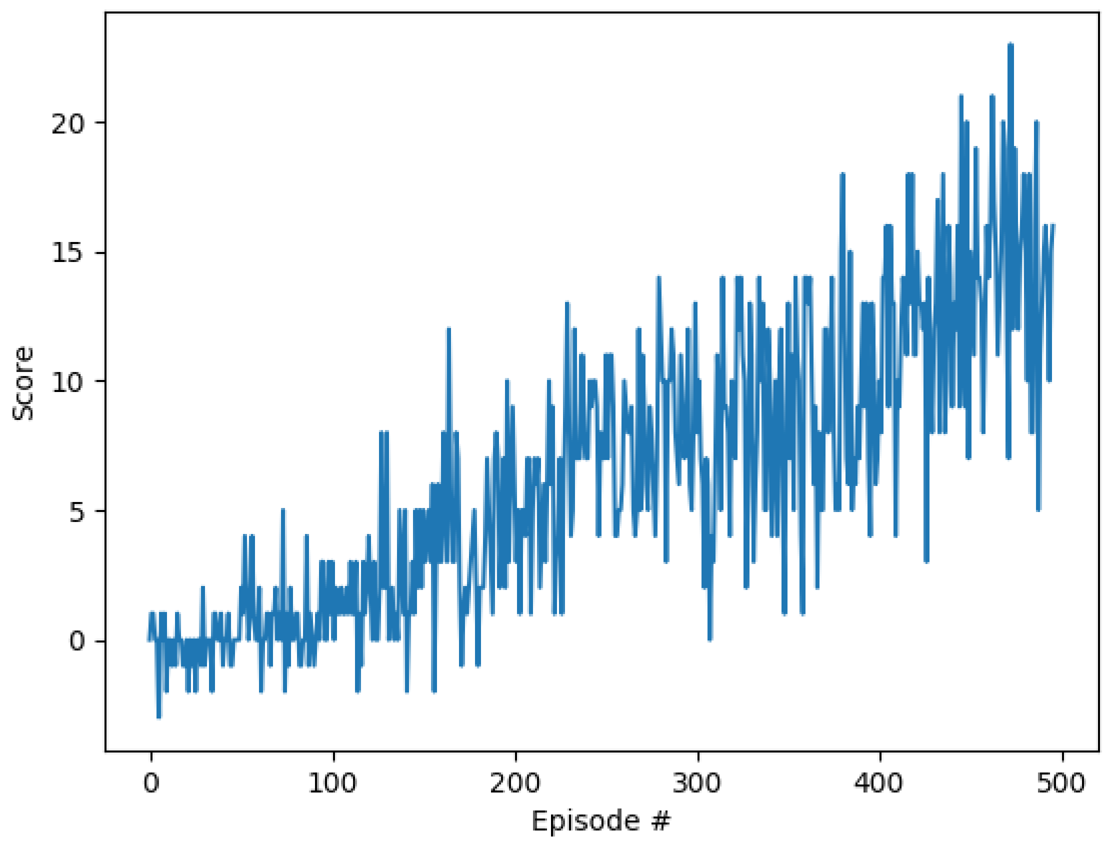
Deep Q Network (417 Episodes)



Double Deep Q Network (393 Episodes)



Double Deep Q Network + Prioritized Experience Replay (396 Episodes)



Ideas for Future Work

Dueling DQN is something that I have left out, and it would be interesting to see how it compares with the rest of the implementations. Then, combining all these independent algorithms by using the Rainbow would be very interesting too.

I would also have liked to see how the various algorithms would have fared by using a pure pixel approach using a much deeper model like VGG-19 or ResNet-101.