Unity Banana Collection Navigation

Austin Silveria

We examine the DQN algorithm applied to the Unity Banana Collection Environment. Our implementation includes the Double DQN and Priortized Experience Replay (PER) methods to enhance the general DQN algorithm. We then analyze the performance of our agent and propose ideas for future improvements.

1 Learning Algorithm

Deep Q-Learning is a reinforcement learning algorithm that uses neural networks to approximate the optimal policy for achieving maximum reward in the given environment. Experiences (state, action, reward, and next state) are added to memory as the agent interacts with the environment to eventually be sampled and learned from. It was first proposed to evaluate states and actions on one network (local model) while the next state and epsilon-greed chosen action be evaluated on a separate network (target model) to maintain fixed Q-Targets and stabilize learning. The Double DQN method enhances this by choosing an action for the current state with the epsilon-greedy policy from the local model but evaluating it on the target model. This reduces overestimation of Q-Values by the online weights and allows further stabilization of learning. This Q-Value from the current state and the Q-Value from the next state are then used to compute the TD Error based on the Bellman Equation.

Based on Prioritized Experience Replay, experiences are also given priorities based on the magnitude of their TD Error and sampled from memory based on this assigned priority. This allows the agent to replay experiences that “it has more to learn from” more often and converge to the optimal policy quicker. To allow sampling of experiences and updating of priorities at O(log N) time complexity, it was necessary to implement a Sum Tree data structure. A random value between 0 and the sum of all priorities is chosen, and the array of priorities (represented as Sum Tree) is “walked” with all priorities being added along the way. The priority that is “landed on” (where the sum of all priorities “walked” equals the randomly chosen value) is the experience we sample from the replay buffer of experiences. The Sum Tree data structure allows us to “walk” the array and update priority sums through propagation in O(log N) time complexity. A bias is introduced to the policy as a result of non-uniform sampling, which can be corrected with an Importance-Sampling Weight (IS Weight) that is folded into the TD Error. The IS Weight is computed based on the probability of the experience being chosen and reduces TD Error to correct for the bias. Two new hyperparameters are introduced for control of prioritization, Alpha and Beta. Alpha was chosen to be .6 which corresponds to sampling depending 60% on prioritization and 40% on uniform. Beta was started at .4 and linearly annealed to 1 by the time of the 325th episode (near end of training). This value corresponds to how much the prioritization bias is corrected by the IS Weight, with a value of 1 giving a full correction which is necessary for stability at the end of training. We also decreased the learning rate slightly for PER to compensate for the fact that we are dealing with higher error samples more often.

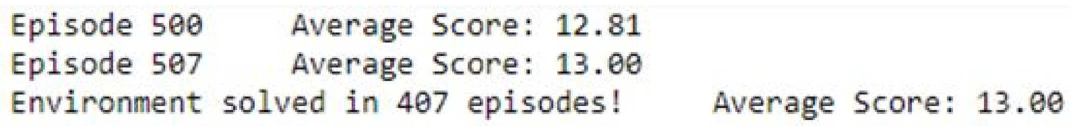
Hyperparameter	**Double DQN**	**Double DQN PER**
Replay Buffer Size	100,000	100,000
Batch Size	64	64
Gamma	.99	.99
Tau	.001	.001
Learning Rate	.0005	.0004
Update Every	4	4
Alpha	~	.6
Beta	~	.4

TABLE 1. The chosen hyperparameters for our implementation.

Table 1 shows the eight chosen hyperparameters for our two implementation. The replay buffer size (size of memory) was chosen to be 100,000 due to the low state and action complexity of the environment. The batch size corresponds to the number of experiences sampled at a time and can be adjusted based on the GPU memory of the running system to maximize utilization if running in parallel. A standard value of .99 was chosen for gamma to discount future rewards when computing the expected Q-Value with the Bellman Equation. A tau value of .001 corresponds to the “softness” of an update to the target model with respect to the local model. A smaller value pushes the weights of the target model’s network in the direction of the local model a smaller amount than a larger value. As a final hyperparameter, it was chosen to have the agent learn every 4 time steps to allow the memory to become saturated with new experiences before sampling again.

2 Performance Analysis

With a state dimension of 37 and an action dimension of 4, a network of two fully connected hidden layers was used to approximate the optimal policy for choosing actions corresponding to maximum Q-Values. Both layers consisted of 64 neurons and used the ReLU activation function. With such a small network it was noticed that there was little utilization when training on a GPU and it was taking longer than training on a CPU. This was most likely due to the network training speed being much faster than the transfer speed of the data to the GPU because of the small network size.



```
Episode 500     Average Score: 12.81
Episode 507     Average Score: 13.00
Environment solved in 407 episodes!     Average Score: 13.00
```

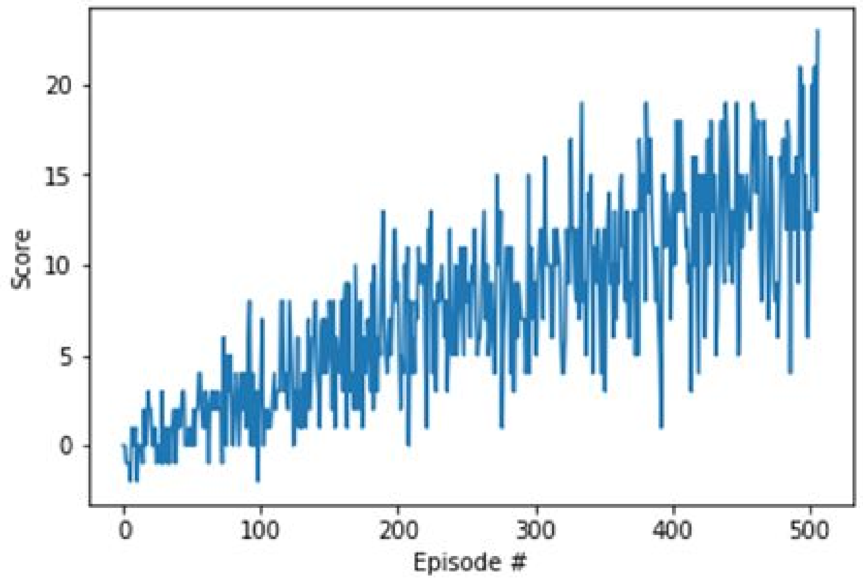


Figure 1: **Vanilla DQN** performance on the Unity Banana Collection environment using the same hyperparameters as Double DQN shown in Table 1.

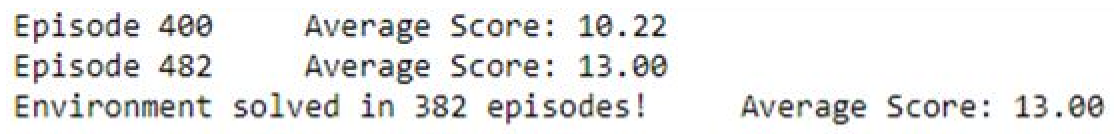
Figure 1 illustrates the learning of an agent using the Vanilla DQN algorithm over the required number of episodes needed to solve the environment. This served as a baseline for our improvements with the implementation of a Double DQN and Prioritized Experience Replay.

Figure 2 and Figure 3 illustrate the learning with the Double DQN and Double DQN PER algorithms respectively. In each iteration of improvements it can be seen that the agent maintains a steeper increase of rewards over time, converging to the optimal policy quicker. Double DQN stabilizes the learning compared to the vanilla algorithm as can be seen with the tighter spread of rewards as it progresses over time. The Double DQN PER does lose some of this stabilization due to the fact that it is generally working with larger gradients because the experiences are prioritized by the magnitude of their TD Error. Some of this stabilization can be recovered by reducing the learning rate, but in our experiments we found that with a somewhat unstable learning rate the agent still converged to the optimal policy faster.

3 Future Work

Dueling DQN: It has been shown that isolating the algorithm's estimation of a state's value from state-dependent action values can lead to more generalized learning and result in a better quality policy at the end of training. This RL innovation follows a pattern of isolation to improve the performance of learning agents. The Double DQN algorithm isolates choice of action from evaluation of action to reduce overestimation due to a single network determining both state and action value. Prioritized Experience Replay further isolates the value of a state but in a different way than a dueling network. While the dueling network learns the value of being in a state, PER estimates the how much can be learned from a state through the proxy of TD Error.

Data Loader: Our implementation suffered from slower runtimes when running on a GPU because its utilization was low, most likely pointing to the fact that data transfer was more costly than evaluation on a CPU. A data loader running in parallel with multiple workers could transfer the experience data to the GPU in advance to better utilize the GPU during training.



```
Episode 400     Average Score: 10.22
Episode 482     Average Score: 13.00
Environment solved in 382 episodes!     Average Score: 13.00
```

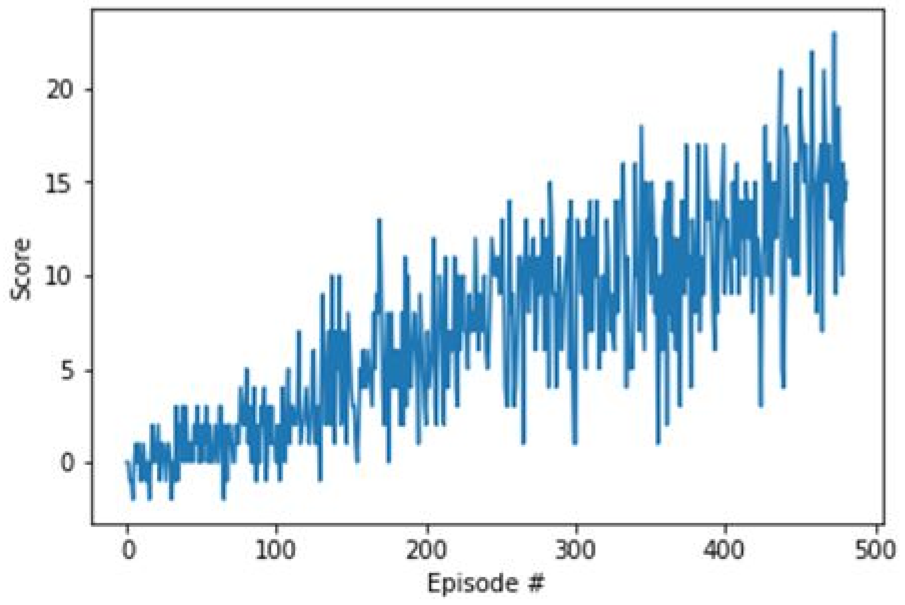
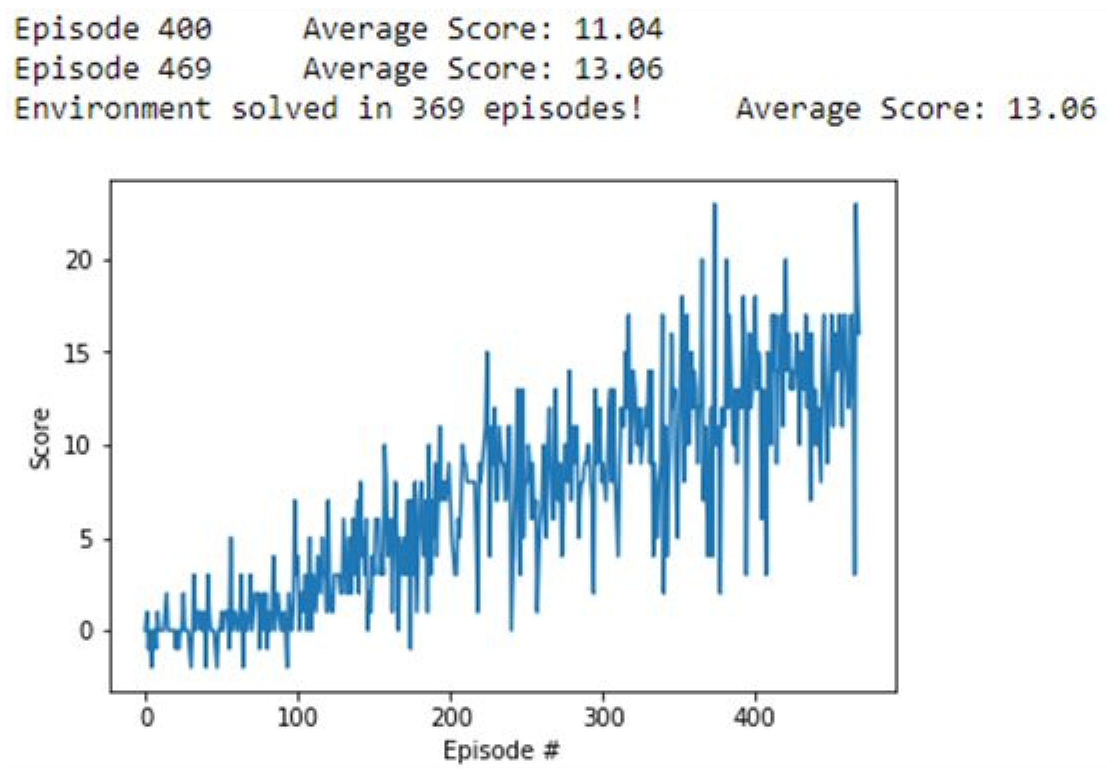


Figure 2: **Double DQN** performance on the Unity Banana Collection environment using the Double DQN hyperparameters shown in Table 1.



```
Episode 400     Average Score: 11.04
Episode 469     Average Score: 13.06
Environment solved in 369 episodes!     Average Score: 13.06
```

Figure 3: **Double DQN PER** performance on the Unity Banana Collection environment using the Double DQN PER hyperparameters shown in Table 1.