CSE 410 Project 2: Value Function Approximation

Peter M. VanNostrand
Department of Computer Science
University at Buffalo
Buffalo, NY 14261
pmvannos@buffalo.edu

Abstract

In this project we implemented a Deep Q-Learning, and a Double Deep Q-Learning. We then applied these learning agents to two learning environments. The first environment used is the OpenAI CartPole-v1 which aims to balance an inverted pendulum. The second environment is the OpenAI MountainCar-v0, which has an agent learn to drive a car to a destination up a hill. We then evaluated the performance of both algorithms on both environments and have included those results below.

1 Deep Q-Learning

Deep Q-Learning (DQN) is an extension of the popular Q-Learning algorithm. While Q-Learning is simple to implement in its tabular form, this limits the applicability of the Q-Learning algorithm. Deep Q-Learning expands this by formulating Q-Learning task as the optimization target of a Convolution Neural Network (CNN). This allows the Q-Learning agent to learn significantly more complex actions and applies to non-linear spaces.

1.1 Experience Replay

The Deep Q-Learning algorithm makes use of an approach called experience replay. In this method, a series of episodes are simulated or performed, and at each time step a tuple is stored containing the following five pieces of information

- Current state
- Action
- Reward
- Next state
- If the episode has completed

Then a random mini-batch of the recorded tuples is selected, and the DQN is trained on this mini-batch. This is done to allow the agent to learn non-linear functions. In general attempting to approximate a non-linear function is not stable, however by taking random mini-batches we can break the similarity of subsequent training samples and allow the agent to learn both linear and non-linear functions. If instead samples were trained sequentially the agent could be forced into a local minimum, creating a strong bias for the learning of earlier samples, and causes the performance of the agent to decline significantly.

1.2 Target Network

Another challenge posed by naïve Deep Q-Learning is the moving target problem. As the network is very sensitive to small changes in weights, backpropagating at every timestep often causes the q-network to undergo large oscillations. These can prevent the network for converging efficiently and greatly increase training times.

To overcome this issue, we make use of a target network. Essentially, we make a copy of the deep network and freeze one copy. We then use the frozen network as the target for during training of the non-frozen network. After a given number of updates we then replaced the frozen copy with an updated copy of the trained network. This process means that the target changes much less frequently, preventing oscillations and allowing the network to converge much more quickly.

1.3 Q Function Representation

In a traditional Q-Learning algorithm, the q-value is often represented as $q(s,a)$, a function of the current state and the action to be taken. However, in DQN we write the q-function $q(s,w)$, which is done for several reasons. Firstly, for a space with m actions it allows all m Q-values to be computed in one forward propagation as the action is not a parameter of the function. Secondly it allows for more efficient backpropagation and therefore network learning. When we take a given sample (s,a,r,s') we calculate the q value for that state-action pair. We then determine the loss between the expected reward and the actual reward and use this value to perform backpropagation through the network. However, this loss only accounts for the one given action, so we want backpropagation to effect only one of the m q-value outputs. Writing the function as $q(s,w)$ rather than $q(s,a)$ allows us to perform backpropagation of one action which holding the remaining values constant. This change also allows the network to learn generalization of actions. In tabular Q-learning we learn a finite set of discrete state-action pairs, with the assumption that we will visit every stat-action pair at some point during training. However, in DQN the states are frequently a continuous value, with only the set of possible actions being discrete. This means that is it impossible for the agent to leave every possible state-action pair. Instead it must learn to generalize what is has learned to apply to similar, but novel states. The use of weights in the q-function allows this continuous generalization as the weights form a continuous function which acts on the input state to produce q-values estimates for all actions.

1.4 Environments

To evaluate our Deep Q-Learning (DQN) and Double Deep Q-Learning (D2QN) algorithms we used two environments from the OpenAI Gym library: CartPole-v0 and MountainCar-v1

1.4.1 CartPole-v0

The first, and simpler, of these environments is the cartpole problem. In this environment the agent controls a cart on a linear 1-dimensional track. On top of the cart a pole is balanced on a hinge acting as an inverted pendulum. The goal of the agent is to move the cart left and right such that the pole remains standing for a long as possible. An image of this environment is shown below

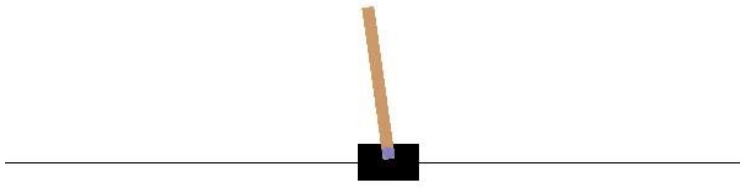


Figure 1: OpenAI CartPole Environment

Observation Space:
The observation space of this environment consists of the following four values

Table 1: OpenAI CartPole Observations

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Action Space:
Using these four observation values the agent must produce one of the two following actions

Table 2: OpenAI CartPole Actions

Num	Action
0	Push cart to the left
1	Push cart to the right

Reward Function:
In this environment the agent is rewarded with +1 for every time step taken, including the termination step

1.4.2 MountainCar-v0

The second, and more complex, of the environments is OpenAI's MountainCar-v0. This environment consists of a car placed in the valley of a two-dimensional slop. Atop the right slope is a hill with a flag marking the goal position. An image of this environment is shown below. The agent's goal is to guide the car from the valley to the flag on top of the hill by moving it left and right. Notably the car has insufficient engine power to climb the hill directly from a standstill. The only way for the car to reach its goal is to "rock" back and forth between the two slopes, building up speed each time until it has sufficient momentum to reach the top of the hill.

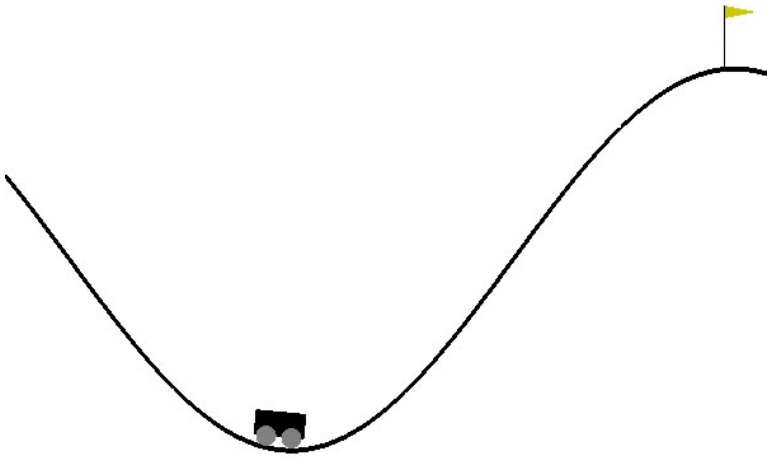


Figure 2: OpenAI MountainCar Environment

Observation Space:
The observation space of this environment consists simply of the following two values. The flag on the right hill is located a position of 0.5

Table 3: OpenAI MountainCar Observations

Num	Observation	Min	Max
0	Position	-1.2	0.6
1	Velocity	-0.07	0.07

Action Space:
The possible actions the agent can take are as follows

Table 4: OpenAI MountainCar Actions

Num	Action
0	Push left
1	No push
2	Push right

Reward Function:
In this environment the agent receives a reward of -1 for each time step until the agent reaches the goal position of 0.5

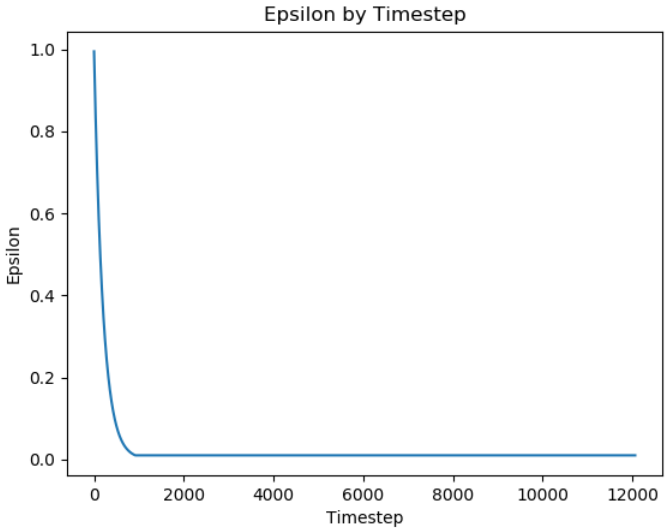
Note that the formulation of this environment makes it significantly more difficult to learn that the CartPole environment. In the CarPole problem the agents is immediately positively rewarded for actions that keep the pole upright, whereas in the MountainCar problem the agent is simply punished for failing to reach the goal. This means that the reward in this environment – the termination of its punishment – is a great many timesteps ahead of actions that lead to its success. This makes the connection between action and reward less clear, making the problem much harder to solve via conventional learning.

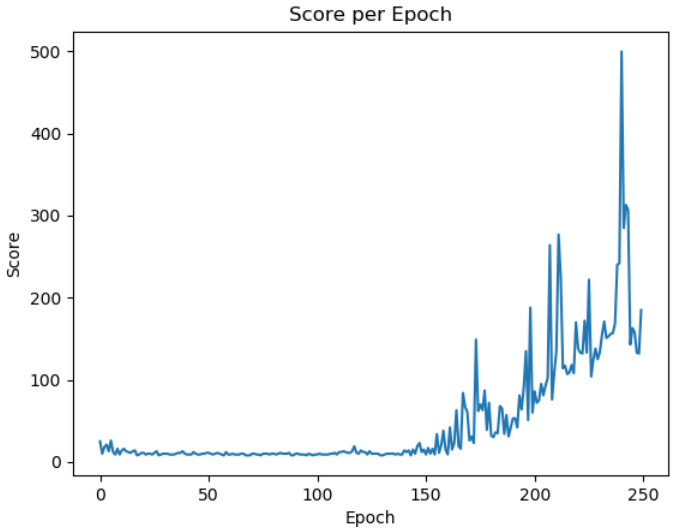
1.5 DQN Results

The following results were obtained from running the implemented Deep Q-Learning agent on the environments described above. Due to limited computational availability and the extremely long training time of some models, data for 250 epochs will be shown for all cases. This should be sufficient to show the converging trends when trained, even if not all cases reach complete convergence.

1.5.1 DQN CartPole Results

Results shown below are for the DQN agent used on the CartPole environment. The CNN for this environment consisted of two dense layers of size 24 with Relu activations, followed by a dense layer of size 2 with a linear activation function for generating the action q-values. The epsilon decay is performed each time step such that a large variety of experiences are sample in the first 1,000 or so timesteps to get the agent started. The agent is then left to refine its model using an $\epsilon = 0.01$ to ensure the occasional deviation from its learned method.



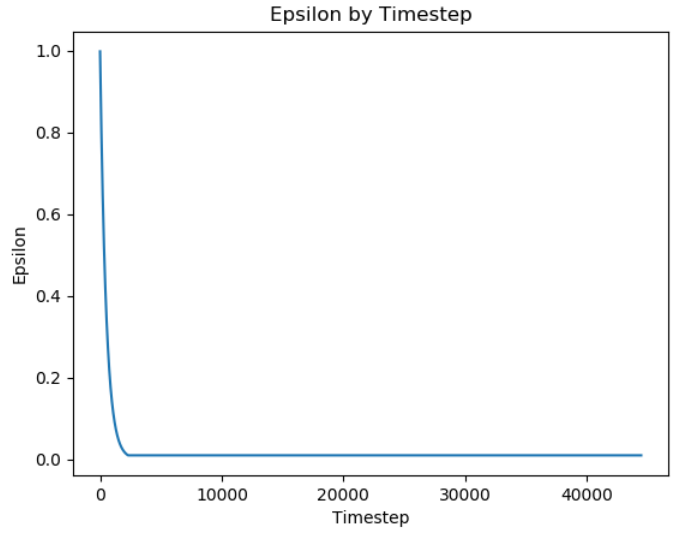


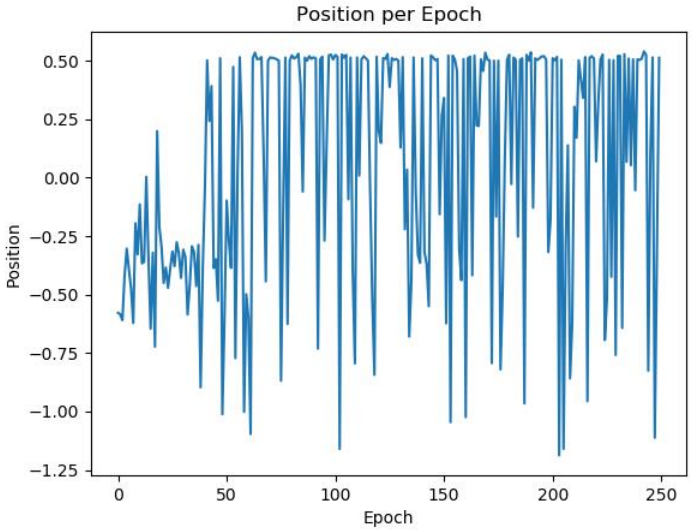
The score shown on the left is the total cumulative reward for a given episode, this is equivalent to the number of time steps the agent managed to keep the pole vertical. We see here that before approximately epoch 150, the score is relatively small as the agent begins to develop a coarse set of weights. Shortly after 150 the agent begins to refine this set of weights and the score increases sharply.

1.5.2 DQN MountainCar Results

Due to the more complex nature of the MountainCar environment a more complex network was necessary to allow the agent to account for the longer time between action and ultimate reward. Additionally, the epsilon decay is slightly lengthened to allow for more exploration. The same minimum $\epsilon = 0.01$ is also employed.

The network for this agent consists of a layer of 128 densely connected nodes with a Relu activation, followed by a layer 24 nodes with the same activation, and lastly a dense liner layer of size 3 for the action q-value outuputs.





Instead of plotting the total number of iterations in each episode, I have elected to plot the ending positing of the car after each episode. This shows whether the agent was successful in reaching its goal of +0.5. We can see from this position plot that during the early phases, while epsilon is high, the agent rarely reaches its position, and resides around -0.3 which is the location of the valley. Then, the agent begins to consistently reach the end goal, with approximately 1 in 6 episodes ending below -0.6. This indicates that the car was in the process of "rocking" on the left hill when it ran out of time. Towards the end of this plot we see a more consistent reaching of +0.5. With additional training time it is very likely that the number of failed episodes would continue to decrease until the model fully converges.

2 Double Deep Q-Learning

2.1 Deep Q-Learning Improvements

Double Deep Q-Learning (D2QN) is a modification of DQN which attempts to improve upon its performance. Specifically, D2QN is design to counteract the bias problem and the moving target problem present in Q-Learning. In Section 1.2 we discussed the moving target problem, and the traditional DQN solution to this problem of using a target network. In the DQN target network solution, a single network is copied, then a frozen copy of this network is used to determine the optimal action as well as the q-value for that action. This means that there is only ever one set weights determined, and that the selection of the best action and the determination of its q-value are coupled. D2QN avoids these issues by making use of two separate networks, which are randomly selected for training and which both contribute to the selection of the optimal action.

In D2QN we initialize two completely separate networks with random weights. Then at each step one of the two networks is selected for training, and is trained based upon the current state of the other network. Mathematically we write the D2QN training target for Q_1 as

$$r(s,a) + \gamma Q_2\big(s', \arg max_{a'}\big(Q_1(s',a')\big)\big)$$

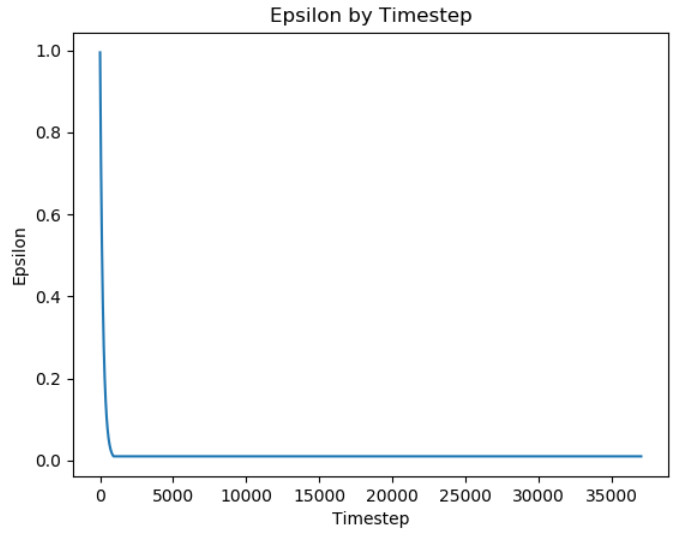
In this equation we see that the Q_1 network is used to select the best action, but the Q_2 network actually evaluates the Q-value of that state. Which of the two networks is trained at a given step is selected randomly with the opposite network always being used for evaluation. This helps to reduce bias by splitting early samples between the two networks, and ensuring that selection and evaluation are decoupled.

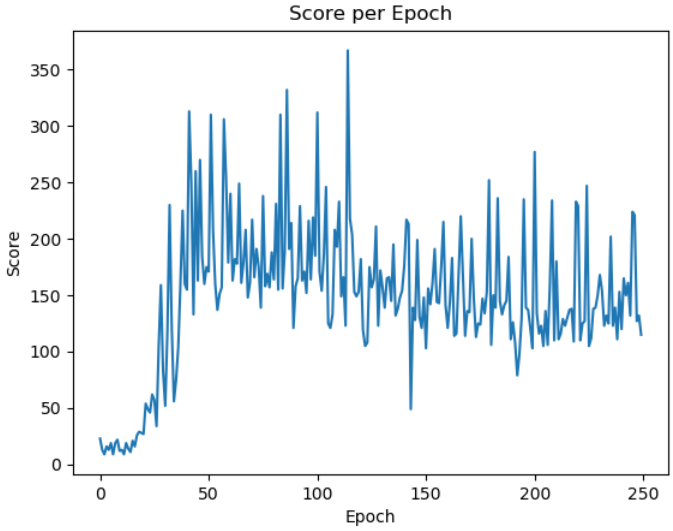
2.2 D2QN Results

The following results were obtained from running the implemented Double Deep Q-Learning agent on the environments described above. Due to limited computational availability and the extremely long training time of some models, data for 250 epochs will be shown for all cases. This should be sufficient to show the converging trends when trained, even if not all cases reach complete convergence.

2.2.1 D2QN CartPole Results

The same metric as described in Section 1.5.1 are shown below. In this case all parameters are the same as those used in the DQN trial, with only the addition of a second q-model and replacement of the target function.

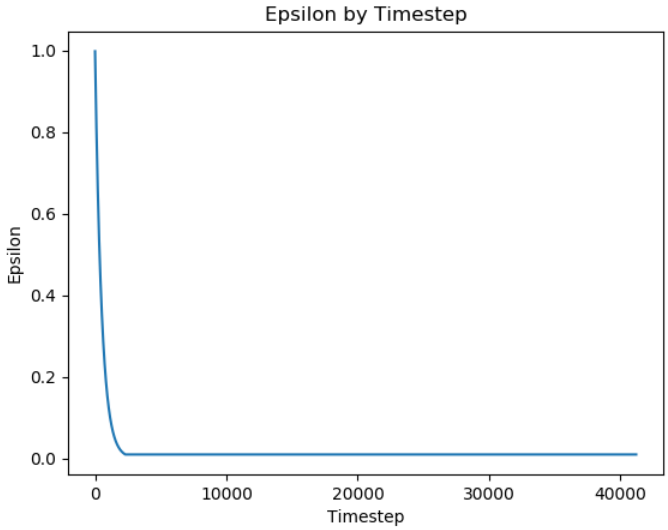


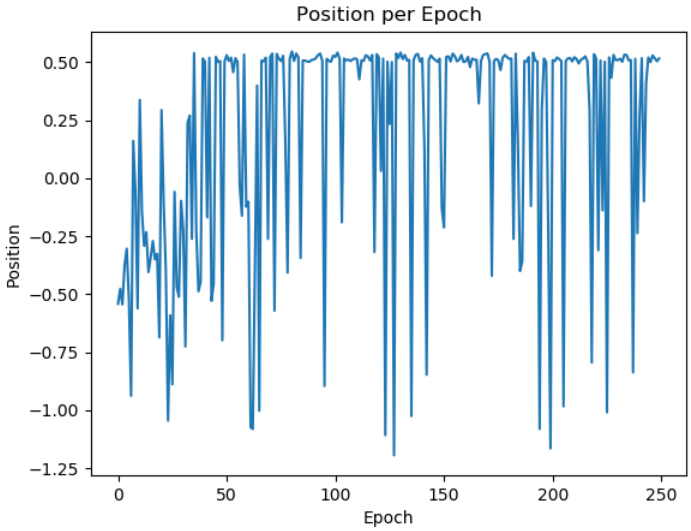


Comparing this plot to that of the equivalent DQN function we can see that the D2QN agent converges much more quickly. In the DQN agent it took approximately 150 before a significant increase in score we see, here we observe the same gains at around 25 epochs. This model actually appears to begin to overfit at around 125 epochs as seen by the slight downward trend. Reducing lower threshold of epsilon from 0.01 to 0.00 would also likely help to smooth out the performance of this model, as given the precarious balance of the pole in this environment, even a single failed movement can cause this agent to fail. Overall the D2QN agent functioned significantly better than the DQN agent on this task.

2.2.2 D2QN MountainCar Results

As in Section 1.5.2 metrics of epsilon and final car position are reported below. The same model structure and hyperparameter with the sole addition of a second model and replacement of the target function being used.





In the position plot above we can observe that the behavior of the D2QN agent is much the same as the DQN agent for this model, task, and training duration. The model achieves a moderately high rate of success after the 50^{th} epoch and then works to refine its methodology. It is likely that both algorithms require additional training time to properly converge. The additional training time needed for this task compared to the CartPole task is likely due to the larger separation between action and reward, as well as the larger number of nodes and therefore the much larger number of weights. The D2QN agent adds the additional wrinkle of doubling the number of weights to train by virtue of its two separate models. More training and hyperparamter tuning would be needed to fully converge this model to the desired level of performance.