Trevor McCants
Project 1 - Navigation
Udacity – Deep Reinforcement Learning

Navigation Report

Algorithm Description

A Deep Q Network (DQN) was used to solve this environment. This algorithm uses a neural network to estimate the action value function, Q. The optimal policy is obtained by selecting the action at each time step with the maximum value according to the Q function.

The specific implementation I used has an epsilon greedy policy meaning at each time step the agent has a probability ϵ of selecting a random action. As training occurs, this value is decreased until it reaches a minimum.

A replay buffer was also used to allow the agent the train on uncorrelated experiences each epoch.

A local and target network were used to stabilize training with a soft update TAU each time step. The Temporal Difference (TD) method was used to calculate Q along with the target network. This approach increases bias but decreases noise allowing the network to train more reliably.

Hyperparameters

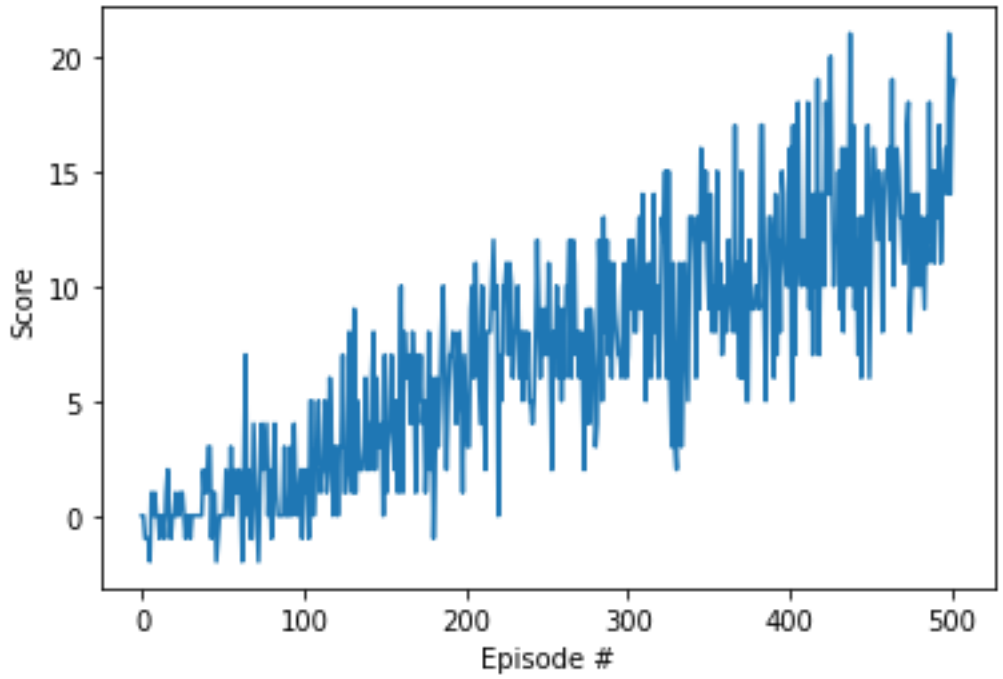
The following hyperparameters were used in this solution:

- BUFFER_SIZE = int(1e5) – Size of the replay buffer
- BATCH_SIZE = 64 – Number of experiences used in each training epoch
- GAMMA = 0.99 – Discount factor on rewards
- TAU = 1e-3 – Soft update interpolation parameter
- LR = 5e-4 – Learning rate for Q network
- UPDATE_EVERY = 4 – Number of time steps to wait before updating the Q network
- eps_start = 1.0 – Starting value of epsilon
- eps_end = 0.01 – Ending value of epsilon
- eps_decay = 0.995 – Decay rate of epsilon (eps(t+1) = eps(t)*eps_decay)

The Q network has 4 layers (1 input, 1 output, and 2 hidden). The input layer is the state size, the output layer is the action size, and the hidden layers are all 64 nodes. Each layer is linear with a relu activation function except the output layer which has no activation function.

Results

The agent took a total of 402 episodes to train. The score per episode can be seen in the plot below:



The final average reward over 100 episodes was 13.09.

Future Work

This problem may be able to be solved more effectively using:

- Prioritized Experience Replay – Assign each experience an importance level based on its TD error then sample from the replay buffer non-uniformly where more important experiences are used more often.
- N step bootstrapping – Instead of using pure TD estimation or Monte Carlo estimation for the Q function, use a mixture of the two where n true rewards are used to compute the value. This increases noise but reduces bias.
- Double DQN – Instead of always using the max Q value to compute the estimated rewards for a new experience, use a second Q network (usually the same one with old weights) to determine the optimal action then query the primary Q network using that action. This ensures you don't propagate forward large rewards that were obtained by random chance.
- Dueling DQN – Make the Q network directly predict the state and advantage values separately then use both to compute Q.