Learning Algorithm

The Deep-Q Learning algorithm was used on this challenge, modeling the same approach used in the Lunar Lander assignment. The hyperparameters were not (ultimately) changed beyond the defaults, as I was able to get solid results with the first three Pytorch neural network (NN) architectures I tried, providing the convenience of not having to permute them unnecessarily. After getting reasonable results, I did modify the hyperparameters a bit, but with minimal improvement to the training results (and often significant degradation) indicating that the NN architecture was resilient enough to provide reasonable results even with subtle changes to the hyperparameters.

The first NN architecture I used only had two hidden layers, with the input and output mapping of:
<state_size(37)> → 512 → 128 → 32 → <action_size(4)>. This resulted in hitting the objective values in 524 total episodes (of which the last 100 hit the goal).

I tried several other NN architectures – both with additional hidden layers as well as different architectures with the same (two) hidden layers:
<state_size(37)> → 256 → 64 → 16 → <action_size(4)>. This resulted in hitting the objective values in 501 total episodes (of which the last 100 hit the goal).
<state_size(37)> → 128 → 32 → 8 → <action_size(4)>. This resulted in hitting the objective values in 520 total episodes (of which the last 100 hit the goal).

The hyperparameters used in the Deep-Q implementation were:

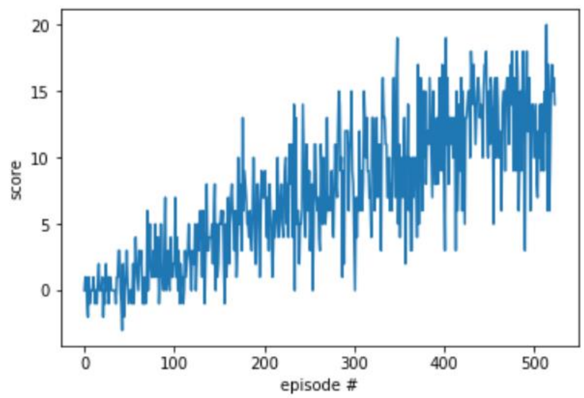
```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

Results and Plot of Rewards

The agent trained more quickly than I expected, as my (somewhat arbitrary) initial selection of supporting NN architecture. The learning algorithm ran a total of 524 times, achieving a score of 13.07 over the last 100 for the first architecture. This was done with the architecture of:

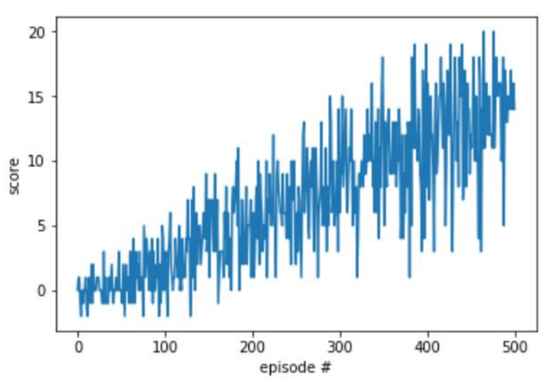
<state_size(37)> → 512 → 128 → 32 → <action_size(4)>

```
Episode 100     Average Score: 0.92
Episode 200     Average Score: 4.12
Episode 300     Average Score: 7.30
Episode 400     Average Score: 9.75
Episode 500     Average Score: 12.72
Episode 524     Average Score: 13.07
Environment solved in 424 episodes!     Average Score: 13.07
```



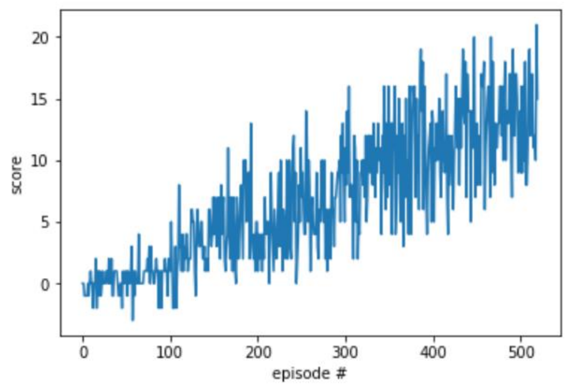
Here is the plot with the architecture of: *<state_size(37)> → 256 → 64 → 16 → <action_size(4)>*

```
Episode 100     Average Score: 0.68
Episode 200     Average Score: 3.74
Episode 300     Average Score: 6.74
Episode 400     Average Score: 10.08
Episode 500     Average Score: 12.96
Episode 501     Average Score: 13.02
Environment solved in 401 episodes!     Average Score: 13.02
```



And here is the plot with the architecture of: *<state_size(37)> → 128 → 32 → 8 → <action_size(4)>*

```
Episode 100     Average Score: 0.37
Episode 200     Average Score: 3.94
Episode 300     Average Score: 5.70
Episode 400     Average Score: 9.96
Episode 500     Average Score: 12.45
Episode 520     Average Score: 13.02
Environment solved in 420 episodes!     Average Score: 13.02
```



Ideas for Future Work

The agent learned in a reasonable amount of time (even on CPU hardware) with the Deep-Q algorithm, even with all of its (well known and documented) limitations. I would expect that many of the standard approaches to improving on the limitations of Deep-Q Networks such as prioritized experience replay, double DQN, dueling DQN, Rainbow, or a combination thereof, would allow the agent to learn more quickly and predictably.