The Report

Description of the implementation;

Jupyter notebook contains the implementation to run the training of the agent on to the environment by using the following files;
model.py file has the neural network architect implementation that agent uses to feed the state input and get the predicted output as Q-value for the [state,action] pairs.
agent.py contains the Q-learning with fixed Q-targets method to interact with the environment and learn from the environment. agent.py also has ReplayBuffer class to be used by the Agent class

The report clearly describes the learning algorithm, along with the chosen hyperparameters. It also describes the model architectures for any neural networks.

Deep Q-learning has been used which takes as input state and gives the output as Q-value for the (state, predicted action) pairs. Q-learning makes the use of epsilon greedy policy selection to explore the environment or exploit the learned Q-values by selecting the actions by using the argmax-Q value. The epsilon probability is decreasing suggesting that the agent exploits more in the later episodes. During the inference, epsilon value needs to be zero because the agent is not supposed to explore the environment space. DQN is meant to solve discrete action space that's why it is suitable for this project.

```
With 3 hidden layers, the agent solved the environment in 534 episodes!
      with the average Score: 14.00
With 4 hidden layers, the agent solved the environment in 415 episodes!
      with the average Score: 14.01
```

ReLU activation has been used with 64 nodes in each of the hidden layers. The last layer is a linear layer (logit) that gives out the predicted value as action values.

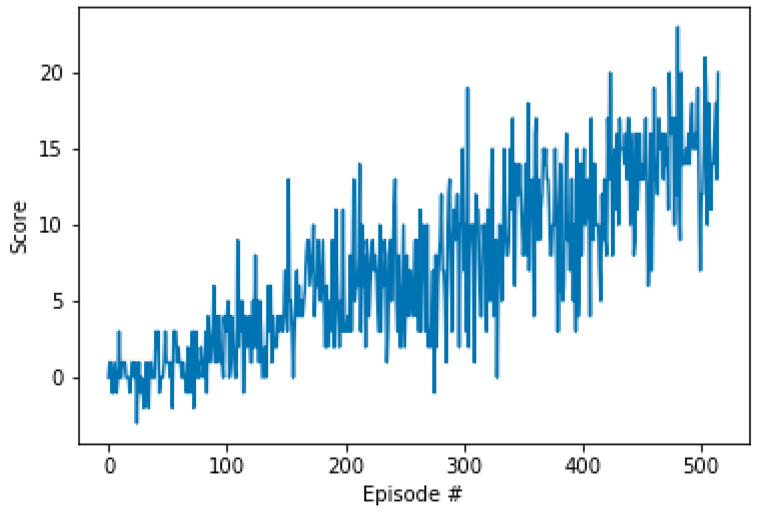
Other Hyper-parameters;

```
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
n_episodes=800          # maximum number of training episodes
max_t=1000              # maximum number of timesteps per episode
eps_start=1.0           # starting value of epsilon, for epsilon-greedy
action selection

eps_end=0.01           # minimum value of epsilon
eps_decay=0.995        # multiplicative factor (per episode) for
decreasing epsilon
```

A plot of rewards per episode is included to illustrate that the agent is able to receive an average reward (over 100 episodes) of at least +13. The submission reports the number of episodes needed to solve the environment.

```
Episode 100     Average Score: 0.86
Episode 200     Average Score: 4.32
Episode 300     Average Score: 6.64
Episode 400     Average Score: 9.78
Episode 500     Average Score: 13.37
Episode 515     Average Score: 14.01
Environment solved in 415 episodes!     Average Score: 14.01
```



The submission has concrete future ideas for improving the agent's performance.

1.
Prioritized experience replay can be used for the improvement. It is a framework for prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently.
Reference https://arxiv.org/abs/1511.05952
This github repo provides a nice way to implement the technique
https://gist.github.com/avalcarce/9ec53aadd17d2a125e53d105ff84f9ed

2.
Input state space can be the image pixel values and so the convolution layers will be used in the model architecture. In this way, the reinforcement learning agent gets to learn about the task and the environment through perception. Images shape can be downsized to (32, 32, 1) as gray-scaled image. I'll also crop the upper-side of the images which represents the sky so that the agent only gets to receive the relevant state information.

3.
Trying out different architect settings like adding batch normalization, dropout, trying different weight initialization functions, and trying different optimizer functions can help to improve the performance of the agent.