Navigation Report

<u>Files</u>
'navigation.ipynb': contains the code to train the agent
'model.py': contains the neural network architecture the agent uses to feed the state input and get the predicted Q value for a state, action pair
'agent.py': contains the Q learning with fixed Q targets method to interact and learn from environment and Replay buffer class

<u>Learning Algorithm</u>

The project's discrete action space makes it suitable for Deep Q Learning.
For this project, 2 different vanilla Deep Q Learning agents (3 hidden layers & 4 hidden layers) were trained.

Addressing the network architecture $_{\text{3HiddenLayers}}$:

The input size is 1x37 states
First hidden layer is 64 fully connect neurons
Second hidden layer is 64 fully connect neurons
Final layer has 4 possible actions

Addressing Q learning agent's hyperparameters:

```
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
```

Training the agent:

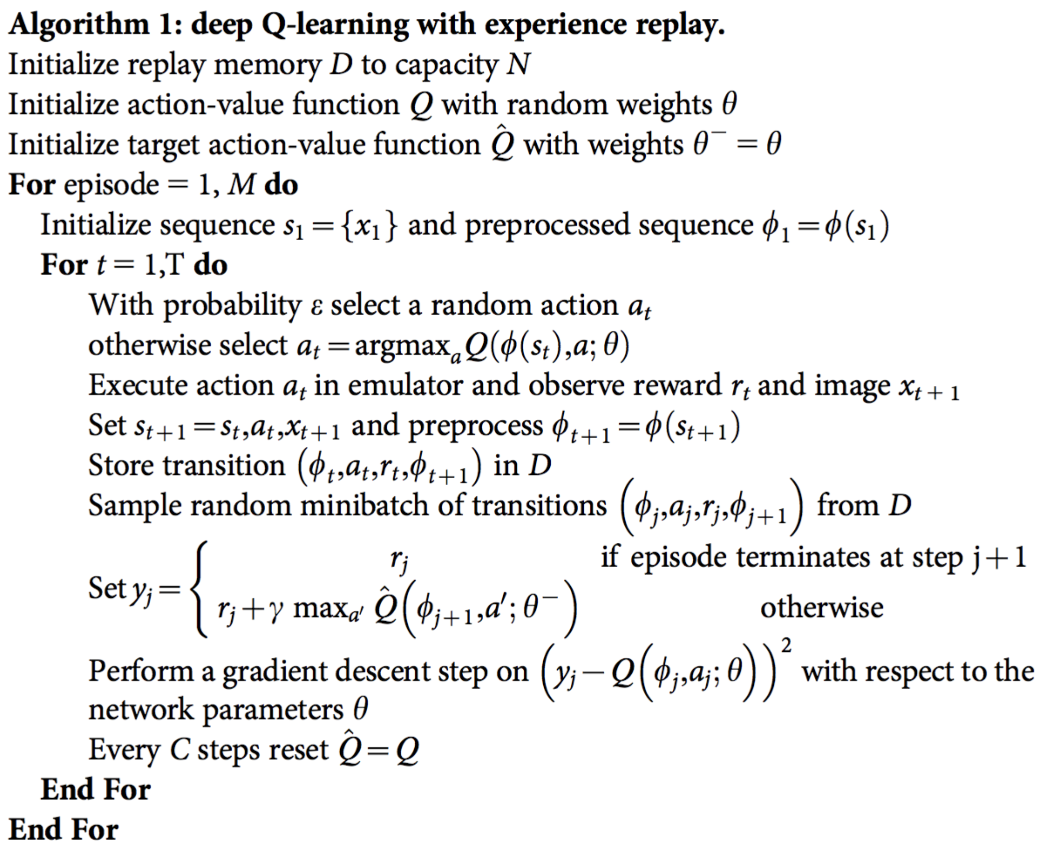
The agent is trained in the "navigation.ipynb" file.
The Q learning function has the following arguments: # of episodes, epsilon start, epsilon end, and epsilon decay rate
And the following variables: score of each episode, a container for the score over the last 100 episodes, initial epsilon value, a reset for the environment, and a variable to get the initial state info.
For each episode:
- environment and the score are reset
- obtain current state

During an episode:
- agent takes an action which is either chosen at random with probability
- epsilon, or by the greedy policy as learned by the online network if a sufficient number of episodes have passed

- send the action to the environment and obtain the next state
- get a reward for taking the previous action
- check if the episode is finished
- move the agent forward one step and have the agent store the state, action, reward, next state, done tuple in memory for use in learning (note: only up to the max mem size set in agent's parameters)
- update current state to be the next state
- update the score for the most recent reward
- check if the episode has finished
- update the total scores for the episode and the rolling episode window
- decay epsilon by the decay parameter
- move to next episode if the environment has not been solved (note: if the environment is solved, or the total number of episodes is reached -> end training)

Learning Info:

Below the DQN algorithm is presented with some general learning info. The pseudocode for the DQN algorithm has been copied from: https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

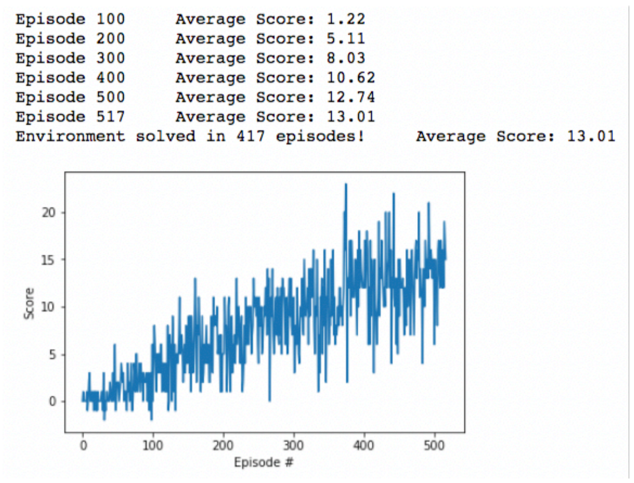


```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

- The agent will every 4 steps randomly sample a mini-batch of tuples from the S, A, R, S, A tuples stored in memory to learn from the actions it took from state observations.
- The agent uses Q values generated from the target network and expected Q values from the

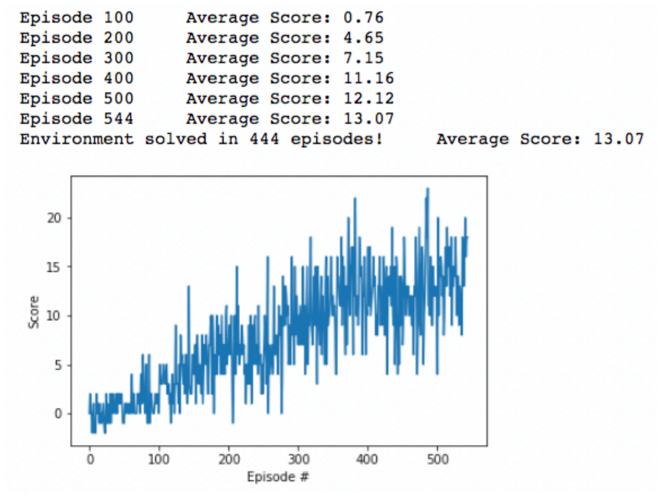
online network as input to the Bellman equation.
- After many episodes, the memories of S, A, R, S, A sequences an agent has observed is used to form an optimal policy (this is encoded in the weights of the online and target neural networks)

Plot of Rewards
3 hidden layers (checkpoint3L.pth)



```
Episode 100     Average Score: 1.22
Episode 200     Average Score: 5.11
Episode 300     Average Score: 8.03
Episode 400     Average Score: 10.62
Episode 500     Average Score: 12.74
Episode 517     Average Score: 13.01
Environment solved in 417 episodes!     Average Score: 13.01
```

4 hidden layers (checkpoint4L.pth):



```
Episode 100     Average Score: 0.76
Episode 200     Average Score: 4.65
Episode 300     Average Score: 7.15
Episode 400     Average Score: 11.16
Episode 500     Average Score: 12.12
Episode 544     Average Score: 13.07
Environment solved in 444 episodes!     Average Score: 13.07
```

Ideas for Improvement

Prioritized experience replay:
- a technique for prioritizing experiences that replays more important transitions more frequently (it does this by boosting the sampling of experiences which are likely to lead to the greatest improvements for the error term of the network)

Try different DQNs
- double DQN, dueling double DQN, different architecture for current neural network (convolution layer, dropout, batch normalization, etc.)

Dyna-Q:

- a technique that blends Q-planning and Q-learning (the agent is able to build a model of the environment as it learns; this reduces the number interactions the agent needs to have with the environment before learning a near optimal policy.