Deep Reinforcement Learning Nanodegree
Project 1 – Navigation

Description
We implement DQN from Deepmind's paper (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf) to tackle this problem. In essence, we use Q-learning to perform online reinforcement learning and use a neural network as function approximator to estimate state action value. In Q-learning, instead of waiting for entire episode to finish to get the total reward hence the Q value, we calculate the Q value using the estimation of Q-value from next state.

The neural network used in this project is rather simple network with following architecture where the input state has dimension of 37 and output is dimension of 4 which is the size of action space:
- fully connected layer (37x64 weights, Relu)
- fully connected layer (64x64 weights, Relu)
- fully connected layer (64x4 weights)

Two identical neural networks are used with different weights where one lag behind another. We implement experience replay by storing the experience in buffer. During training, we retrive the experience randomly, and use the following loss function (from Deepmind's paper) to to perform gradient descent optimization (we use the Adam variant)

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathrm{U}(D)} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i) \right)^2 \right]$$

where r is the reward, gamma is discount factor, theta_i is the neural network weight while theta_i_minus is the target neural network weights that only updated after a number of steps. In other words, in each backprop, theta_i will be updated but not theta_i_minus.

We use epsilon-greedy to select the policy. In other words, for probablity of 1-epsilon, we select the action that has highest score from neural network, with probability of epsilon, we select action from the rest of the state with uniform random sampling.

Hyper parameters
Hyper parameters include neural network layers and neuron. The more layers and neurons then the more accurate is the prediction but will result in lower inference and training speed. The following are other hyperparmaters for the neural networks

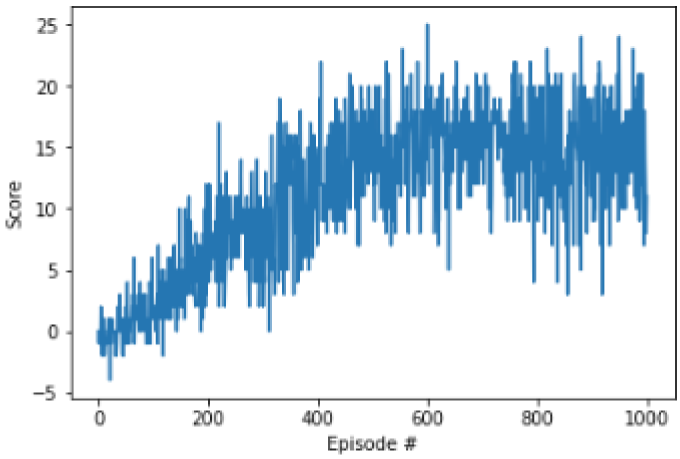
```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size, the bigger the better, limited by GPU size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4     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

For Q learning
    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

The code are:
Navigation.ipynb – jupyter notebook to run the training
dqn_agent.py – DQN agent
model.py – contain the neural network definition
model.pth – saved neural network model

Result
The desired score of +13 is achieved after 500 episode. The chart is shown in jupyter notebook and below.

```
Episode 100     Average Score: 0.60
Episode 200     Average Score: 4.02
Episode 300     Average Score: 7.93
Episode 400     Average Score: 10.23
Episode 500     Average Score: 13.45
Episode 600     Average Score: 14.75
Episode 700     Average Score: 15.53
Episode 800     Average Score: 15.59
Episode 900     Average Score: 15.06
Episode 1000    Average Score: 14.75
```



Ideas to improving agent's performance
Some approaches:
1. Double DQN to solve overestimation of Q-value by using two set of network weights that must agree on the best action.
2. Priotized Experience Replay. Instead of sampling experience replay randomly, we assign higher priority to experience that we want our algorithm to learn more from.
3. Dueling DQN. The network estimates two things – state values and advantage value which are combined to provude Q-values.