Deep Q-learning for Navigation

In this project an agent learns to navigate and collect bananas. For this task, a deep learning model learns policy from environment features obtained from the simulator. The model is a feed forward neural network and we train the network using a Q-learning inspired approach. The input to the neural network is the environment features and outputs directional actions. As an addition, we introduce a tweak which allows to the Q-learning inspired learning in order to reduce overestimation of state-action values. The new tweak which is called double Q-learning helps to estimate using Q values using two separate estimators.

Deep Q-learning (DQN)applies fundamental reinforcement learning mechanics. The future rewards are discounted per time-step and the future discounted reward R_t at time t is:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

The optimal state-action value helps us to find the optimal strategy. After taking a series of actions **a** and observing states **s** the optimal state-action value is:

$$Q^*(s,a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$$

where the policy Π maps sequences to actions. The state-action value function concurs with Bellman equation which we display below states that given a s, the optimal policy can be reached by choosing the action a' that will result in state s' that will yield the maximum expected reward.

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') \middle| s, a\right]$$

With problems with very large state space, finding the optimal Q values requires iterating over the entire state space at each update. Therefore, researchers have suggested approximation methods. Neural networks are universal function approximators, therefore they are a good fit for the problem at hand. Thus, neural networks are used in order to estimate the action-value function.

$$Q(s,a;\theta) \approx Q^*(s,a)$$

The optimal state-action value is replaced with a neural network with weights that are represented with Θ. The neural network can be trained by minimizing the error between estimated state-action value and the observed value which can be written as:

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right]$$

where

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$$

represents the target over at the previous state i-1 over sequences **s** and actions **a.**

The gradient of the loss function with respect to the weight results in:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

The main difference between supervised learning is that the weights of the target are not fixed. They are also evolving during training and depends on the quality of the exploration of the agent.

The behavior distribution is selected by epsilon greedy strategy with a decaying characteristics. Moreover, the algorithm uses a structure called replay buffer. The replay buffer stores episodes during agents exploration phase. During training, random sequences are obtained from the replay buffer. This is helpful for multiple reasons. First reason is that it is possible to experience multiple sequences for training. Secondly, the training batches are formed by random sequences as opposed to successive explorations. This randomness improves training of the neural networks. In addition to the replay buffer, the authors present a second neural network which is called the target network. The target network is used to estimate the target values. The target neural network weights are only updated with with the Q-network parameters at certain intervals and the weights are fixed fixed between individual updates of the Q-learning network.
The overall algorithm is presented below:

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
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 ε 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 $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

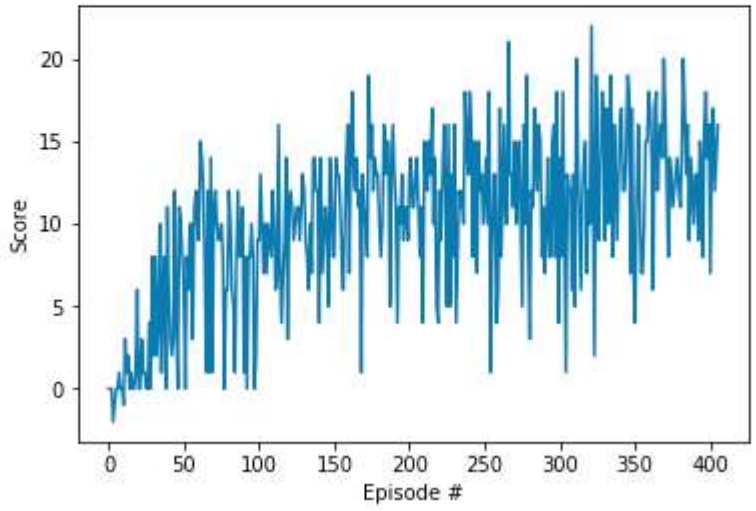
The hyperparameters that we have used to solve the algorithm is given below.

Maximum number of episodes: 2000
Maximum number of timesteps per episode: 300 (Environment default)
eps_start, starting value of epsilon: 0.003
eps_end, end value of epsilon: 0.001
eps_decay, multiplicative factor (per episode) for decreasing epsilon: 0.9
Replay Buffer size: 1e10
Minibatch size = 32
Discount factor (gamma) = 0.99 # discount factor
Learning Rate = 2e-4
Target network update steps: 2

With the given hyperparameters, the defined goal is to get an average score of +13 over 100 consecutive episodes. We are able to achieve the goal in 406 episodes. Below we present the average score vs episode.

```
Episode 100     Average Score: 5.18
Episode 200     Average Score: 10.66
Episode 300     Average Score: 11.63
Episode 400     Average Score: 12.83

Environment solved in 406 episodes!     Average Score: 13.01
```



In future, the algorithm could be enhanced to update the Q estimates after multiple steps. Rather than taking a single action and using the Q-estimate for the Bellman update, the agent can take multiple steps and combine multiple rewards with Q-estimates. In this scenario, the challenge could be to make the replay memory and the multi-step updates work together.

Double Deep Q-learning

Double Q-learning introduces a simple update to prevent early over estimations. This is because for both target value selection and estimations, same neural network is used. Hence, the update rule uses the same values both to select and to evaluate an action. Thus Q-learning target selection is switched from:

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

to the updated version where the target value estimate is selected from the target neural network.

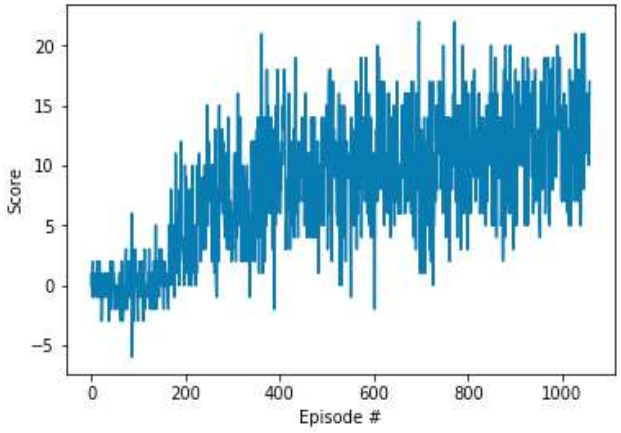
$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)$$

This shows that the selection of the action is using the Q-learning network, however we use the target network to evaluate the value of the policy for the error calculation. Below shows the rewards per episode. The hyperparameters below are the ones that are different than the Q-learning.

Target network update steps: 5

```
Episode 100     Average Score: -0.23
Episode 200     Average Score: 1.54
Episode 300     Average Score: 6.44
Episode 400     Average Score: 7.67
Episode 500     Average Score: 9.23
Episode 600     Average Score: 10.11
Episode 700     Average Score: 10.40
Episode 800     Average Score: 10.53
Episode 900     Average Score: 11.62
Episode 1000    Average Score: 12.65

Environment solved in 1059 episodes!    Average Score: 13.01
```



Double Q-learning starts to obtain higher rewards with less fluctuations earlier than the Q-learning. However, it takes 1059 episodes to reach the desired goal. A thorough hyperparameter search can decrease the number of episodes.