Deep Reinforcement Learning- Continuous Control

Udacity- Project 1

Unity Banana Collector

Introduction

In the report for the first project required for partial fulfillment of requirements pertaining to Udacity's Deep Reinforcement Learning Nanodegree, different Deep Q Learning Agents were implemented to solve Unity's Banana Collector Environment. The grading rubric calls for the report to delve into Learning Algorithms, Reward Plots and Ideas for Future work. Details about implementation and the environment could be found in the 'README.md' file of the home repository.

Algorithms

Different deep learning implementations of the Q-Learning algorithm were used to solve the environment. Deep Q-Networks are neural networks implementation of the Q functions that calculate Q values for given action-value pairs. The project explores Deep Q Learning, Double Q Learning, Dueling Q Learning and Dueling Double Q Learning algorithms to solve the environment.

Q-Learning

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite Markov decision process, Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that returns the reward used to provide reinforcement and can be said to stand for the "quality" of an action taken in a given state.

The algorithm has a function that calculates the quality of a state-action combination,

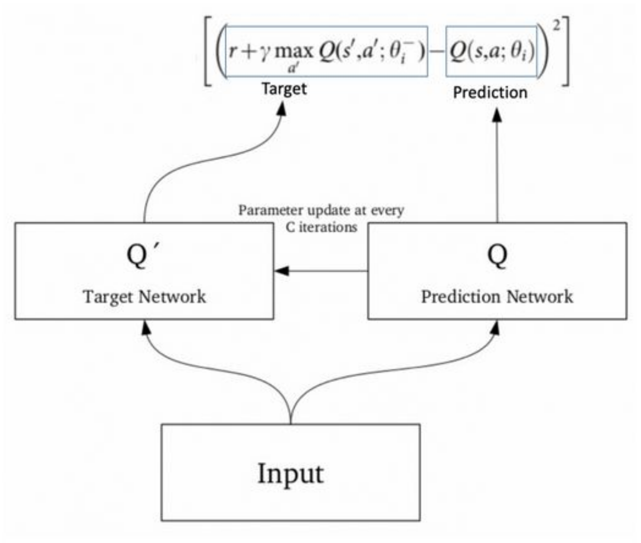
$$Q : S \times A \to \mathbb{R}$$

Before learning begins, Q values are initialized to a possibly arbitrary fixed value. Then, at each time t, the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} and Q is updated. The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information,

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}.$$

Deep Q-Learning

In deep Q-learning, a neural network is used to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output.The next action is determined by the maximum output of the Q-network. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. The target or actual value are unknown and a two network approach is utilized. Since the same network architecture is used to calculate the predicted value and the target value, a separate network is used to estimate the target. This target network has the same architecture as the function approximator but with frozen parameters. For every Cth iteration, the parameters from the prediction network are copied to the target network. This leads to more stable training.


$\left[\left(\boxed{r+\gamma \max_{a'} Q(s',a';\theta_i^-)} - \boxed{Q(s,a;\theta_i)}\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

Double Deep Q-Learning

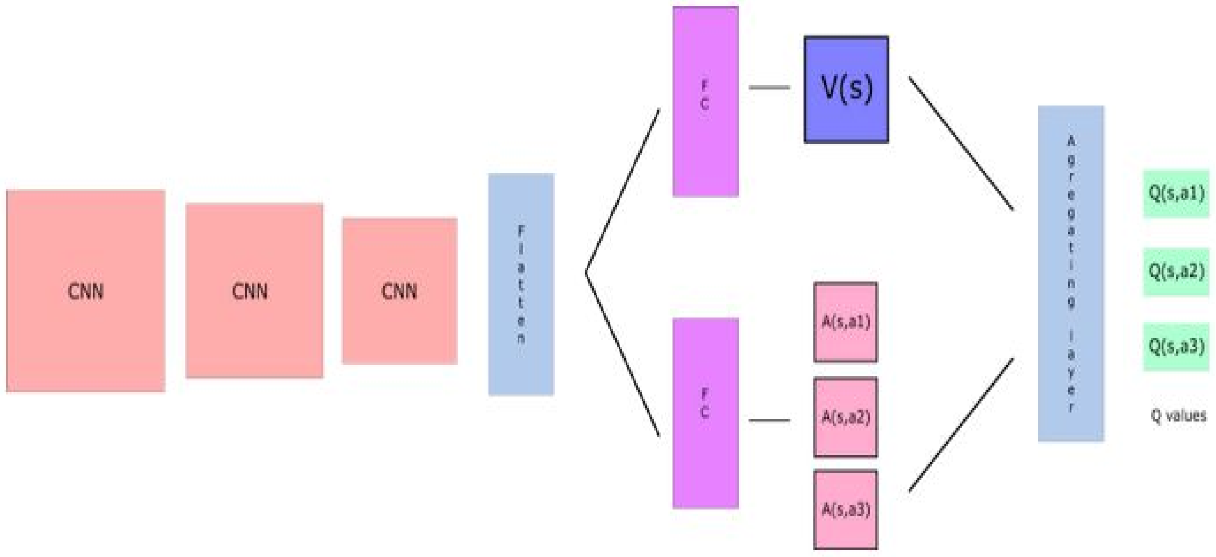
Double Q-learning is used to reduce overestimations by decomposing the *max* operation in the target into action selection and action evaluation. In the vanilla implementation, the action selection and action evaluation are coupled. Target-Network is used to select the action and at the same time to estimate the quality of the action. The Target-Network is used to calculate *Q(s,* *a_i)* for each possible action *a_i* in state *s*. The greedy policy is used to decide upon the action *a_i* to select. In Double Q-Learning the TD-Target looks like

$$y_i^{\textbf{DoubleQ}} = \mathbb{E}_{a' \sim \mu}\left[r + \gamma Q(s', \arg\max_a Q(s', a; \theta_i); \theta_{i-1}) | S_t = s, A_t = a \right]$$

While the Target-Network with parameters ***θ(i-1)*** evaluates the quality of the action, the action itself is determined by the Q-Network that has parameters ***θ(i).*** This procedure is in contrast to the vanilla implementation of Deep Q-Learning where the Target-Network was responsible for action selection and evaluation.

Dueling Deep Q-Learning

Unlike the standard DQN architecture, the network is split into two separate streams, one for estimating the state-value and the other for estimating state-dependent action advantages. After the two streams, the last module of the network combines the state-value and advantage outputs.



$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$$

Implementation Details

The Q-Network and the Dueling Q-Network have similar architecture with two hidden layers and an output layer. Input batch has dimension equal to the size of the state space. ReLu activation is applied to both the hidden layers. The output layer for the Q-Network has a size equal to the size of the action space. For the Dueling Q-Network, the output from the second hidden layer is passed to two different output layers, one for calculating the state value and the other for calculating action value. The output from the two layers are added to calculate the state-action value. The following hyperparameters were used for the agents,

1. Batch Size = 64
2. Buffer size = 100000
3. Gamma = 0.99
4. Tau = 0.001
5. Learning Rate = 0.0005

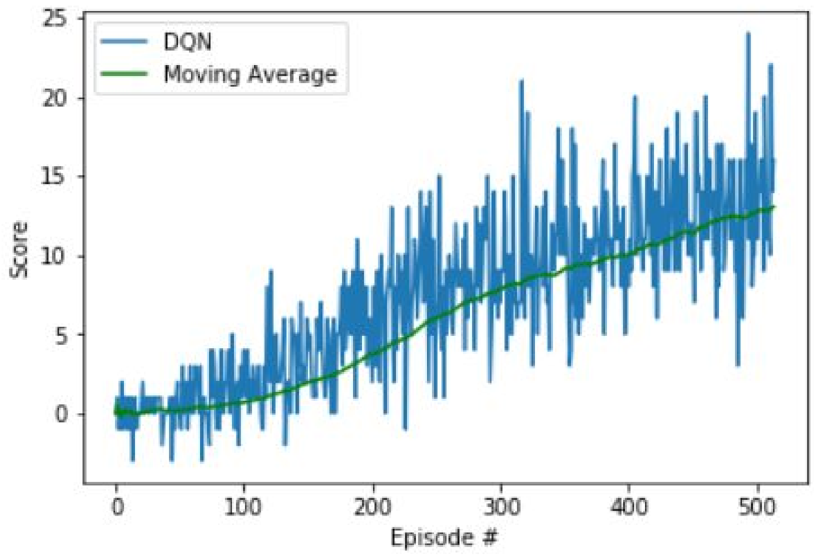
Updates were made every 4 timesteps. Other hyperparameters and more specific implementation details could be found in the ‘Navigation.ipynb’ notebook.

Reward Plots

The project has four different implementations including Deep Q-Network, Double Deep Q-Network, Dueling Q-Network, and Dueling Double Q-Network. The average reward for hundred episodes and the moving average is plotted on the same plot and the training terminates when the average reward is more than or equal to thirteen.

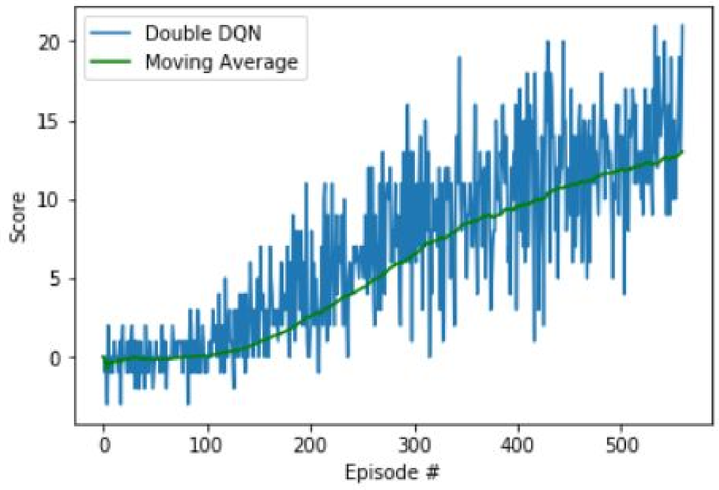
Deep Q-Network

```
Episode 100	Average Score: 0.67
Episode 200	Average Score: 3.71
Episode 300	Average Score: 7.78
Episode 400	Average Score: 10.03
Episode 500	Average Score: 12.71
Episode 514	Average Score: 13.03
Environment solved in 414 episodes!	Average Score: 13.03
```



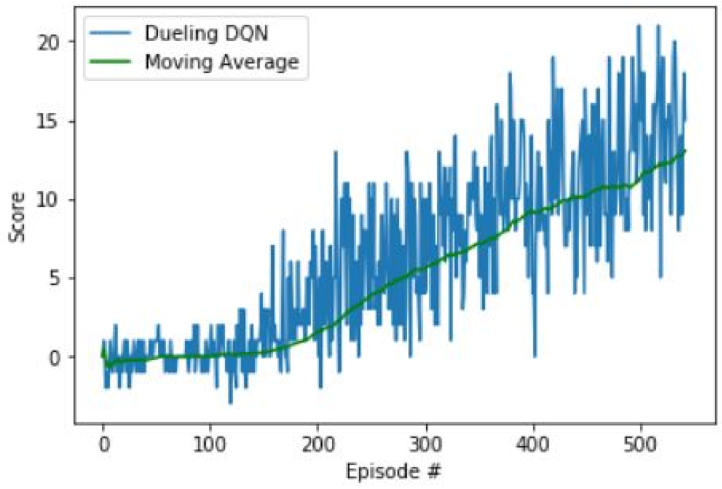
Double Deep Q-Network

```
Episode 100     Average Score: 0.08
Episode 200     Average Score: 2.52
Episode 300     Average Score: 6.49
Episode 400     Average Score: 9.42
Episode 500     Average Score: 11.89
Episode 560     Average Score: 13.01
Environment solved in 460 episodes!     Average Score: 13.01
```



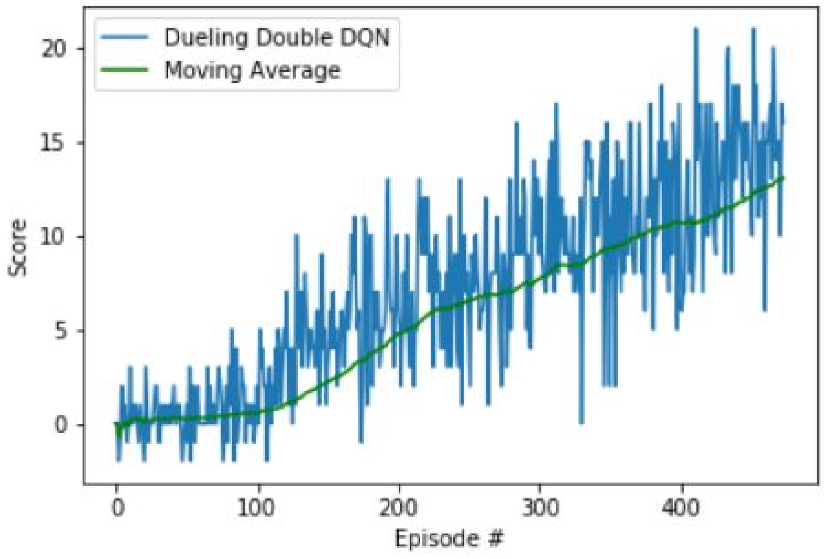
Dueling Deep-Q Network

```
Episode 100     Average Score: -0.02
Episode 200     Average Score: 1.531
Episode 300     Average Score: 5.58
Episode 400     Average Score: 9.21
Episode 500     Average Score: 11.18
Episode 543     Average Score: 13.05
Environment solved in 443 episodes!     Average Score: 13.05
```



Dueling Double Deep-Q Network

```
Episode 100     Average Score: 0.53
Episode 200     Average Score: 4.74
Episode 300     Average Score: 7.64
Episode 400     Average Score: 10.71
Episode 474     Average Score: 13.06
Environment solved in 374 episodes!     Average Score: 13.06
```



Ideas for Future work

1. Prioritization for replay buffer
2. Test shared network between agents
3. Noisy DQN
4. Hyper parameter optimization
5. Experiment with different network architecture and activation layers