Report Project1: Navigation

1. Introduction:

Reinforcement Learning problems are based on a framework called Markov Decision Processes (MDP). An MDP is easily described using a State Machine structure. An agent is present in an environment. The agent is present in state (s_i), takes an action (a_i), goes to a new state (s_{i+1}) and as a result gets a reward (r_i) from the environment. Environments are classified into episodic and continuous. Episodic tasks are ones that have a terminal state, i.e., the task ends when an agent reaches the terminal state. Video games are great examples of episodic tasks. Whereas continuous tasks don't have a terminal state. The task of an agent whose job is the collect trash bottles in a designated area can be viewed as continuous since it's an ongoing task. In this project we work on the Banana Environment provided by Unity Inc. This is an example of an episodic task.

The algorithm that is used to solve the problem is Deep Q-Network (DQN) and Double DQN, a variant of DQN. Simply put, DQN substitutes a Neural network to approximate the Q value of each state [1]. Q-Learning starts with arbitrary Q values and iteratively updates the Q values. An epsilon-greedy policy is used to select an action (a_i), from a current state (s_i), which results in the agent into a new state (s_{i+1}). To update the current state Q-value $Q(s_i, a_i)$, a new action is selected to compute expected return of the new state and this is used to compute the loss. The loss is in turn used to update the Q-value as in below equation:

$$\mathrm{Q}(\mathrm{s_t}, \mathrm{a_t}) \leftarrow Q(s_t, a_t) + \alpha\,(\mathrm{R}_{t+1} + \gamma\, max_a\, Q(s_{t+1}, a) - Q(s_t, a_t))$$

Where α is the learning rate of Q-value update. When run for enough number of iterations Q-learning converges and hence finds an optimal policy for the agent [1].

2. DQN Algorithm:

As mentioned above, DQN algorithm uses a Neural Network to approximate the Q-value function. The neural network acts as a mechanism to approximate a continuous state space. Q-Learning is prone to divergence easily, since small updates to Q values may result in different policies [2]. The paper [2], creates a novel approach to using a DQN, a neural network that is able to approximate the Q-values. Following are the important points to be noted:

- Use of Replay Memory: There is a natural temporal correlation when exploring the environment. So, it does not necessarily result in a good learning, when we use such transition tuples to learn the Q values. So a replay buffer, for example 1 million entries, is used. While the agent is exploring the environment, the transition tuples are stored in the memory. A mini-batch is sampled, of size for example 128, and used to pass thru the neural net for training. This way there is some inherent. Randomness that's associated with transition tuples, which will remove the temporal correlation between the transitions.
- Target Q Network: In Q-Learning we need to calculate a target Q value for the next state. This is used to compute the TD error term. The error term is used to update the Q-value.

Since we NN is used to approximate the Q value of the current state action pair, it is basically learning its weights to get the correct Q value. Hence, it's not a good idea to use the same NN to calculate the target Q value. This can potentially lead to oscillations in Q values causing the algorithm to diverge. A target NN is used to calculate the target Q-values and this target NN updates its weights, once every "C" no. of steps. A soft update is performed to learn the values of the target Q neural network, as in the below equation:

$$\theta^{\text{target}} \leftarrow \tau\theta^{\text{local}} + (1-\tau)\theta^{\text{target}}$$

- Double DQN: In Q-Learning we need to perform an argmax operation to get the max Q value for the next state, in order to compute the TD error. This could be prone to error, especially during the early phases of learning. To avoid this, Double Q-learning performs argmax to select the action using the local NN and the Q-value is looked up using the target NN.

The below sections define the target and local NN Model architecture and the hyper parameters selected for solving the Banana environment's task. Note that the target and local NN have the same architecture.

2.1 DQN Neural Network Architecture:

The input layer takes the state observation, 37 units. There are 2 hidden layers, 1st hidden layer has 74 units and 2nd hidden layer has 74 units. And the output layer represents the continuous action vector, with 4 units, representing the Q-values of for all actions with respect to the observed state. ReLu activation was used after each hidden layer.

2.2 Hyperparameters:

The following were the optimal tuned hyperparameter settings:

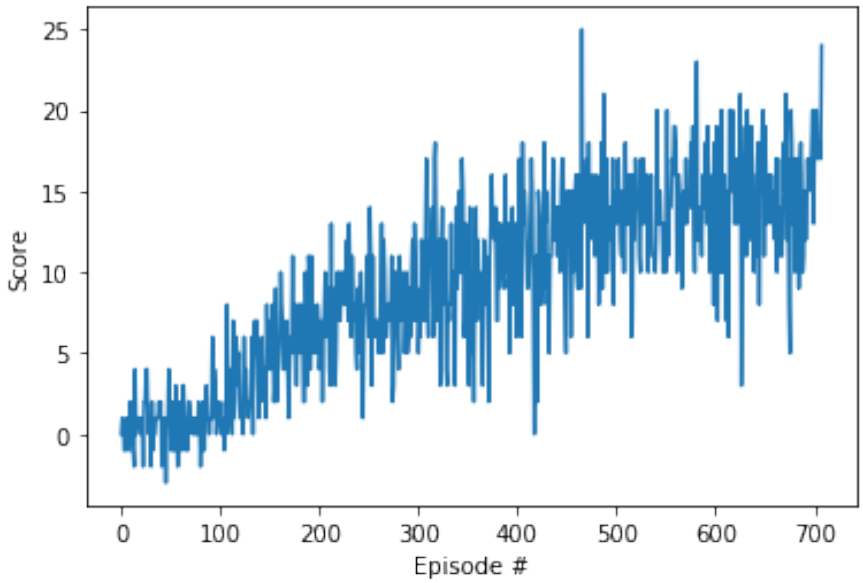
Hyperparameter	**Value**
Replay Buffer Size	1e5
Batch Size	64
Gamma	0.99
Tau (Soft Update rate)	1e-3
Actor Learning Rate	5e-4
Target Network Update Every "C" Iterations	4

2.3 Pseudocode:

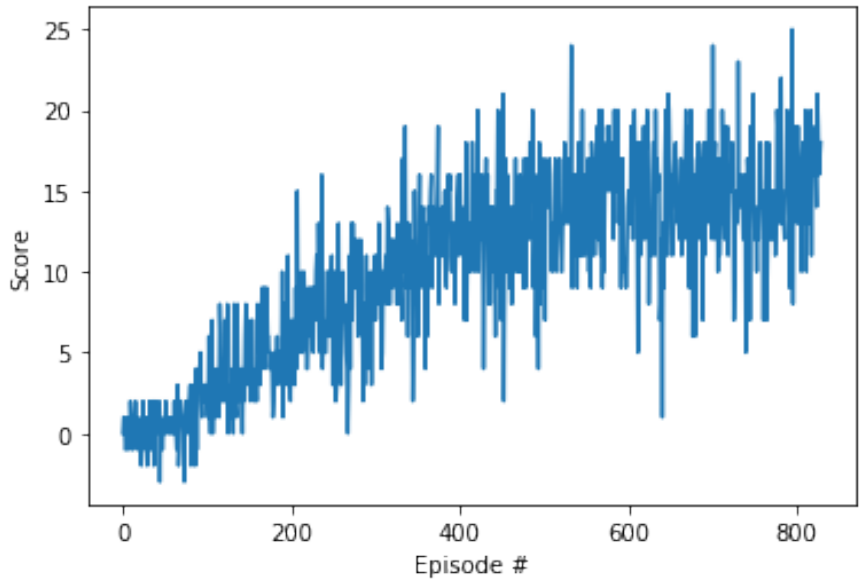
Pseudocode is in Methods section of 2nd reference (Human Level Control through Deep Reinforcement Learning). The code was based from Udacity's Deep Reinforcement Learning Repo [3]. Modifications were made to the NN architectures in the model.py file. And a configuration hook was added to agent. By default, the Agent instance is configured to work in DQN. If "*ddqn*" flag is set to True when instantiating the agent, then DDQN is performed.

3. Results:

With DQN environment was solved in 609 episodes. Following is the plot of the rewards:



With Double-DQN environment was solved in 729 episodes. Following is the plot of the rewards:



As we can see there is not much that distinguishes between in the 2 variants of DQN.

4. Ideas for future work:

- As suggested in the lectures, implementing Prioritized experience replay and Dueling DQN. And comparing it with the performance of DQN and DDQN.

- Since there is no major difference between DQN and DDQN, trying it on a more complex environment, for example, Banana with pixels, might help highlight differences in performance between the 2 variants of DQN.

5. References:

1 – Q-Learning, Christopher J.C.H Watkins, Peter Dayan
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.7501&rep=rep1&type=pdf
2 – Human Level Control through Deep Reinforcement Learning
https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf
3 - https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn/solution