Solving Navigation “BananaBrain” Environment using DQN

#### The algorithm I use is largely based on the DQN algorithm provided by the Udacity code example for solving the OpenAI Gym's LunarLander. The code is very much readable and provides a clear framework for further fine-tuning to solve other problems using DQN. The link to the code as below:

#### <https://github.com/udacity/deep-reinforcement-learning/blob/master/dqn/solution/dqn_agent.py>

#### In this report, I will have a brief review on the DQN paper (<https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>), discuss the model architecture of the implementation and the process of the training and reaching the final solution.

#### Deep Q-learning (DQN)

#### In the DQN paper, the author presented a single model-free and off-policy algorithm using deep function approximators that would be able to develop a wide range of competencies on a varied angle of challenging tasks.

#### The authors consider tasks in which the agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function, which is the maximum sum of rewards rt , discounted by γ at each timestep t, achievable by a behaviour policy π = P(a|s), after making an observation (s) and taking an action (a)

#### Key elements in the DQN algorithm:

#### Uses function approximator to estimate the action-value function, Q(s,a;θ) = Q\*(s,a)

#### The target value method, using parameters θi- from some previous iteration, effectively weights θ will only be updated every N step, by resetting θi- = θi-1

#### A model-free method, without estimating the reward and transition dynamics P(r,s’|s,a).

#### An off-policy method, as it learns about the greedy policy a = argmaxa’Q(s,a’; θ), while following a behavior distribution that ensures adequate exploration of the state space, for which the ε–greedy policy is often used.

#### Replay buffer: a finite sized cache storing *(st, at, rt, st+1).* When the replay buffer was full the oldest samples were discarded

#### Soft target updates: using a separate network for generating the targets yj in the Q-learning update. The weights of these target networks are then updated by having them slowly track the learned networks: θ’ ← τθ + (1 − τ )θ’ with τ << 1

#### Error-clipping: clip the error term from the updated r + maxa’Q(s’,a’; θi-) – Q(s,a; θi) to be between -1 and 1

#### Below is the pseudo code for the DQN algorithm:

#### https://cdn-images-1.medium.com/max/1600/1*8coZ4g_pRtfyoHmsuzMH6g.png

#### Model Architecture

#### I have followed the suggested benchmark implementation to adjust the model architecture. The original model architecture was the same to the Udacity DDPG-pendulum algorithm. The Actor network is a neural network with 2 full connected layers with size of 400 and 300. Tanh is used as the activation function to map states to actions. The Critic network is similar to the Actor network except that the activation function is a fully connected layer with Relu, which maps states and actions to Q values.

#### Each agent adds its experience to a replay buffer that is shared by all agents. The trajectory has max\_t = 2000, and the networks are updated 10 times every other 10 timesteps. It is implemented as below, with LEARN\_EVERY = 10:

if int (t / LEARN\_EVERY ) % 2 == 1:

self.learn(experiences, GAMMA, t)

#### Hyperparameters

#### Below is the final hyperparameters I used to for getting the solution:

#### 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 of the actor

#### UPDATE\_EVERY = 4 # update the target network every N timesteps

#### Things have tried

