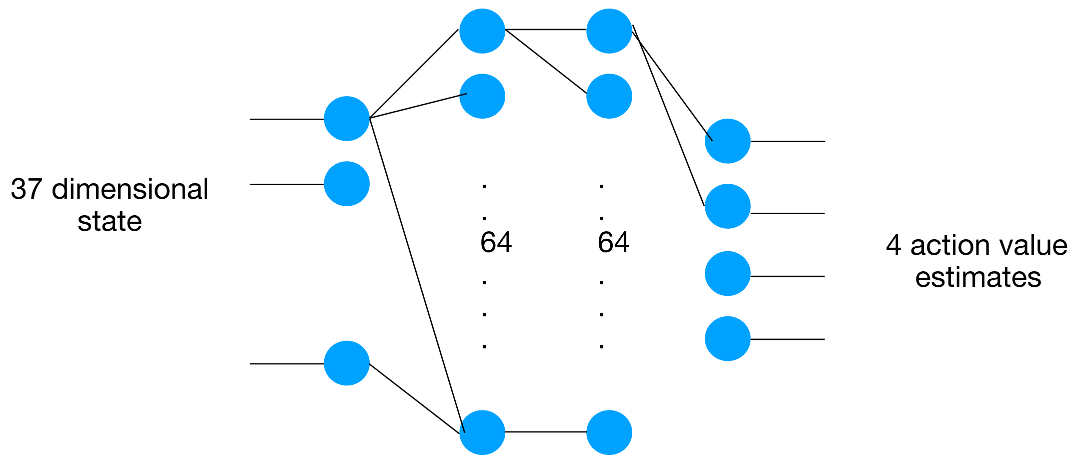
Project Report: Deep Reinforcement Learning Nanodegree - Project 1: Navigation

Learning Algorithm

In the project we train a Deep Reinforcement Learning agent based on the Deep Q Network approach. Specifically, a neural network model is used as a function approximator for the value function.

Here we use a 3 layer network consisting of [64, 64, 4] units.



The network takes in a 37 dimensional vector that indicates the state that the agent is currently observing, and generates a 4 dimensional vector that contains the predicted Q values for each action taken in this state.

Fixed Q-Targets

The approach taken here involves 2 identically structured networks (with separate weights):

- a Local Network
- a Target Network

By having 2 networks, we can avoid a situation where the model is trying to generate a loss function using a target that was generated by itself.

The Local Network is the one that is actively trained through the process. During training, it asks the Target Network for what it thinks would be the Q value for the next state. The weights of this Target Network are updated with a soft-update (i.e. a weighted update that takes into consideration both the current weights of the Local Network and Target Network).

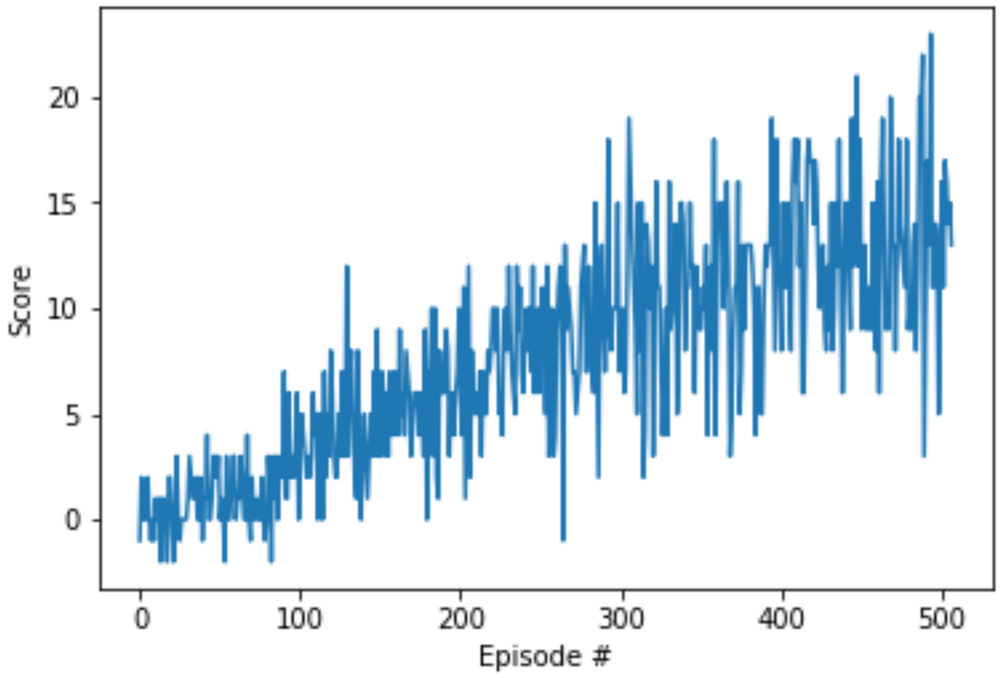
Experience Replay

An experience is defined as a tuple of (state, action, reward, next_state, done). As we progress through training, these tuples are stored in a deque of buffer size 1e5. Batches of size 64 are drawn from this deque, and used to train the DQN.

We use a discounting factor of 0.99 (gamma) while computing rewards. i.e. target reward for the current state = the current reward + 0.99 times the expected reward for the next state.

Rewards

The model converges in around 507 episodes, i.e. reaching an average score greater than 13.



Ideas for Future Work

The following approaches can be potentially used to improve the agent's performance.

a. Prioritized Experience Replay: We can enhance the ReplayBuffer class in dqn_agent.py to also capture the TD error. Later while sampling from this buffer, we can prioritize the experiences that have high error.
See https://arxiv.org/pdf/1511.05952.pdf

b. Double DQN: The current DQN computes the Q value for the next state by doing an max on the different action values for that state. However this approach has been shown to be prone to overestimating Q values. The Double DQN avoid this by having 2 networks — one to get the argmax (i.e index) of the action that maximizes the action value from one network, and the corresponding value from another network
   See https://arxiv.org/pdf/1509.06461.pdf
c. Dueling DQN: Having 2 separate estimators for the state value function, and the state-dependent action advantage function can help in improving generalization. This is another avenue that can be explored.
   See https://arxiv.org/pdf/1511.06581.pdf