Deep Reinforcement Learning Nanodegree – Project 1 “Navigation”

Tim Wu

April 28th, 2020

Introduction

In this project, I built multiple reinforcement learning (RL) DQN agents to navigate Unity's Banana Collector environment. In the following sessions of the report, I summarized how to use different methods to improve the learning of DQN agent and how I selected best agent. I am able to solve the environment in 250 − 350 episodes with a number of different DQN architectures.

Code Location

- The Deep Q-Network model implemented in the file (model.py)
- The Agent Class implemented in the (Navigation.ipynb)
- The model training and etc. implemented in the (Navigation.ipynb)
- The model weights are saved in the (checkpoints) folder

DQN Architectures

The solutions are based on Deep Q-learning Network (DQN) + Improvements:

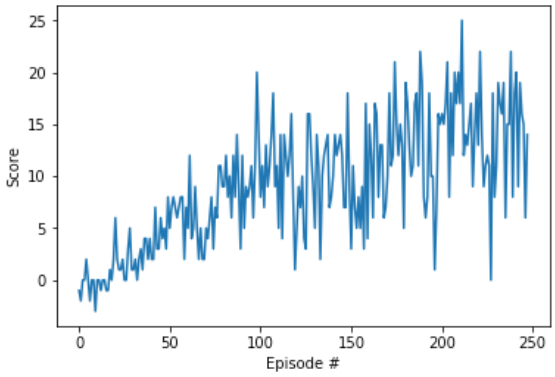
- Deep Q-learning Network (DQN) + Experience Replay
- Double Deep Q-learning Network (DDQN) + Experience Replay
- Dueling Deep Q-learning Network (Dueling-DQN) + Experience Replay
- Deep Q-learning Network (DQN) + Experience Replay + Clip Error
- Dueling + Double-DQN + Experience Replay + Clip Error

Results

The best performing agent is the rainbow network (Dueling + Double-DQN + Experience Replay + Clip Error). This agent has all the improvement and it can solve the environment in 248 episodes. The file with the saved model weights of the best agent saved in the checkpoint folder and named `rainbow_dqn_v1_checkpoint.pth`.

The Best Agent Navigation Result:

Episode 100 Average Score: 4.63
Episode 200 Average Score: 10.83
Episode 248 Average Score: 13.01
Average Score: 13.01



Result Summary (the best agent highlighted in red text):

Agent	Number of episodes to solve the Environment	Average Score
Deep Q-learning Network (DQN) + Experience Replay	334	13.01
Double Deep Q-learning Network (DDQN) + Experience Replay	294	13.00
Dueling Deep Q-learning Network (Dueling-DQN) + Experience Replay	274	13.02
Deep Q-learning Network (DQN) + Experience Replay + Clip Error	321	13.00
Dueling + Double-DQN + Experience Replay + Clip Error	**248**	**13.01**

List the best Agent's Hyperparameters

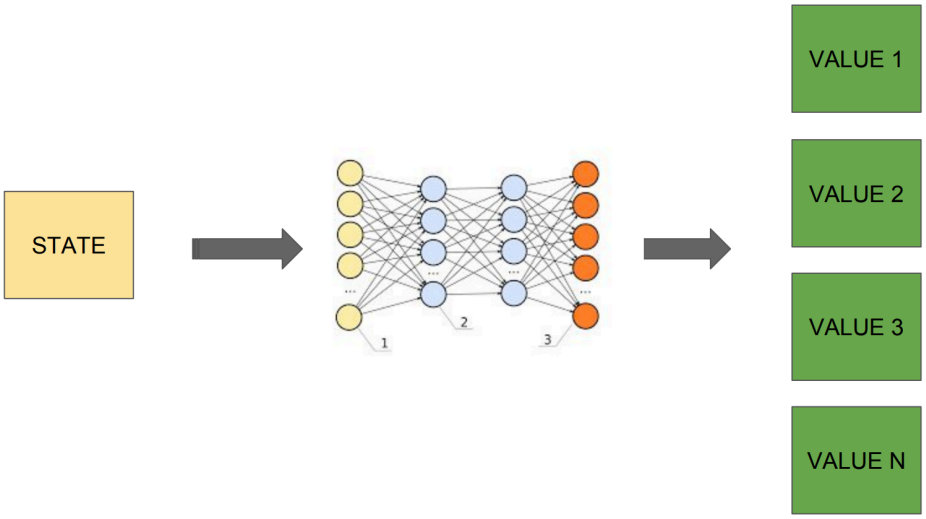
The hyperparameters for the best agent shown below.

```
double_dqn = True          # True and False for double_dqn
dueling = True             # True and False for dueling
clip_error = True          # True and False for clip_error

seed = 123                 # random seed number
hidden_sizes = [64, 64]    # hidden layer units

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
UPDATE_EVERY = 4           # how often to update the network

n_episodes_max = 500       # num_episodes
eps_start = 1.0            # egreedy_start
eps_decay = .97            # egreedy_decay
eps_end = .005             # egreedy_final
ckpt_path = 'rainbow_dqn_v1_checkpoint.pth'
```

Deep Q Network Architecture

After experimenting with different numbers of hidden layers (particularly 1, 2) I concluded that 2 standard feed-forward 64 units layers with ReLu activation give good results. With state space dimension of 37 and output/action space dimension of 4 the problem does not need high numbers of hidden layers and high number of units within the layers.

```
----------------------------------------------------------------
      Layer (type)          Output Shape         Param #
================================================================
          Linear-1             [-1, 64]         2,432
          Linear-2             [-1, 64]         4,160
          Linear-3              [-1, 4]           260
================================================================
Total params: 6,852
Trainable params: 6,852
Non-trainable params: 0
----------------------------------------------------------------
```



Deep Q Network Improvement Methods

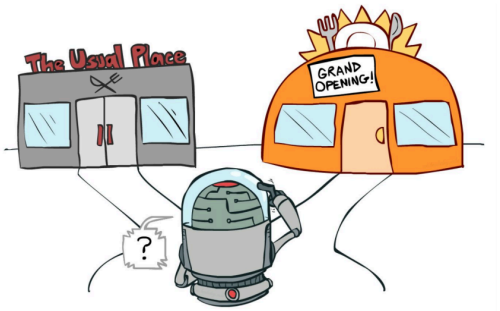
Epsilon Greedy Algorithm

One challenge with the Q-function is choosing which action to take while the agent is still learning the optimal policy. Should the agent go for best decision vs. more information? This is known as the exploration vs. exploitation dilemma.

best decision vs more information

e - greedy:

$$a = \begin{cases} optimal\ a^* & 1 - \varepsilon \\ random & \varepsilon \end{cases}$$



Source: berkeley.edu

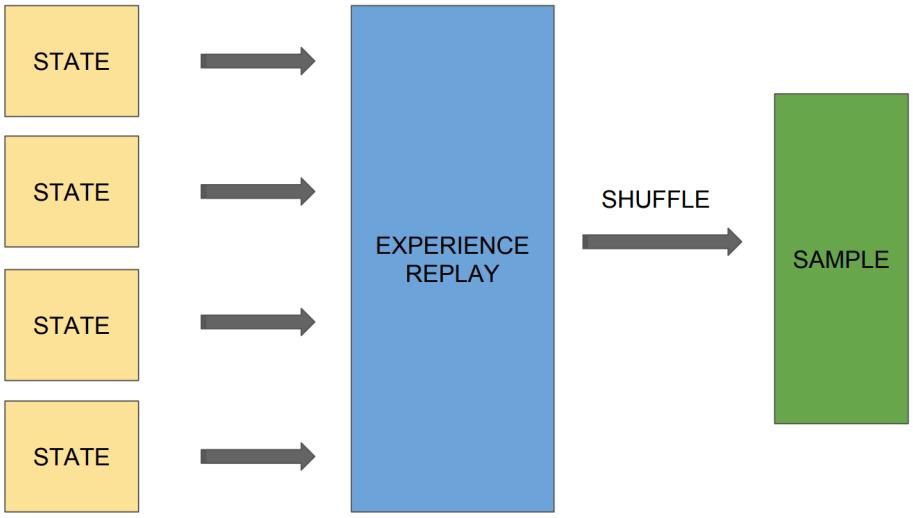
To address this, I implemented an ε-greedy logic implemented as part of the agent.act() method in the (Navigation.ipynb) Jupyter notebook.

Experience Replay

One of the problems listed in Deepmind's paper is that the agent sometimes face highly correlated state and actions and it makes hard converge. Experience replay allows the RL agent to learn from past experience and give 2 major advantages.

- More efficient use of previous experience by learning with its multiple times. This is key when gaining real-world experience is costly, we can get full use of it.
- Better convergence behavior when training a function approximator.

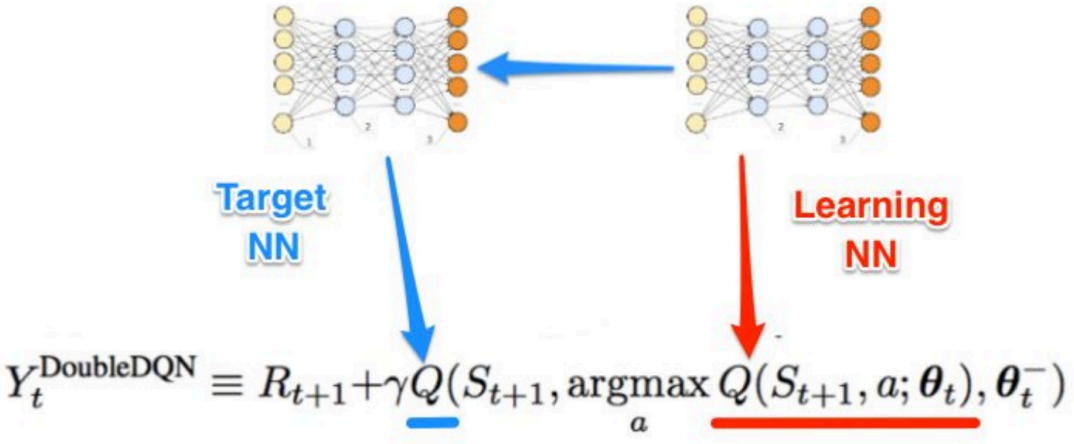
Each experience is stored in a replay buffer as the agent interacts with the environment. The replay buffer contains a collection of experience tuples with the state, action, reward, and next state (s, a, r, s'). The agent then samples from this buffer as part of the learning step. Experiences are sampled randomly, so that the data is uncorrelated. This prevents action values from oscillating or diverging catastrophically, since a naive Q-learning algorithm could otherwise become biased by correlations between sequential experience tuples.



The implementation of the replay buffer can be found in the class Agent(): in the `Navigation.ipynb` file of the source code.

Double Deep Q-Network (DDQN)

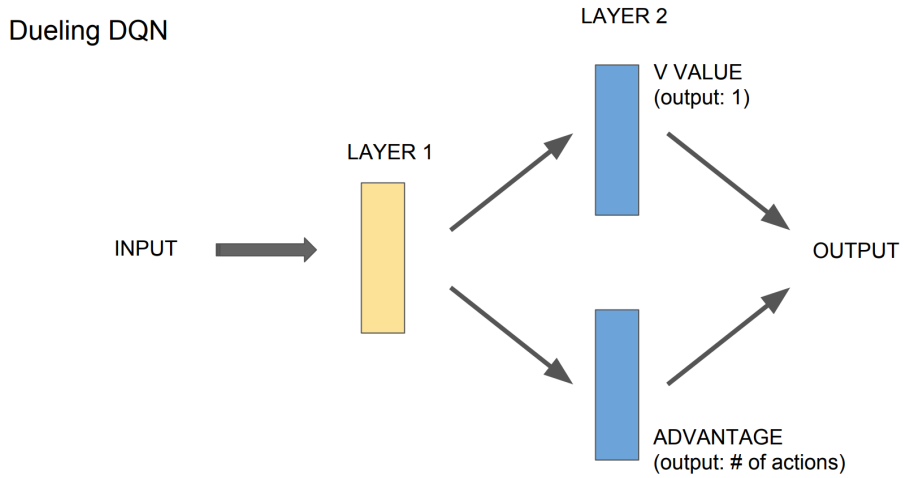
Another problem is that Deep Q-Networks can overestimate Q-values. To solve this issue, we can apply two neural networks, one neural network calculates the Target value and the other neural network chooses the best action.



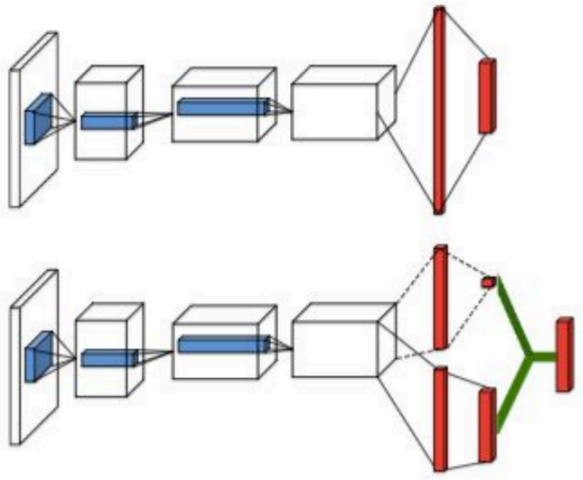
The DDQN implementation can be found in the class Agent(): in the `Navigation.ipynb` file of the source code.

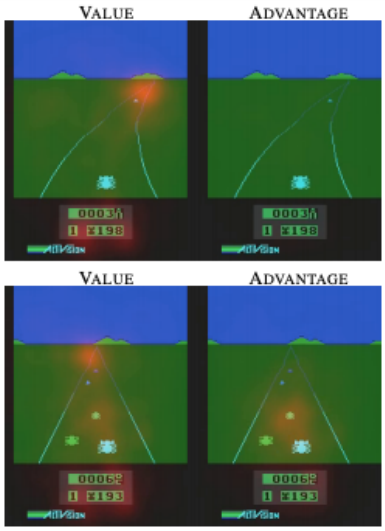
Dueling Network Architecture

Dueling Network Architecture has two steams, which can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way. One of the steam estimates the state value function V(s), and another steam that estimates the advantage for each action A(s,a) – (shown in the figure below).



From the figure below, it shows the value stream learns to pay attention to the road, and the advantage stream learns to pay attention only when there are cars immediately in front, so as to avoid collisions.



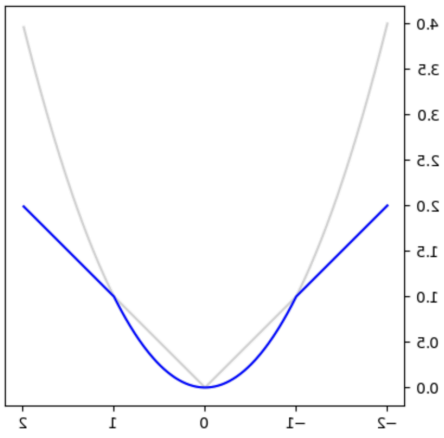


Based on the paper (https://arxiv.org/pdf/1511.06581.pdf), this Dueling Network leads to better policy evaluation in the presence of many similar-valued actions.

The dueling network architecture are implemented within the fully connected layers in the model.py file of the source code.

Error Clipping to avoid exploding gradients

Apply Error Clipping to avoid exploding gradients. Gradients are clipped to a certain threshold value, if they exceed it. Observe that in comparison to the quadratic loss function the derivate of the green curve in the plot shown below does not increase (or decrease) for x>1 (or x<−1).



Future Improvements

---

1. Extensive hyperparameter optimization, fine tune the experience replay feeding buffer size and update frequency
2. Add prioritized experience replay