Udacity
Deep Reinforcement Learning
Nanodegree Program

Project: Navigation

Developer: Koh Thiam Chye

1. Background

For this project, we used the Unity ML-Agents Banana simulated environment to train an agent to navigate (and collect bananas) in a large, square world. A reward of +1 is awarded for collecting a yellow banana while -1 is awarded for collecting a blue banana. The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. The agent will take one of four actions at each time step i.e. walk forward or backward and turn left or right.

The environment is considered solved if the agent is able to receive an average reward (over 100 episodes) of at least +13.

2. Training Environment Setup

a) Operating Environment: Ubuntu 18.04 LTS

b) Imported Libraries:

   i. UnityAgents (ver. 0.4.0)
   ii. Pandas
   iii. Numpy
   iv. Random
   v. Sys
   vi. Torch
   vii. Matplotlib
   viii. Collections

3. Implementation

a) Deep Q-Learning Model (DQN)

Hyperparameters:

- n_episodes (int): 2000 # max no. of episode
- max_t (int): 1000 # no. of time-step in each episode
- eps_start (float): 1.0 # start value of epsilon
- eps_end (float): 0.01 # end value of epsilon
- eps_decay (float): 0.995 # epsilon decay factor
- BUFFER_SIZE = int(1e5) # max size for replay buffer
- BATCH_SIZE = 128 # size of experiences to be sampled
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR = 5e-4 # learning rate

Description of Model Architecture:

For a maximum of 2000 episodes, the agent will interact with the environment to get an action based on initial state and epsilon value of 1.0 (decaying with each episode). The state values are channeled into the local instance of the QNetwork model to compute corresponding scores for each of the four actions. An epsilon-greedy approach, with a degree of randomness between exploration and exploitation (of learned knowledge), is then used to select the action to be taken by the agent.

A step is then taken by the agent in the environment, which in turn returns the next state, reward and done status. The agent updates its memory buffer with the information and after some pre-determined time-steps, it draws a sample of past experiences (state, action, reward, next_state, done) to train the QNetwork model.

To implement reinforcement learning on the QNetwork model, the maximum predicted Q-values for the next state is generated from the target instance of the QNetwork model, which is then used, together with other parameters including reward, gamma discount factor and done status, to compute the targeted Q-values for current state. In addition, the expected Q-values for current state is separately computed using the local instance of the QNetwork model. Finally, the mean-square-loss between the targeted Q-values and expected Q-values are used in the back-propagation of the local instance of the QNetwork model to adjust its weights. These weights are then copied to the target instance of the QNetwork model for the subsequent rounds of reinforcement learning.

When the environment is done, the episode ends, and the cumulative scores are averaged to check whether target reward (in this case +13) has been achieved. The training cycle continues with the next episode until the target reward is achieved or maximum episode is reached.

Please refer to Navigation.ipynb and dqn_agent.py for more details.

Neural Network Model:

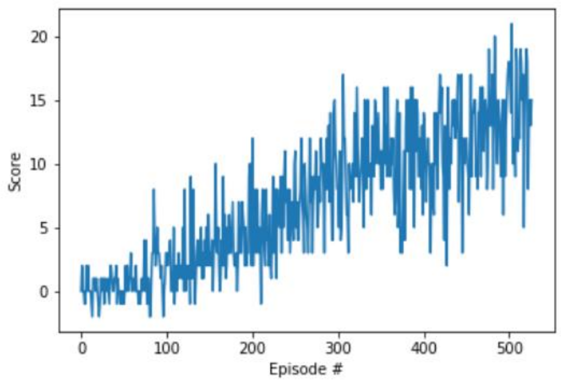
Using a fully connected neural network model (i.e. QNetwork), the state space of 37 values was embedded onto one input layer with 256 nodes, three hidden layers (128 nodes, 64 nodes and 32 nodes respectively) and an output layer with 4 nodes for each of the four actions (walk forward, walk backward, turn left and turn right). ReLu activation was implemented for all the layers.

Please refer to model.py for more details.

Training Outcome:

For a target score of at least +13, the environment was solved using the above mentioned DQN model in 427 episodes.

```
Episode 100     Average Score: 0.70
Episode 200     Average Score: 3.15
Episode 300     Average Score: 6.79
Episode 400     Average Score: 10.24
Episode 500     Average Score: 12.00
Episode 527     Average Score: 13.03
Environment solved in 427 episodes!     Average Score: 13.03
```



b) Double DQN

Hyperparameters:

The same hyperparameters and settings for Deep Q-Learning were utilized for Double DQN.

Description of Model Architecture:

Generally speaking, the Double DQN is considered an enhancement over the DQN technique because at early stages of the learning process, the Q-values in the target instance are still evolving and therefore might not always lead to the selection of best action. Over time, this results in overestimation of Q-values as the action with maximum Q-values is always chosen.

As a workaround, two separate function approximators are implemented for Double DQN. Specifically, Double DQN selects the best action for the next state using one set of parameters (e.g. local instance) and then evaluate that choice using another set of parameters (e.g. target instance). For DQN, the best action for the next state is simply determined by the target instance of the QNetwork alone.

Please refer to Navigation_double_dqns.ipynb and dqn_agent_ddqns.py for more details.

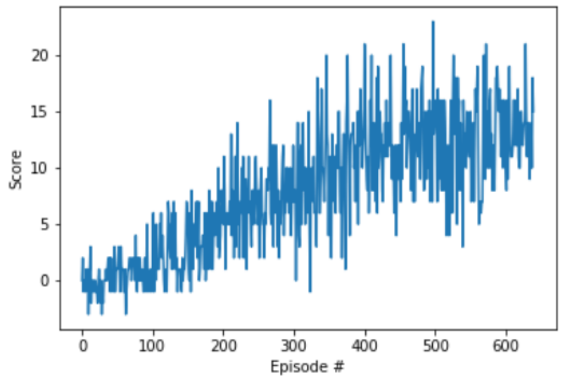
Neural Network Model:

The same neural network model (QNetwork) for Deep Q-Learning was utilized for Double DQN.

Training Outcome:

For a target score of at least +13, the environment was solved using the above mentioned Double DQN model in 540 episodes.

```
Episode 100	Average Score: 0.37
Episode 200	Average Score: 3.22
Episode 300	Average Score: 7.13
Episode 400	Average Score: 9.91
Episode 500	Average Score: 12.33
Episode 600	Average Score: 12.51
Episode 640	Average Score: 13.07
Environment solved in 540 episodes!	Average Score: 13.07
```



c) Double DQN with Dueling

Hyperparameters:

The same hyperparameters and settings for Deep Q-Learning were utilized for Double DQN with Dueling.

Description of Model Architecture:

The model architecture for Double DQN was utilized for Double DQN with Dueling.

Please refer to Navigation_double_dueling_dqns.ipynb and dqn_agent_double_dueling_dqns.py for more details.

Neural Network Model:

The neural network for Dueling consists of a feature layer and two fully connected sequential sub-networks for advantages and state values.

The state space of 37 values was embedded onto the feature (input) layer with 128 nodes. After which, the output from the feature layer are further channeled through the three layers (64 nodes, 32 nodes and 4 nodes respectively) of the advantage values sub-network and the three layers (64 nodes, 32 nodes and 1 node respectively) of the state values sub-network.

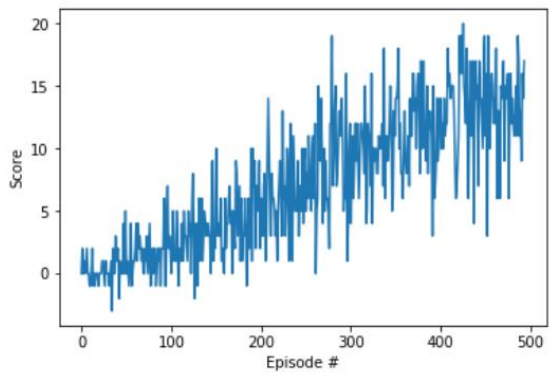
Lastly, the final output (value + advantage - advantage.mean()) is returned.

Please refer to model_dueling_dqn.py for more details.

Training Outcome:

For a target score of at least +13, the environment was solved using the above mentioned Double DQN with Dueling model in 394 episodes.

```
Episode 100     Average Score: 0.95
Episode 200     Average Score: 3.59
Episode 300     Average Score: 7.61
Episode 400     Average Score: 10.64
Episode 494     Average Score: 13.09
Environment solved in 394 episodes!     Average Score: 13.09
```



d) Double DQN with Dueling and Prioritized Experience Replay (PER)

Hyperparameters:

- n_episodes (int): 2000 # max no. of episode
- max_t (int): 1000 # no. of time-step in each episode
- eps_start (float): 1.0 # start value of epsilon
- eps_end (float): 0.01 # end value of epsilon
- eps_decay (float): 0.995 # epsilon decay factor
- BUFFER_SIZE = int(1e5) # max size for replay buffer
- BATCH_SIZE = 128 # size of experiences to be sampled
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR = 5e-4 # learning rate
- PER_e = 0.01 # to avoid experiences with 0 probability of being taken
- PER_a = 0.6 # tradeoff between choosing experience with high priority and sampling randomly
- PER_b = 0.4 # weight of sampling, increasing from initial value to 1
- PER_b_increment_per_sampling = 0.001

Description of Model Architecture:

Prioritized Experience Replay (PER) was enabled by means of a prioritized replay buffer implemented using a binary SumTree* object. The experiences are stored in an array while the priority of the experiences is stored in the nodes of the SumTree. In this way, experiences can be selected according to desired priority by traversing along the leaf nodes of the SumTree and progressively moving up the tree structure.

There are methods to initialize the SumTree with random experiences, to add new experience to the SumTree, to uniformly sample experiences from all segments of priority weights etc.

During the learning process, the agent updates its prioritized replay buffer with new experiences and after some pre-determined time-steps, it draws a sample of past experiences (state, action, reward, next_state, done, indices, weights) to train the QNetwork model. The weights are used for computation of loss between expected Q-values and targeted Q-values during back-propagation of the local instance of the QNetwork model. The priority scores in the SumTree are updated after each round of training.

Please refer to Navigation_double_dueling_PER_dqns.ipynb and dqn_agent_double_dueling_dqns_PER.py for more details.

* Reference: https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/

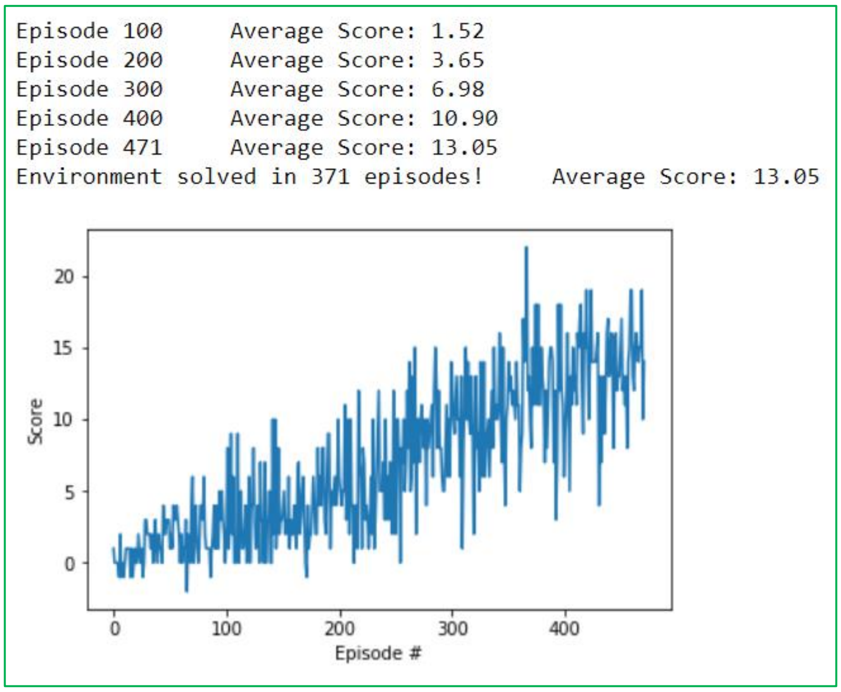
Neural Network Model:

The same neural network for Double DQN with Dueling was utilized for Double DQN with Dueling and PER.

Please refer to model_dueling_dqn.py for more details.

Training Outcome:

For a target score of at least +13, the environment was solved using the above mentioned Double DQN with Dueling and PER model in 371 episodes.



```
Episode 100	Average Score: 1.52
Episode 200	Average Score: 3.65
Episode 300	Average Score: 6.98
Episode 400	Average Score: 10.90
Episode 471	Average Score: 13.05
Environment solved in 371 episodes!	Average Score: 13.05
```