Udacity Deep Reinforcment Learning NanoDegree Beta Test

Project 1: **Navigation**
Beta Tester: **David Calloway**
echooooo@gmail.com

July 17, 2018

Algorithm Overview

For this project, I used an implementation of the Deep-Q Learning algorithm as described in the DQN lesson and provided in the OpenAI Gym Lunar Lander project that was introduced in that lesson. I modified the DQN by:

1. Changing the interface used to communicate with the environment to match up with the Unity ML_Agents environment. I also updated the main navigation.ipynb notebook to initialize the agent to recognize the 37-dimensions of the observation state and the 4 options available for discrete actions.

2. Changing the agent.py to increase the learning rate and use a 3-layer deep-learning model with 64 nodes in the first hidden layer, 64 in the second, and 32 in the third.

3. Changing the model.py class to extend it from a network with two hidden layers to one with 3 hidden layers.

4. Adding a check-pointing feature to both the model.py and agent.py classes to enable the system to save the current state of the model and restore it at a later time. A saved copy of the final model state is included in the navigation_checkpoint.pt file.

DQN Architecture

The main loop of the DQN method executes the specified number of episodes (5,000 in my particular implementation). For each episode, the algorithm:

1. Resets the Unity environment and obtains from the environment an env_info record containing information on the current state, the current reward, and information on whether or not the current episode has finished.

2. Records the current state information (as provided in env_info).

3. Initializes the score for the current episode to zero.

4. And, finally, executes a set of time steps making up a single episode (up to 1,000 in my implementation).

For each of the 1,000 time steps of an episode, my implementation of the DQN logic:

1. Obtains from the agent the next action to take (based on the current state and the current value of epsilon).

2. Tells the environment to take a step using the indicated action.

3. Obtains from the environment the next state, the reward based on the current state and the specified action, and an indication of whether or not the episode is complete.

4. Tells the agent make a training/learning step, based on the current state, specified action, the next state, and the resulting reward.

Agent

Agent Hyper-Parameters

The agent contains several hyper-parameters, including:

1. The replay buffer size (100,000 for my agent)

2. The mini-batch size (64)

3. Gamma, the discount factor (0.99)

4. Tau – a soft update parameter for target data (0.001)

5. Initial learning rate for the fully-connected MLP (0.005), a multiplication factor to reduce learning rate each episode (0.999), and a minimum learning rate (0.00001).

6. UPDATE_EVERY – specifying how often the network is updated (every 4 episodes).

Agent Methods

The agent also has methods to:

1. Step – save experience in replay memory, learn every UPDATE_EVERY time steps.

2. Act – Return an action based on the current state and the epsilon factor. This is accomplished by taking a forward pass through the MLP with the input layer being the current state information and the output layer specifying the resulting action to take. If a randomly-drawn number (between 0.0 and 1.0) is greater than the specified epsilon value, then the best action is returned. Otherwise, a random action is performed (to enable exploration of the state space).

3. Learn - This function performs a back-prop learning pass on the MLP.

4. Soft_Update – performs a soft update of model parameters based on he tau hyper-parameter. The local model is updated based on tau, while the target model is updated based on (1.0 – tau).

5. ReplayBuffer – This is a separate class contained within the agent.py module. It

stores experiences in a replay buffer so that learning can take place not just on the current experience, but also on past experiences.

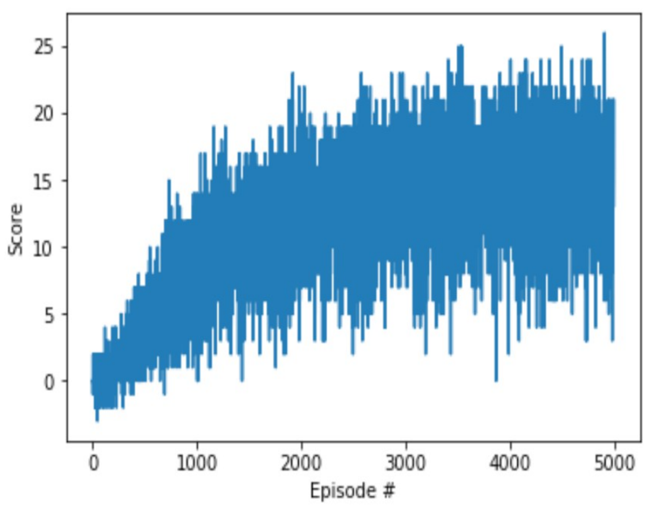
Model

The model is a multi-layer perceptron (MLP): a 3-layer, fully-connected neural network where the first layer takes as input the current state (a 37-dimensional vector for this problem, containing the agent's velocity and a ray-based perception of objects around the agent's forward direction). Three fully-connected hidden layers with RELU activation then feed into an output layer (with linear activation) – where the output is one of 4 actions that might be taken by the agent. The three hidden layers contained 64, 64, and 32 fully-connected neurons, respectively.

Results

I ran my DQN process for 5,000 episodes, and it achieved a final score of 15.65:

```
Episode 4600    Average Score: 15.24    Eps: 0.01       LR: 0.0005014
Episode 4700    Average Score: 14.49    Eps: 0.01       LR: 0.0004537
Episode 4800    Average Score: 15.39    Eps: 0.01       LR: 0.0004105
Episode 4900    Average Score: 15.88    Eps: 0.01       LR: 0.0003714
Episode 5000    Average Score: 15.65    Eps: 0.01       LR: 0.0003361
```



Ideas for Future Improvements

Several things could be done to improve the performance of this agent:

1. The size of the DQN neural network could be increased, either by adding additional layers or by increasing the number of neurons in each layer.

2. The training time could be increased, enabling additional experiences to be learned and encoded in the DQN network weights.

3. The learning rate could be changed – perhaps a higher initial learning rate would enable faster training, or perhaps a lower learning rate might ultimately attain better results.

4. A more-sophisticated version of Deep-Q learning could be implemented, such as the Double-DQN algorithm (see: https://arxiv.org/abs/1509.06461).