Udacity Banana Project Report
by: Jon F. Hauris

Introduction

This project consists of a large square environment filled with yellow and purple bananas. Whenever a yellow banana is encountered and collected a reward of +1 is obtained. When a purple banana is collected a reward of -1 is obtained. Thus, the goal of the agent is to collect as many yellow bananas as possible while avoiding the purple bananas. The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction. Given this information, the agent must learn how to best select actions. Four discrete actions are available, corresponding to:

- 0 - move forward
- 1 - move backward
- 2 - turn left
- 3 - turn right

The task is episodic, and to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

Learning Algorithm

The Double DQN (Seep Q-Learning Network) was employed in the solution of this project. In addition Experience Replay was incorporated to improve network performance. The Deep Q-Learning Algorithm from Lesson 2 was used as a template and inspiration for this work. Double DQN is similar to Deep Q-Learning with the exception that is uses two identical deep Q-learning networks: a local network and a target network.

The local and target networks are identical except the local network evaluates the Q-value at every time step, thus updating its value. The target network only evaluates its “Q-value” periodically. In reality the target network calculates the maximum possible Q-value for the next state.

The variable UPDATE_EVERY defines the periodicity of the target update and is equal to 4 for this model. Thus, the target network will hold Q-values and weights for 4 time steps and then will get updated with the weights from the local network. Meanwhile, the local network is being updated (current predicted Q-values and weights) every time step. After 4 time steps, the weights are transferred from the local network to the target network, thus periodically updating its values. This procedure compensates for an over estimate of the Q-values that would occur if only a single network were employed.

In summary:
local network:

- update current predicted Q-values every time step
- performs an “act” and “evaluate”
- weights are copied from
- used to select appropriate action every time step (using decay epsilon greedy algorithm)

target network:

- update maximum possible Q-value for next state every 4 time steps
- holds weights (and Q-values) for 4 time steps
- weights copied to

Qtarget = R + gamma * Qtarget(next_state, argmax{Qlocal(next_state,next_action, w}, w-)

Here this equation indicates that the next action is obtained from the local network via argmax{Qlocal(next_state,next_action, w}, except my implementation uses decaying epsilon greed instead of argmax to obtain the next action. Thus,
Every time step:
- the next actions are determined from the local network
- these actions are used to calculate the max possible Q-value of the next state via the target network, via the above equation
- This max possible Q-value is calculated using weight from the target network, but actions determined by the local network

Every T time steps:
- learning occurs in the local network and these weights are transferred to the target network

Defined classes and functions

Decaying epsilon greedy is implemented in the "act" function of the agent. This selects an argmax (greedy) action with probability 1-epsilon and selects a random action with probability epsilon. Epsilon is decayed by epsilon_decay each time step.

The agent contains the following functions:
- step: this adds the present state, action, reward, next_state, and done to the replay buffer. Additionally, every 4 time steps it randomly samples BATCH_SIZE data from the replay buffer and calls the "learn" function
- act: this function uses the local network to select the next action
- learn: this gets the maximum predicted Q values for the next state from the target model. It also gets the expected Q value from the local model and then performs optimization and back prop to train and update the weights in the local model. It then transfers these weights to the target network
- soft_update: transfers weights from local network to target network

The ReplayBuffer established the queue for a replay buffer. This buffer is used to randomly select past experiences so as to avoid correlation among actions and next state. It has functions to add and randomly sample data to and from the queue.

The QNetwork is a PyTorch implementation of a class from which the local and target networks are derived. This particular implementation consists of 5 layers of fully connected linear networks. The first 4 layers using an ReLu activation. This large implementation was used because of the large size on the input feature vector and the depth of layers provided good discrimination of the feature. The following is a print out of the model:

```
QNetwork(
  (fc1): Linear(in_features=37, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=64, bias=True)
  (fc5): Linear(in_features=64, out_features=4, bias=True)
)
```

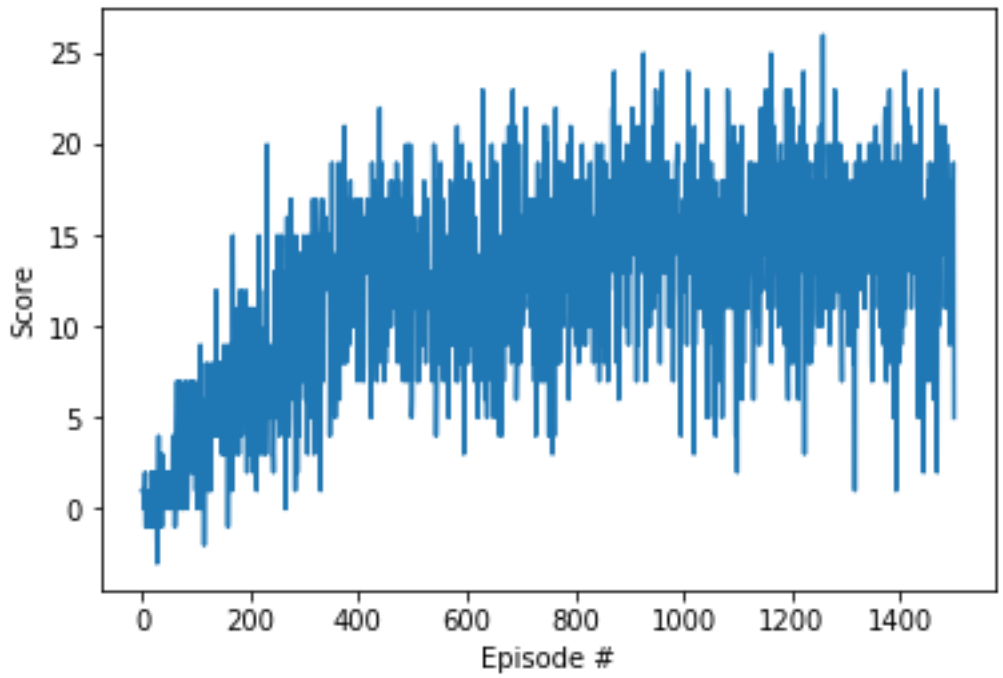
The hyperparameters were chosen via learning what other networks used and by trial and error exploration of various numbers. Finally, the following list was settled on:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.9 #0.99           # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 0.0001 #5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

The learning rate was chosen to speed up learning but not over estimate. And GAMMA was chosen to provide a slight horizon look ahead (instead of an immediate look ahead).

Results

The above model proved to be very good at solving this problem. It achieved an average score well under the 1800 benchmark (typically within 600 iterations). In addition, it achieved a score of +15.0 in several cases. These results are summarized in the following plot and table. Thus, demonstrating that the agent is able to receive an average reward (over 100 episodes) of at least +13.



Episode 500 Average Score: 13.14
Episode 600 Average Score: 12.46
Episode 700 Average Score: 12.53
Episode 748 Average Score: 13.47
Environment solved in 748 episodes! Average Score: 13.47
Episode 749 Average Score: 13.48
Environment solved in 749 episodes! Average Score: 13.48
Episode 750 Average Score: 13.50
Environment solved in 750 episodes! Average Score: 13.50

Episode 751 Average Score: 13.61
Environment solved in 751 episodes! Average Score: 13.61
Episode 765 Average Score: 13.74
Environment solved in 765 episodes! Average Score: 13.74
Episode 766 Average Score: 13.77
Environment solved in 766 episodes! Average Score: 13.77
Episode 767 Average Score: 13.81
Environment solved in 767 episodes! Average Score: 13.81
Episode 768 Average Score: 13.86
Environment solved in 768 episodes! Average Score: 13.86
Episode 775 Average Score: 13.87
…
Episode 1000 Average Score: 15.26
Episode 1100 Average Score: 13.85
Episode 1200 Average Score: 15.28
Episode 1220 Average Score: 15.58
Environment solved in 1220 episodes! Average Score: 15.58
Episode 1222 Average Score: 15.62
Environment solved in 1222 episodes! Average Score: 15.62
Episode 1231 Average Score: 15.65
Environment solved in 1231 episodes! Average Score: 15.65

Suggestions for future work

There are three main suggestions for future work. First would be to implement Dueling DQN. This uses a single front end DQN but incorporates two extension networks at the back end. One of these networks calculates the state values V(s) while the other extension calculates the Advantage values A(s,a) for each action. Here the values of most states do not vary across actions, so these can be directly estimated. However, we still need to capture the differences actions make in each state and the Advantage does this. Finally, $Q(s,a) = V(s) + A(s,a)$. This network has demonstrated significant improvements.

The second improvement would be to implement prioritized experience replay. Since older and important experiences may get lost, a priority is assigned to each tuple in the replay buffer. This priority is related to the TD error. The larger this error, the more we expect to learn from that tuple. So tuples with higher TD error are assigned a higher priority.

Finally, a third improvement would be to operate on the input image itself rather than the state vector. Or operate on a combination of the 37 dimension state vector and the input image. This combined data would provide a lot more information, particularly color. However, convolutional neural networks would have to be employed. I believe this would give a more accurate result but would also increase training time.