Navigation Project Report

By Kevin Lee

Sep 3, 2018

Learning Algorithm

This project uses a deep Q network to train an agent to pick bananas. A deep Q network algorithm consists of two main parts :

1. Re-inforcement learning
2. Deep Neural Network

The deep neural network is used as a universal function approximator. The main RL algorithm navigates the environment and feed to the DNN experience tuples that contain current state, action and next state as input. Then, the DNN learns to produce the appropriate next actions by trying to maximize future rewards.

There is another important component of the algorithm, which is the replay buffer. Instead of feeding the experience tuples directly in to the DNN as they are generated from the environment, the tuples are instead temporarily stored in a replay buffer. Only after certain number of tuples have accumulated in the replay buffer, learning will be carried out.

To learn, a random batch of experience tuples will be extracted from the replay buffer. By using a batch of random samples from the replay buffer, the correlation among the input tuples which were generated in sequence will be broken. That improves convergence. In addition, depending on the application, some data might be more important than others. Therefore, we could potentially vary sampling probability according to the importance of the data.

There is another critical component of the learning algorithm. As shown below, the Q_targets and Q_expected are kept as two separate sets. If it is not done this way, oscillations will happen while the neural network tries to learn and the results will become unpredictable.

```
# Get max predicted Q values (for next states) from target model
Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
# Compute Q targets for current states
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

# Get expected Q values from local model
Q_expected = self.qnetwork_local(states).gather(1, actions)
```

Hyper Parameters

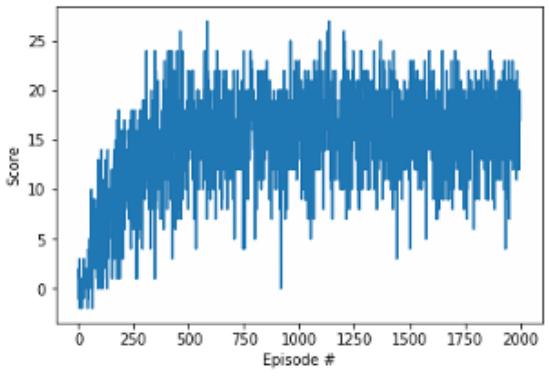
There are many parameters that can be tuned. Examples are :

BUFFER_SIZE	- replay buffer size
BATCH_SIZE	- minibatch size
GAMMA	- discount factor
LR	- learning rate
UPDATE_EVERY	- how often to update the network
EPS	- Epsilon

I tried varying buffer size, batch size, learning rate and update_every but found that I couldn't improve the results very much. Then I tried changing epsilon.

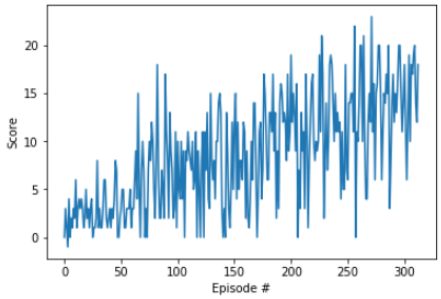
Instead of a linear decay of epsilon, I tried starting with a slow decay. That means letting more random learning in the beginning. I started epsilon at 0.7. And then decay it aggressively, ie multiplying by 0.7 every 50 episodes. Then I achieved very good results. At about 1000 episodes, I could hit average score between 16 - 17. As shown by the graph below, the score actually hit a high and plateaud after about 600 to 700 episodes.

Episode 50	Average Score: 0.72	Epsilon: 0.70000
Episode 100	Average Score: 2.77	Epsilon: 0.34300
Episode 150	Average Score: 6.13	Epsilon: 0.16807
Episode 200	Average Score: 8.07	Epsilon: 0.08235
Episode 250	Average Score: 10.00	Epsilon: 0.04035
Episode 300	Average Score: 11.64	Epsilon: 0.01977
Episode 350	Average Score: 12.87	Epsilon: 0.00969
Episode 400	Average Score: 14.02	Epsilon: 0.00475
Episode 450	Average Score: 15.12	Epsilon: 0.00233
Episode 500	Average Score: 15.33	Epsilon: 0.00114
Episode 550	Average Score: 15.15	Epsilon: 0.00056
Episode 600	Average Score: 16.07	Epsilon: 0.00027
Episode 650	Average Score: 16.44	Epsilon: 0.00013
Episode 700	Average Score: 16.07	Epsilon: 0.00010
Episode 750	Average Score: 15.88	Epsilon: 0.00010
Episode 800	Average Score: 16.39	Epsilon: 0.00010
Episode 850	Average Score: 16.38	Epsilon: 0.00010
Episode 900	Average Score: 15.87	Epsilon: 0.00010
Episode 950	Average Score: 15.64	Epsilon: 0.00010
Episode 1000	Average Score: 16.13	Epsilon: 0.00010
Episode 1100	Average Score: 15.89	Epsilon: 0.00010
Episode 1150	Average Score: 17.00	Epsilon: 0.00010
Episode 1200	Average Score: 16.69	Epsilon: 0.00010
Episode 1250	Average Score: 16.35	Epsilon: 0.00010
Episode 1300	Average Score: 16.45	Epsilon: 0.00010
Episode 1350	Average Score: 16.47	Epsilon: 0.00010
Episode 1400	Average Score: 16.62	Epsilon: 0.00010
Episode 1450	Average Score: 16.44	Epsilon: 0.00010
Episode 1500	Average Score: 16.07	Epsilon: 0.00010
Episode 1550	Average Score: 16.48	Epsilon: 0.00010
Episode 1600	Average Score: 16.89	Epsilon: 0.00010
Episode 1650	Average Score: 15.92	Epsilon: 0.00010
Episode 1700	Average Score: 15.05	Epsilon: 0.00010
Episode 1750	Average Score: 15.66	Epsilon: 0.00010
Episode 1800	Average Score: 16.44	Epsilon: 0.00010
Episode 1850	Average Score: 16.35	Epsilon: 0.00010
Episode 1900	Average Score: 16.18	Epsilon: 0.00010
Episode 1950	Average Score: 16.28	Epsilon: 0.00010
Episode 2000	Average Score: 16.48	Epsilon: 0.00010



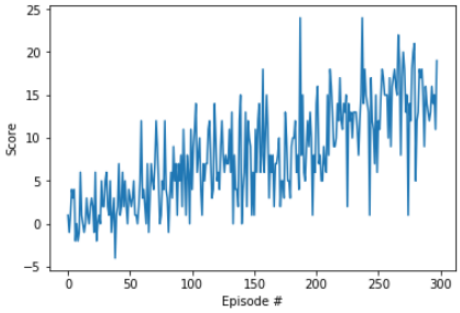
Other Results

```
Episode 100     Average Score: 4.19
Episode 200     Average Score: 8.27
Episode 300     Average Score: 12.38
Episode 313     Average Score: 13.03
Environment solved in 213 episodes!     Average Score: 13.03
```




```
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
```

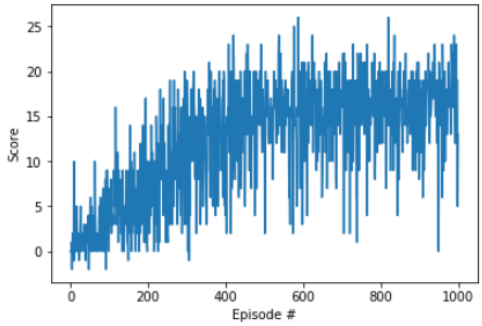
```
Episode 100     Average Score: 3.15
Episode 200     Average Score: 7.77
Episode 298     Average Score: 13.04
Environment solved in 198 episodes!     Average Score: 13.04
```




```
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
```

```
if i_episode % 100 == 0:
    print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
    counter += 1
    eps_decay = eps_decay_set * counter
```

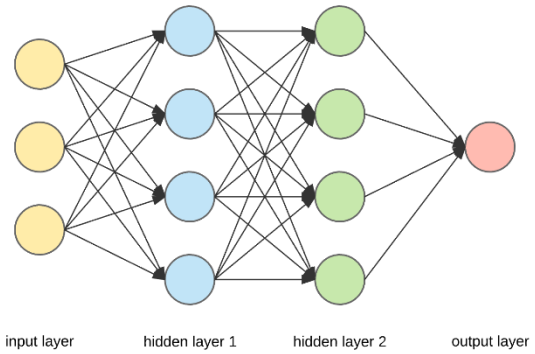
```
Episode 100     Average Score: 2.11
Episode 200     Average Score: 5.46
Episode 300     Average Score: 8.81
Episode 400     Average Score: 12.29
Episode 500     Average Score: 14.71
Episode 600     Average Score: 15.29
Episode 700     Average Score: 15.63
Episode 800     Average Score: 15.77
Episode 900     Average Score: 16.11
Episode 1000    Average Score: 16.63
```




```
counter += 1
eps_decay = eps_decay_set * counter * 2
```

```
eps_decay_set=0.05
```

Model Architecture



Just for illustrating the network architecture
Not the exact number of units for each layer

```python
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

The deep neural network model used in this project closely resembles the graph above, although the actualy number of units are not fully drawn by the graph. The network is fully connected.

There are a total of two hidden layers with 64 units each and an output layer with 4 units. The activation functions used is the RELU.

Next Steps

Further steps can be taken to potentially improve the performance of the agent. These steps are beyond the scope of current project.

The firs thing I would try is to vary the architecture of the DNN. I would try deepening as well as widening the network.

The next thing I would try would probably be double DQN, dueling DQN, or prioritized experience replay. Some of these methods might potentially improve the performance of the agent.