Report

November 27, 2018

1 Project 1

Written by: Anthony Vergottis

The aim of this project is to explore a value-based method for solving the reinforcement learning problem. The algorithic implementation is based upon a simple Deep Q Network, implementing a model-free approach. The agent interacts with a enviroment whose state space is composed of 37 continuous values. There are for possible actions for the actions space: - `0` - walk forward - `1` - walk backward - `2` - turn left - `3` - turn right

The goal of the agent is to pick up yellow bananas (receiving a reward: +1) whilst avoiding blue bananas (receiving a reward: -1). The criteria for a good solution implementation require that the agent receives a average reward greater that 13 for 100 episodes. The simulation environment is provided by Unity ML.

The report is split in four parts:

1. **Solution Strategy**
2. **Hyperparameters**
3. **Results**
4. **Future work for improvements**

2 Solution Strategy

Given the problem takes place in a continuous state space we cannot use tabular methods to approximate the values $Q(s,a)$. In fact we require a function approximator that maps the state of the environment to the $Q(s,a_{1\ldots n})$ for each possible action. We deploy a neural network to solve this problem. Using PyTorch a fully connected neural network with two hidden layers was created. Each layer having 64 fully connected nodes. The input to the neural network is a state space vector of length 37, whilst the output is a vector of length 4 (for possible actions). The Adam gradient descent algorithm was used to update the weights.

The weights θ of the neural network act as additional parameters used to approximate the $Q(s,a)$ by defining the expected value as $\hat{Q}(s,a,\theta)$ values. $\hat{Q}(s,a,\theta)$ is expected to be the return $R+\gamma * max(Q(s',a))$. Given that the values $\hat{Q}(s,a,\theta)$ depend on trainable parameters, this means that values will oscillate backwards and forwards if some considerations are not taken. This is the inherent problem with the Deep Q Network algorithm. Thankfully, there is a solution.

1. Target Network: We turn the problem into a supervised learning process. We deploy two identical neural networks with the same initialised values for θ. One network (Network 1)

acts as the target by not updating the weights at each iteration. The next state is now used to compute $\hat{Q}(s', a)$ to update the current state values by not updating the weights at each iteration. The next state is used to compute $\hat{Q}(s, a)$. The local online neural network (Network 2) takes the current state and outputs the expected $Q(s, a)$, whilst constantly update its θ parameters at each iteration. A specified update frequency is defined to allow for the parameters is Network 1 to be updated using the new exact values of the parameters in Network 2.

2. Experience Replay: A memory buffer is filled with (s, a, r, s') tuples that is used for training the network. In order to further stabilizes learning, we would like to perform the training steps of non-consecutive observation from the environment. This is done to remove the negative impact of correlations between to adjacent transitions. Once the memory buffer has at least enough samples for the specified batch size, samples are then drawn at random and training starts.

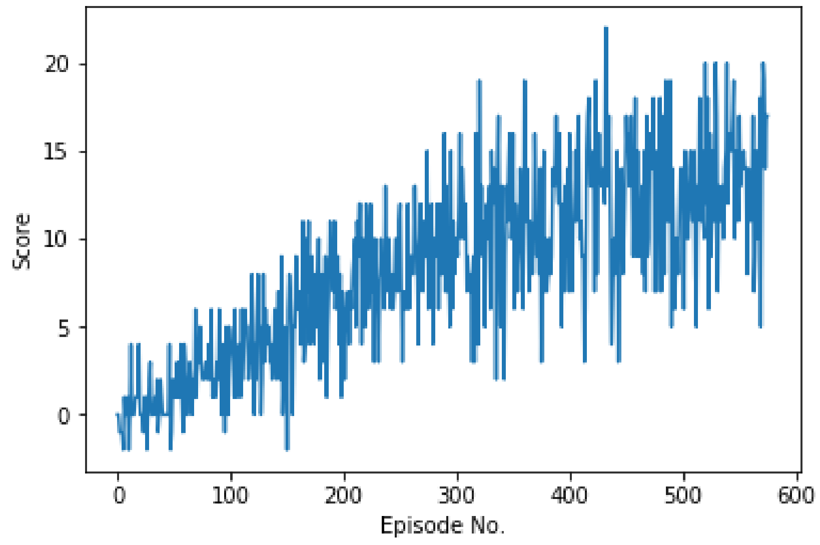
3 Hyperparameters

The values are named as they are found in the code:

```
- 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 = 2000       # Limit of number of episodes to run
- max_t = 1000            # Max No. of time steps per episode
- eps_start = 1.0         # Epsilon starting value
- eps_end = 0.01          # Smallest epsilon allowed
- eps_decay = 0.995       # Epsilon decay rate
```

4 Results

The DQN agent solved the environment in 449 episodes, with an Average Score: 13.03



5 Future work for improvements

1. Implement Dual Deep Q Network algorithm
2. Try using prioritized replay memory
3. Look into the Dueling Deep Network algorithm
4. Search for optimal hyperparameters and network architectures

```
In [ ]:
```