Udacity Deep Reinforcement Learning Nanodegree

Project 1 – Navigation

Submitted by Harry Shaun Lippy

03 April 2021

1 Implementation and Learning Algorithm

The project uses the Unity ML environment for collecting bananas. The environment contains yellow bananas and blue bananas. The goal of the agent is to collect as many yellow bananas as possible while avoiding the blue bananas. Agent controllers are provided by the environment and are referred to as **brains**.

The implementation uses a Deep Q Network (DQN) to allow the agent to use deep reinforcement learning to arrive at a solution. There are three primary implementation blocks, described below.

1.1 Q Network

The **`QNetwork`** class defines the model that the DQN algorithm uses to update Q values that are used in learning. The model consists of a neural network with two hidden layers, each using a ReLU activation function. Each layer is fully connected and contains 1024 nodes. Several different values for the hyperparameters representing the node counts were tested; the values of 1024 were chosen based on the ability of the network to converge on a solution in under 1000 episodes.

1.2 Agent and ReplayBuffer

The agent is the entity that interacts with the environment and takes actions that direct its movement. The four actions available to the agent are:

- 0 – walk forward
- 1 – walk backward
- 2 – turn left
- 3 – turn right

Several hyperparameters are set for the agent, as described in Table 1.

Hyperparameter	Description	Value
BUFFER_SIZE	Number of experiences to store in the experience replay buffer	10000
BATCH_SIZE	Size of the sample batch of experiences to be returned in each sampling from the experience replay buffer	64
GAMMA	The discount factor used in the learning	0.99
TAU	Used in the weight updates	0.001
LR	The learning rate	0.0005
UPDATE_EVERY	How often to update the network	4

Table 1 – Agent hyperparameters

1.2.1 Agent

The **`Agent`** class defines the agent. The size of the state and action spaces are stored, along with two Q Network fields – one local and one target, as prescribed by the DQN algorithm. An Adam optimizer is used by the neural networks. Finally, a replay buffer (see definition below) is used to incorporate experience replay to improve stability in the algorithm.

The class contains four method, defined below.

1.2.1.1 step
This method moves the agent one step forward in the episode. The experience replay buffer is updated with the current experience, and if a batch of samples are available in the buffer a random subset is drawn and the learn method is called using this subset.

1.2.1.2 act
This method is used to determine an action for the agent to take after the local network is trained and the weights of the network updated. Epsilon greedy is used, so that with a probability of epsilon a random action will be chosen from the set of available actions; otherwise, the action producing the highest reward is chosen.

1.2.1.3 learn
In this method, Q values from the local and target Q networks are used to compute the loss function. This loss is then used with the optimizer to compute a round of back propagation in the neural networks. Finally, the target network weights are updated using the **`soft_update`** method.

1.2.1.4 soft_update
This method simply updates the weights in target network by using the weights from the local network in the following formula:

```
θ_target = τ*θ_local + (1 - τ)*θ_target
```

1.2.2 ReplayBuffer
The ReplayBuffer class implements the experience replay component of the DQN algorithm, the usage of which was briefly described above. Without experience replay, only the most current experiences are used to update the learning parameters in the network. This is inefficient, as experiences from earlier in the training can still provide value to the agent learning in the environment. The replay buffer allows the training to incorporate earlier experiences by using random sampling from these earlier experiences.

1.2.2.1 add
This method adds an experience to the experience replay memory buffer.

1.2.2.2 sample
This method returns a sample batch of experiences from the buffer. The batch size is a parameter set for each instance – in this case, the batch size is set to 64.

1.3 DQN
The DQN algorithm is implement in the Python function **`dqn`**. An agent is instantiated above the function that will be used to interact with the environment in the function. The function consists of a loop across a defined number of actions, which defaults to 2000. The maximum number of timesteps per episode defaults to 1000.

1.3.1 epsilon
A decaying epsilon is used for the epsilon-greedy implementation, with the value starting at 1.0 (to enable pure random action selection) and the final value set to 0.01. The value of epsilon is decayed using a factor of 0.995.

1.3.2 Main loop

In the main **`for`** loop, we loop over the number of episodes, resetting the environment before each episode is executed in the inner loop. The starting state is pulled from the environment and then the inner loop executes the episode.

1.3.3 Inner loop (episode execution)

The inner **`for`** loop executes each individual episode. The loop is over the maximum number of time steps (set to 1000 for this implementation, but the parameter can be modified if desired). An action is selected from the agent's **`act`** method, and a step in the environment is undertaken to update the state. A reward for the agent taking the action in the given state is provided by the environment, and we determine if the episode is complete (stored in the **`done`** variable). Finally, the agent uses the start state, the next state, the action taken and the reward received to take the next step.

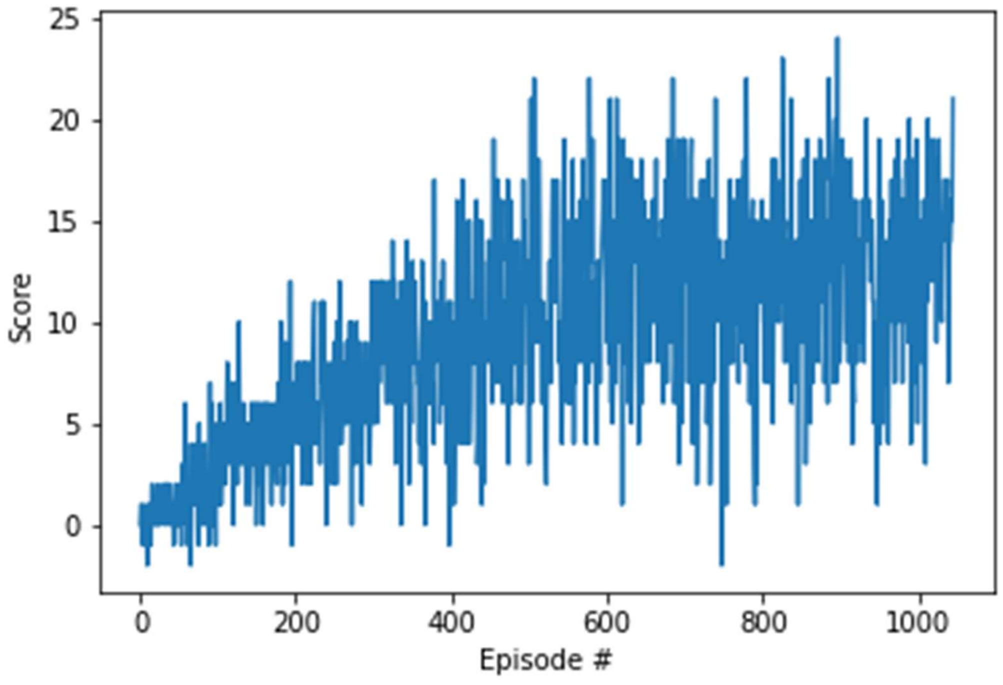
1.3.4 Scores

After each episode is completed, the score, which is the sum total of rewards received by the agent in that episode, is appended to a list. For each 100 episodes completed, the mean score for that 100-episode window is computed and displayed in the output.

After all episodes have been completed – either the maximum number of episodes is hit, or a mean score of 13 or higher was achieved in a 100-episode window – the scores are plotted in a window so that the learning progression of the agent can be understood. The plot produced for this implementation and execution is given in section 2.

2 Plot of Rewards

The environment was solved in 945 episodes. A plot of rewards per episode is provided below to demonstrate the learning progression.



3 Ideas for Future Work

There are several improvements that have been devised to improve the efficiency and efficacy of the DQN algorithm. Two such improvements are described here.

3.1 Prioritized Experience Replay

The experience replay used in the algorithm implemented here using pure random sampling to draw from the remembered experiences. But some experiences are more valuable to the algorithm, such as those that lead to higher loss. By assigning priorities to the experiences in the experience replay buffer, the algorithm can maximize its use of experiences that will allow it to learn more efficiently.

3.2 Dueling DQN

A paper produced by several researchers at Google DeepMind in 2016 updates the "standard" DQN algorithm to use two estimators that provide separate estimates of the state-value function and the state-dependent action advantage function. This allows the algorithm to generalize learning across the set of actions that the agent can take in the environment.[1]

[1] Wang, Ziyu et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. Arxiv.org