Udacity Deep Reinforcement Learning – Project 1 (Navigation)

Author – Benji Alwis

Introduction

Reinforcement Learning provides a framework to learn what actions to take in a given situation with the ultimate goal of maximising long term returns. In other words, it is learning the mapping between situations and actions.

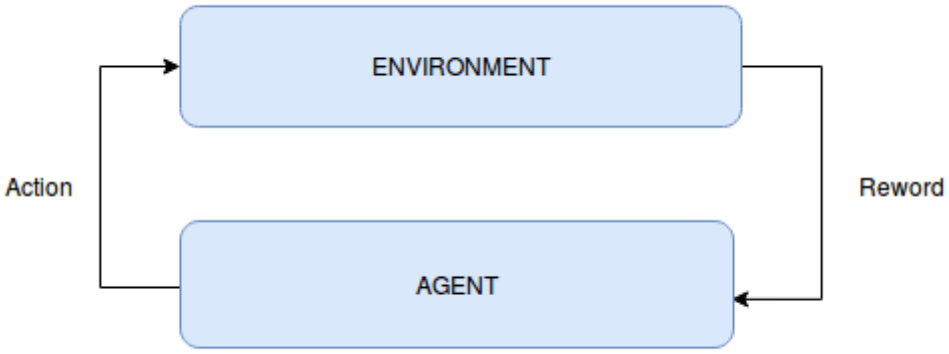


Figure 1 – Reinforcement learning as interactions between an agent and the environment

As shown in figure 1, reinforcement learning problems can be modelled as interactions between an agent and the environment. The agent interacts with the environment by taking actions. This interaction results in agent transitioning to a new state and receiving a reward. The key objective in reinforcement learning is to maximise the cumulative reward received by the agent.

An agent may include one or more of the following components:

Policy – agent's behaviour function. It maps from states to actions. It can be either stochastic or deterministic.

Value function – goodness of each state and/ or action. It indicates the expectation of long term reward.

Model – agent's representation of the environment. It can be represented using Markov Decision Process (MDP) under the assumption that the future actions and transitions depend on the present state and the past states are irrelevant to the calculation once the present state is known.

Reinforcement learning is different to supervised learning due to a number of key characteristics.

- there is no supervision
- feeback is delayed

- states and actions are sequential (not i.i.d)
- action that an agaent takes affects the future states and actions.

Model-based Reinforcement Learning

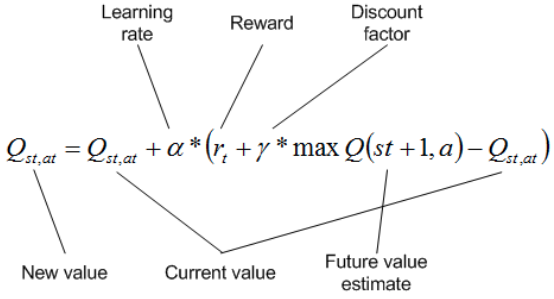
In model based reinforcement learning, a model is utilised to construct the agent's representation of the environment. In MDP, discrete states are represnted as nodes in a graph and discrete actions, rewards and transitions are represnted with the use of edges linking the nodes. State value (V(s)) and State Action value (Q(s,a)) can be calculated using the Bellman equation.

Model Free Reinforcement Learning

It is common to have situations where full information on state transitions are not available in order to construct Markov models or the number of states is too large making such approaches impractical.

Model-free methods utilise direct experience from episodes. There are two widely used approaches for mode-free learning.
Monte-Carlo (MC) Learning adopts direct learning from episodes without bootstrapping. It uses the simple idea that the value of a state is the average return obtained from that state over all episodes experienced by the agent. The main limitation of this approach is that it can only be applied for complete episodes. The MC estimates are typically of high variance (noisy) but low bias.

Temporal Difference (TD) Learning can be used to learn also from non-episodic experiences via bootstrapping. This is an advantage over MC since TD can learn from incomplete, non-terminating sequences. The main disadvantage is its high bias caused due to bootstrapping estimates. It can be summarised by the following equation.



Q Learning

Q Learning is an off-policy learning algorithm that follows a form of Temporal-Difference learning.The core idea is that an agent takes an action and transitions into a new state. Then, we use the reward and maximum state-action value (often denoted as Q(*s*, *a*) , where *s* represents the state and *a* represents the action being evaluated) of the new state to improve the estimate of the current state. The idea is that once we take an action and transitions into a new state, we use the Q-value of the new state as an estimate for future rewards.

Value Function Approximation Using Neural Networks

Use of a neural network to approximate action values allows estimation of actions against a large state spaces.

In a Deep Q-Learning Network (DQN), states are represented as an input vector and the Q-value of different actions are given as outputs. From this, we can choose the optimum action based on the optimum action value. However, this approach can be highly unstable. Deep Q-Learning algorithm addresses these instabilities by using two key features:

Experience Replay:

One key assumption in training a neural work is that the dataset is taken from independent (i.i.d) samples. However, if learning is done using sequential experiences, then they are linked to each other. To avoid this issue, a rolling history of past data is kept inside a re-play pool of fixed size. After each episode, the experience is stored inside the buffer. After a fixed number of iterations, a few experiences are sampled from this replay buffer and use that to calculate the loss and eventually update the parameters. Sampling randomly this way breaks the sequential nature of experiences and stabilizes learning. It also helps use an experience more than once.

Fixed Q-Targets:

The difference between the TD-target and the estimation of the Q-value obtained from the local network determines the update of weights of the local network. However, the TD-target itself is dependent on the same network weights , the parameter being updated. This leads to constantly moving targets and hurts training. The idea behind fixed q-targets is to fix the weights used in the calculation of the TD-target. This is achieved by having two separate networks, one for the local network and the other being the target network. The weights of the target network are taken from the local network itself by freezing the model parameters for a few iterations and updating it periodically after a few steps.

Double DQN:

One of the known drawback of the DQN method is that it over-estimates the state action value due to maximum Q value calculation for the next state-action pair. The idea of Double DQN is to seperate finding the best action from calculating the Q-value for that action. This achieved by using one neural network to identify the best action and another to evaluate that action.

Implementation

model.py:

Q-Network class implements a 3-layer (2 hidden layers and the output layer) fully connected neural network implemented using the PyTorch Framework.

The size of the input layer depends on the size of the state vector representation. This parameter is passed as *state_size* in the constructor. Each of the hidden fully connected layers has 64 nodes. The size of the output layer is set using the *action_size* parameter passed in the constructor

dqn_agent.py:

This file contains the *DQN agent* and *replay buffer* implementations.

DQN agent

The *DQN agent* class has the following methods:

constructor():

It initialises two Q learning networks: the **target network** (*qnetwork_target* - used for Temporal Difference based calculation) and the **local network** (*qnetwork_local* - used for function approximation of state action values). The width of the input and output layers are set using the parameters, *state_size* and *state_size*, respectively.

It also initialises the memory buffer (*Replay Buffer*).

Step() :
This function receives state, action, reward, next_state, done as inputs and stores them in the *ReplayBuffer*.

After every fixed number of steps, it updates the target network (*qnetwork_target*) weights using the current weight values from the local network (*qnetwork_local*). Before this update, a check is performed to verify if there are enough samples in the memory buffer,

act():
This function receives a state value as input and returns actions for the given state according to the epsilon-greedy action policy.

learn():
This function receives a set of experiences from the *Replay Buffer*. The neural network used for action value network approximation (*qnetwork_local*) is updated inside this function. It can use two alternative types of update methods. The first is the conventional DQN update. The second is the Double DQN rule in which the best action is first identified using *qnetwork_local* and then the validation step (i.e. obtaining the state action value) is done using *qnetwork_target*.

soft_update():
Thid function is called during the learning process to update the value from the target neural network using the local network weights.

The ReplayBuffer

This class contains the functions needed to manage the temporary storage of experiences (*state, action, reward, next state, done*).
The functions are:

add(): to add an experience into memory
sample(): to randomly retrieve a batch of experiences for learning

DQN_Navigation_Banana_Project.ipynb

This file contains the reference project starter code used to train the agent. It is a Jupyter notebook file.

Hyper-parameters

The following hyper-parameters were used during the experiment.

Hyper-parameter	**Value**
Replay buffer size	1e5
Batch size	64
Discount factor	0.99
Tau	1e-3
Learning rate	5e-4
update interval	4
Number of episodes	5000
Max number of time-steps per episode	2000
Epsilon start	1.0
Epsilon minimum	0.1
Epsilon decay	0.995

Results

I ran the training process first using the standard DQN method (results in figure 2) and then did the same using Double DQN method (results in figure 3). Using the first method, the score of 13+ was achieved in 462 episodes while it was achieved in 437 episodes using the second method, indicating that the double DQN method is better.

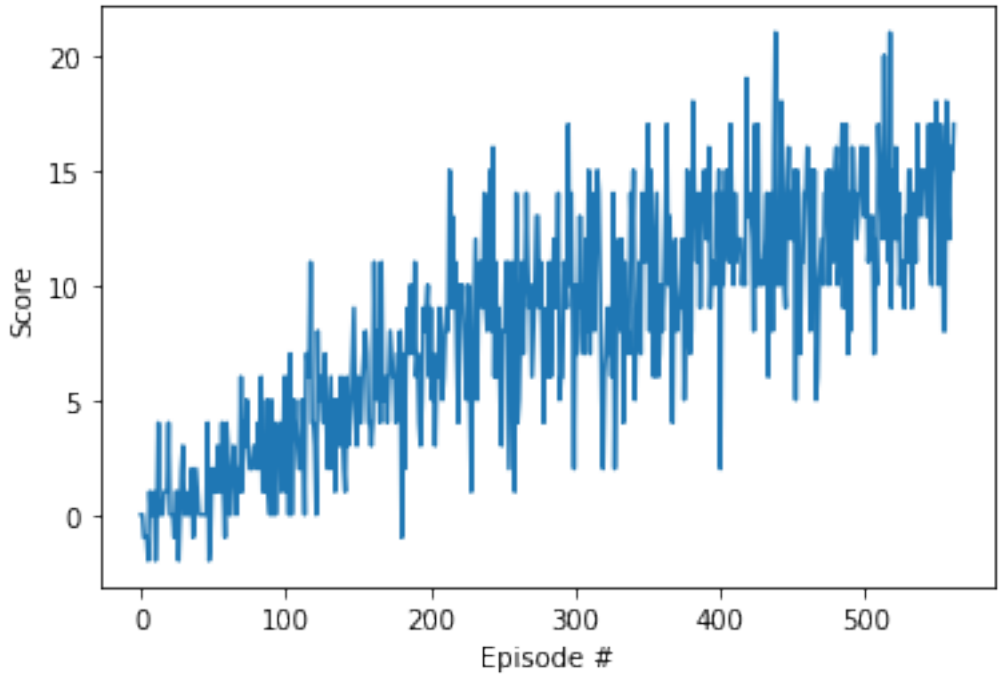


Figure 2 – Results obtained using the standard DQN method.

```
Episode 100     Average score:  1.35
Episode 200     Average score:  5.25
Episode 300     Average score:  8.59
Episode 400     Average score:  10.31
Episode 500     Average score:  12.40
Episode 562     Average score:  13.02
Environment solved in  462 episodes!    Average Score:  13.02
```

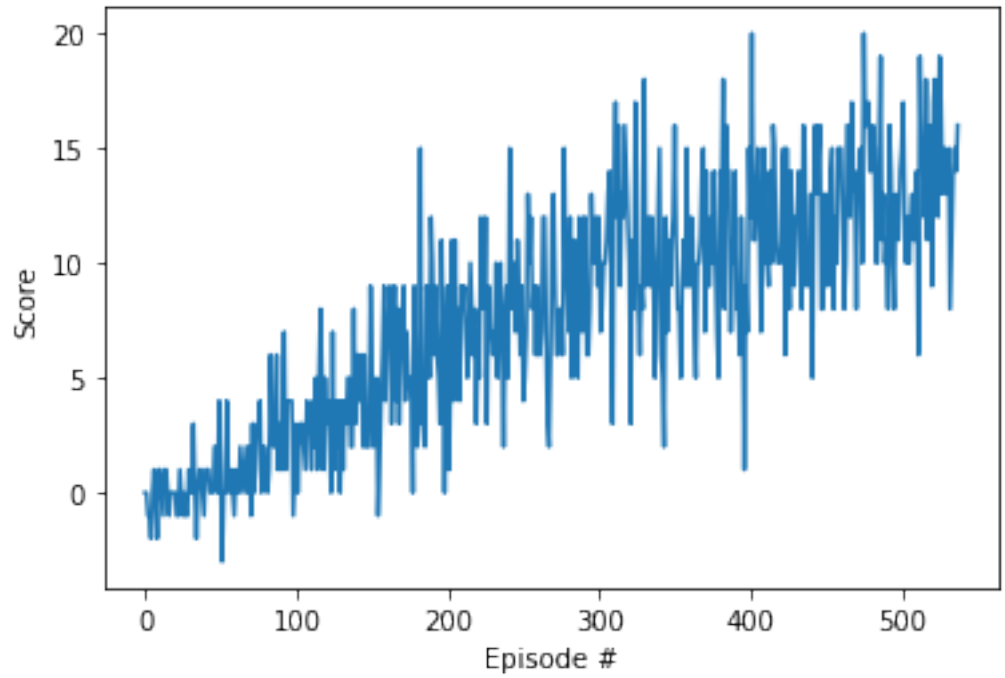


Figure 3 – Results obtained using the Double DQN method.

```
Episode 100     Average score:  0.98
Episode 200     Average score:  5.47
Episode 300     Average score:  8.36
Episode 400     Average score:  11.34
Episode 500     Average score:  12.54
Episode 537     Average score:  13.08
Environment solved in  437 episodes!    Average Score:  13.08
```

Ideas for improvement

I would like to try the Prioritised Replay Buffer, Dueling DQN and RainBow methods to see whether I can achieve further improvements.

Prioritised Replay Buffer

The objective of this approach is to assign normalised priority values for the training samples based on the TD error before they are stored inside the replay buffer. The logic behind this approach is to guage the usefulness of different training samples.

Dueling DQN

Instead of estimating state action values (*Q(s,a)*), this method calculated the state value (*V(s)*) and the advantage of different actions for the same state (*A(s,a)*) as separate outputs from a neural network. The intution behind this approach is that, in some occassions, the state value does not greately vary over different actions and hence better to directly calculate state values.

RainBow Method

In this method we will combine six different approaches. They are Double DQN, Prioritised Replay Buffer, Dueling DQN, multi-step bootstrap targets (as in A3C), Distributional DQN and Noisy DQN.