Reinforcement Learning (DEEP Q-LEARNING)

Abstract

This paper surveys a subfield of Artificial Intelligence, Reinforcement Learning. Reinforcement Learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward. In this paper, Deep reinforcement learning which is a combination of reinforcement learning (RL) and deep learning is discussed as an application to solve one of *Open AI Gym's* environments, the Lunar-Lander environment.

PROBLEM DESCRIPTION:

The problem is based on one of *Open AI gym's* environments, the *LunarLander-V2* environment. *Open AI Gym* helps by providing a lot of environments to provide users with a platform to train and benchmark their reinforcement learning agents. The goal is to teach and train the agent to successfully land on the 'moon'.

This paper implements and records the corresponding results of the DQN algorithm on the *LunarLander-V2* environment.

ENVIRONMENT:

In the LunarLander-v2 environment there exists a landing pad at co-ordinates **(x, y) = (0,0)**. The goal of the lander is to reach the landing pad as soon as possible in the most safe and efficient manner.

-**The state space** includes an 8-dimensional vector:

(*X, Y,* V_x, V_y, θ*, left leg, right leg)*

(X, Y) denotes the X and Y co-ordinates with respect to the current position of the lander, V_x and V_y denotes the velocity of the lander in the X and Y directions respectively. Θ denotes the angular velocity of the lander and (left leg, right leg) are two flag values (0,1) indicating whether which leg left/right (or both) has made contact with the ground.

-**The action space** involves a discrete set of 4 actions to choose from: do nothing, fire left engine, fire main engine and fire right engine.

-**Reward:**

At each episode the landing pad is always at the same position(co-ordinates). The fuel is infinite and landing away from the landing pad is possible. The episode finishes if the lander crashes or comes to rest, reward for crashing involves an additional -100 points and the reward for coming to rest involves +100 increment. Reward for moving from the top of the screen to the landing pad and zero speed is about 100 to 140 points. If the lander moves away from the landing pad it loses reward. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing

the side engine is -0.03 points each frame. Finally, if the lander manages to land at the landing pad it receives an additional 200 points.

2 Q-LEARNING WITH NEURAL NETWORKS

The tabular based Q-learning algorithm is defined as:

$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$ Here, s' is the state at the next step and γ denotes the discount factor which indicates the scale of importance to be given to future rewards. This update is done online on each sample that's collected by the behaviour policy.

In Deep Q-Learning, a neural network is used and the network learns a set of weights to approximate the Q-value function. The objective is to optimize the weight, θ so that the $Q_\theta(s,a)$ reaches the optimal Q-value function but since the target Q-value function is not known we can take small steps to minimize the Temporal difference error $r + \gamma max_{a'} Q(s',a') - Q(s,a)$

In Deep Q-Learning, instead of updating the parameters of the network with respect to each state transition, mini-batches are used.

To adapt Q-learning with deep neural networks we have to come up with a loss function (or objective) to minimize.

The loss function that is used is the **MSE (MEAN-SQUARED-ERROR).**

$L(\theta) = E_{(s,a,r,s')}[(y_i - Q_\theta(s_i, a_i))^2]$ here y_i is the target Q-value and is denoted by:

$y_i = r_i + \gamma max_{a_i'} Q_\theta(s_i', a_i')$

The network parameter, θ, is updated by gradient descent on the MSE loss function

L(θ): $\theta = \theta - \alpha \nabla_\theta L(\theta)$

The gradient of the Q-function with respect to the parameters, θ, is given as:

$\theta \leftarrow \theta - \alpha[r + \gamma max_{a'} Q_\theta(s',a') - Q_\theta(s,a)] \nabla_\theta Q_\theta(s,a)$

Here, $\nabla_\theta Q_\theta(s,a)$ is the partial derivate of *Q* with respect to θ. α is called the learning rate, which denotes the amount by which the weights should be modified to reach the gradient.

The key innovations brought by DQN involve a *replay buffer* to get over the data correlation drawback, and a separate *target network* to get over the non-stationarity problem. The non-stationary problem is due to continuously updating the network during training, which also modifies the target values. Nevertheless, the neural network has to update itself in order to provide the best possible state-action values. The solution that's employed in DQNs is to use two neural networks. One is called the *prediction* network, which is constantly updated, while the other is called the *target* network, which is updated only after certain iterations of training.

Neural network:

The neural network is first defined by specifying the number of hidden units as well as the number of neurons per hidden unit. The input the network shall be fixed to the size of the state vector dimension, in this case 8, a *Rectified Linear Unit* activation function is applied to each hidden layer as it allow faster and effective training of deep neural architectures on large and complex datasets, the neural network would then output a vector of dimension 4, one for each state action pair Q(s, a),

We use two networks:

- *A prediction* network
- A *target* network

Experience Replay Buffer:

Every state the Lander encounters is stored in the form of a tuple, $(s, a, r, s_t, terminal)$, to a memory unit referred to as the *Experience Replay Buffer*. Where *s* denotes the current state, *a* denotes the action being taken at that state to reach the new state s_t receiving reward *r*. *Terminal* is a 0-1 flag indicating whether the episode has ended.

Number of replay steps indicate how many times the network is updated per transition. In each replay step, we sample a batch of experiences from the replay buffer and update the *prediction* network weights. Across these N replay steps, we will use the current "un-updated" *target* network at time t, Q_t, for computing the action-values of the next-states. This contradicts using the most recent action-values from the last replay step $Q^i{}_{t+1}$. We make this choice to have targets that are stable across the replay steps.

Parameters

ε -> Epsilon denotes the exploration-exploitation trade off parameter, where *ε* ∈ [0,1]. The agent chooses a random action with probability *ε* and chooses the best action with a probability of *1 - ε.*

γ -> Gamma denotes the discount rate, *γ* ∈ [0,1].

α->Learning rate.

Remarks:

If epsilon is high than the chances of choosing a random action is high and if epsilon is low then the chances of choosing the best action is high. Higher the value of γ, more importance is given to future rewards and lower the value, immediate rewards are considered more significant.

ALGORITHM:

Algorithm 1: DQN-ALGORITHM

Input: Environment parameters, all the parameters for the network and max no.of episodes

Output: Trained optimal Q-value Function

$\epsilon \leftarrow \epsilon_{start}$
$episode \leftarrow 0$
for $episode \leq EPISODE_{max}$ **do**
  reset the environment;
  while *episode is not done* **do**
    with probability ϵ *sample an action a from the sample space*
    with probability $1-\epsilon$ *choose a that maximises* $Q(s,a)$
    Given an action a the environment returns observation $(s_t, r, terminal, info)$ *where terminal is a 0-1 flag indicating whether the episode ended*
    if *Replay_Buffer is full* **then**
      remove the oldest transition
    Add the transition $(s, a, r, s_t, terminal)$ *to the Replay_Buffer*
    if *Minibatch_size* $\geq$*BUFFER_SIZE* **then**
      set Target_Model weights to Prediction_Model weights
      for *No.of Replay Steps* **do**
        Sample mini_batch transitions from $Replay\_Buffer$
        Update the parameters of the Prediction_model through optimization with learning rate α *using :*
        $r_t + max_a Q(s', a)$ *as target values and the loss function*
    if $\epsilon \geq \epsilon_{min}$ **then**
      $\epsilon \leftarrow \epsilon * \epsilon_{decay}$

Experiments:

The algorithm is implemented using the *TensorFlow* and *Keras* libraries. We need to select an appropriate network size, how many hidden layers and units, the hyperparameters are then tuned to improve the performance of the model.

Each network was trained for only for 500 episodes, since their weights were updated during each step of an episode for a certain number of replay steps. The networks were then tested for 300 episodes.

Since it is necessary for the agent to explore in the beginning stages of learning and exploit its new learnt knowledge during the later stages of training, we decrease the value of epsilon by a specific rate, *epsilon decay rate,* with each new step. ε starts of at a ε_{max} value of 1.0 and is then decayed by $\varepsilon_{decay\text{-}rate}$ of 0.998 until it reaches ε_{min} value of 0.01.

As the end goal of the agent is to land safely, we need to prioritize future rewards as well and hence a large *discount factor(γ)* of 0.99 was chosen. The *step-size* parameter used was 0.001. A *replay buffer* of size 50000 was used with a mini batch size of 64. The number of replay steps used was 4.

During parameter tuning, only one parameter is varied keeping the others constant to see the effect.

Difficulties:

Finding the optimal parameters and tuning them for the best performance is not feasible as there exists a large number of parameters and training a neural network is time consuming.

Experiment 1: Network size selection.

One 3-layer network with size 512/256/128, two 2-layer networks with size 64/32 and 256/128 and two 1-layer networks with 128 and 256 hidden units were tested. The 3-layer network took a fairly large amount of time to train and never managed to receive a positive reward throughout the first 100 episodes. The networks with size 128 and 64/32 yielded in low results and had an average reward of **0.981** and **3.109** during training.

The networks 256/128 and 256 yielded optimal results with an average result of **141.129** and **65.776** during training. After testing these networks resulted in an average reward of **256.547** and **101.112** respectively.

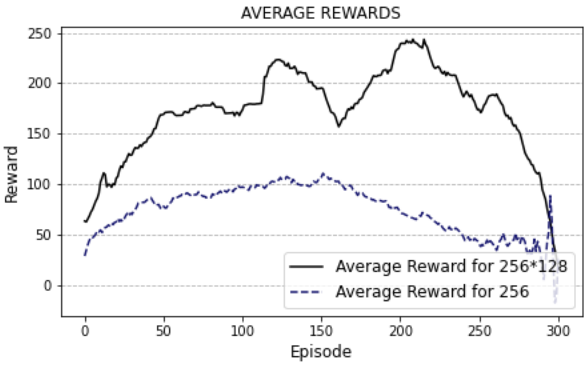

Fig 1.0, illustrates that during training both the 256 and 256/128 sized networks had received a positive reward within the first 100 episodes.

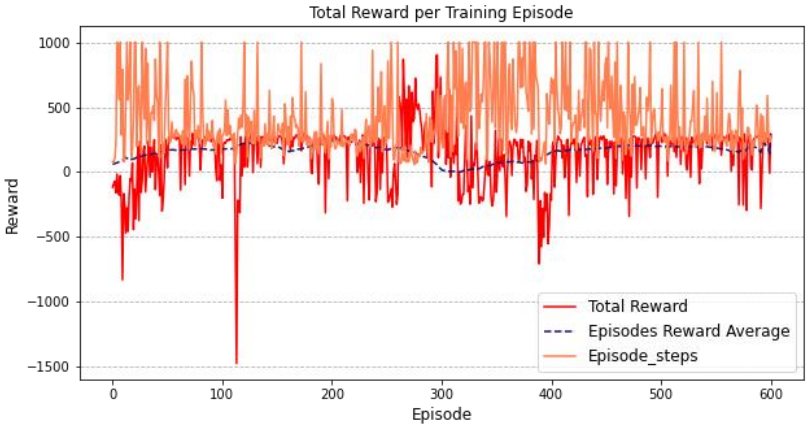

Fig 2.0 shows that when the 256/128 network was trained for around 600 episodes, it soon had optimal results. The oscillation of the total reward per episode is due to the agent selecting a random action from the action space, as the epsilon value is never reduced to 0.

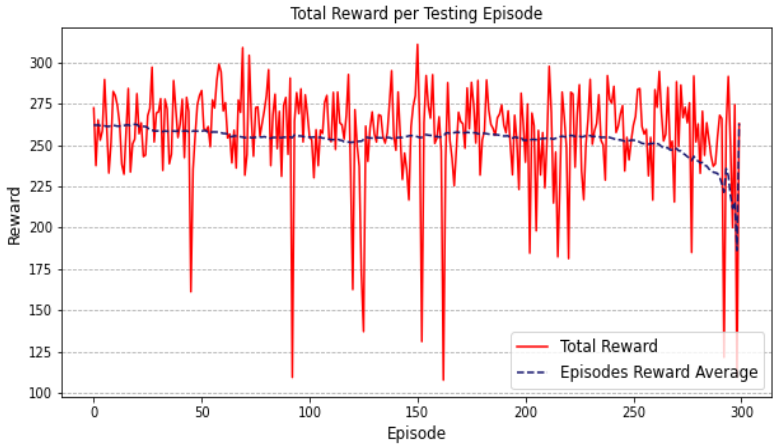

Fig 3.0 illustrates the performance of the 256/128 network during testing. Hence the **best result** noted was the **256/128** sized network layer.

Experiment 2: Discount factor γ

3 trials were conducted with gamma as 0.99,0.95 and 0.90. The network 256/128 was used and the rest of the parameters were unchanged.

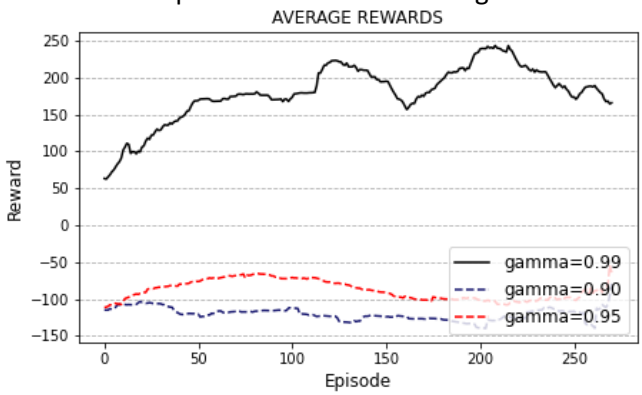

Fig 4.0 shows that the **best** value for the *discount factor γ* is **0.99**. The values 0.90 and 0.95 valued immediate rewards more in comparison to the future rewards and hence did not receive any positive reward within the first 300 episodes.

Experiment 3: The step size/ learning rate parameter α The 1-layer 256 sized network was used to tune the learning rate parameters. 3 values of α was used with 0.01,0.001 and 0.0001.

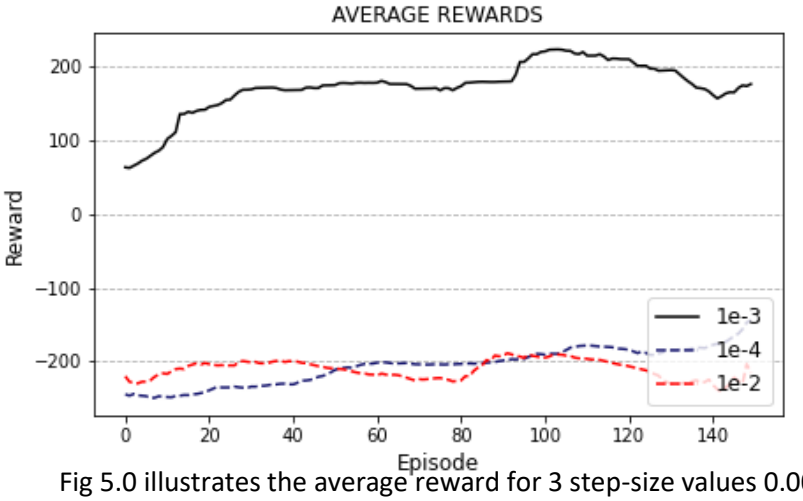

Fig 5.0 illustrates the average reward for 3 step-size values 0.001,0.0001 and 0.01. The model responded well with a learning rate of 1e-3 as shown but for both 0.0001 and 0.01 performed inaccurately. With a large learning rate of 0.01, the network weights are updated by a large factor whereas for 0.0001 the weights are affected by a small percentage. An optimal value of 0.001 manages to update the weights in the required manner per updation step and hence has a better reward.

Experiment 4: Epsilon$_{start}$ value

3 trials were conducted with ε_{max} of 1.0, 0.9 as well as 0.5. The single 256-layered network was considered for this experiment to reduce computational time. It was seen that lower the ε_{max} value lower was the mean average reward and higher was the computational time.

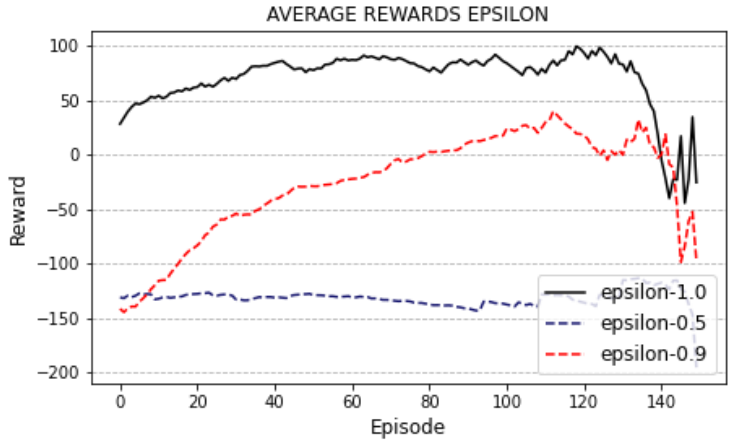

Fig 6.0 indicates the change in average reward with the change in *exploration factor*, ε. Since the ε value is reduced by a *decay rate* of 0.998 per step until it reaches a ε_{min} value of 0.1, if ε_{max} value is small then it takes lesser number of steps to reach the minimum value and hence reduces the chances of exploration. For ε_{max} = 1.0, 0.9 and 0.5 the number of steps of allowed exploration include **2300, 2200** and **1900**, respectively.

It was also noted that reducing the starting value of epsilon caused the agent to take more steps per episode, i.e. it was unable to reach the terminal state rapidly. Since **ε_{max} = 1.0** has the highest number of exploration steps to reach its minimum value, it has the highest average reward and has a comparatively lesser computational time.

It is seen that when ε_{max}= 1.0 takes 2300 steps to reach a ε_{min} of 0.1 when the decay rate was 0.998 but when the decay rate is reduced to 0.9 and 0.5 it takes only 43 and 6 steps respectively to reach the minimum value, which means less exploration. Furthermore, reducing the ε_{min} value closer to 0 leads to more exploitation in the later stages and hence better results.

REFERENCES

1. Mnih, Volodymyr, et al. "Human-level control through

deep reinforcement learning."

2. Mnih, Volodymyr, et al. "Playing atari with deep

reinforcement learning."

3. Andrew Barto and Richard S. Sutton, “Reinforcement Learning: An Introduction”

4. Sayon Dutta, “Reinforcement Learning with TensorFlow “