Q Learning Applications: Basic and Advanced Applications Using OpenAI Environments

Tareq Galala

Abstract: In this report, we will present the methods and tools used to build and apply Q-learning and DQN algorithms (and their variations) in two separate OpenAI Gym environments (Taxi v_3, Lunar_Lander v_2) to demonstrate their effectiveness as model free methods for training goal driven agents. The tasks are navigation based and episodic, but include both discrete and continuous environments. We seek to identify and investigate the most important methods and parameters for improving performance. We achieved our best results with Dueling Double DQN by solving the lunar landing environment in 298 episodes, surpassing our expectations.

Index Terms: Q-learning, Deep Q-Learning, Double Deep Q-Learning, Dueling Double Deep Q-Learning, Taxi V3, Lunar Landing V2

1. Basic

1.1 Domain and Task Description

For this project we will adopt a basic environment for a randomised task-driven agent and set the parameters based on the approach taken in Dietterich (2000) for the taxi pick up environment (in our case the "Taxi_v3") which operates in a 5 by 5 grid. Sutton and Barto (2018) state that reinforcement learning starts with a complete, interactive goal-seeking agent that "have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments."[1]

For the basic portion of our work, we will adopt the third version of the classic taxi task utilizing a pre-existing OpenAI Gym environment. The environment is based on a simple grid but adds additional complications with walls and specific pick up or drop off points. The taxi itself has two states, occupied or empty, and can travel up, down, left or right as well as pick up and drop off (six actions possible). The task is to correctly pick up and drop off passengers at the designated points. Both parts have starting points that are generated at random rather than fixed points. We plan to build on this analysis and create a more advanced environment based on the Lunar Lander OpenAI Atari environment that will also rely on dynamic state assessment by the agent, but in a much more complicated environment with a wider range of possible actions required.

1.2 State Transition and Reward Functions

The state transition function for the taxi task is updated for each of the actions according to the classic Markov Decision Process (MDP), as show in figure 1 below. Sutton and Barto state that MDPs are "a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards."[2] This involves a trade-off between

[1] Sutton and Barto, Reinforcement Learning, p.3.

immediate and delayed rewards. In our case there are a finite set of states (S) in the environment, observable by the agent. Let A be the available actions in a given current state (s). Therefore, as Dietterich (2000) states, the agent makes its transition from s to s' (resulting state) according to the following probability distribution:

$$s:\ P(s' \mid s,a)$$

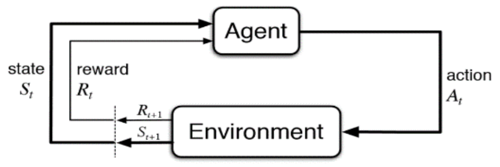


Fig. 1. Classical interpretation of the agent environment interaction in a Markov Decision Process. Source: Sutton and Barto (2018)

In this case, we will be applying the Q-learning algorithm developed by Watkins (1989) which can be represented as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[R + \gamma \max_{a'} Q(S',a') - Q(s,a)\Big]$$

Q-learning is an off-policy temporal difference (TD) control algorithm. This means that the Q values (discussed below) are updated separately from the policy, updated episodically and with some randomisation in the actions. As defined by Sutton and Barto, Q represents the learned action-value function, which estimates the optimal Q separately from the

[2] Sutton and Barto, Reinforcement Learning, p.47

policy. “This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs.”[3]

The state transition function for our purposes depends on the environment we have selected, which is episodic and randomised. The four pickup and drop-off locations (R, B, G, Y) interact with the random start point for the taxi and the passenger pickup/drop-off to create the domain. Each episode ends at a successful drop-off. The different tasks are shown in fig. 2 below.

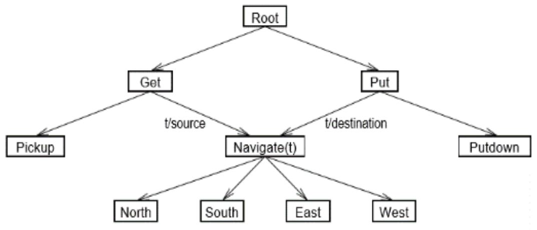
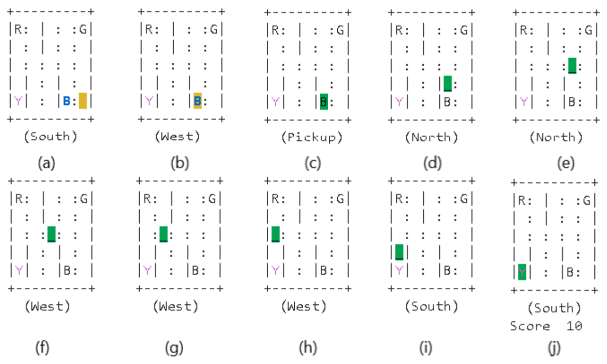


Fig. 2. Task graph for Taxi problem, Source: Dietterich (2000)

There are thus six possible actions for the taxi agent (our action space consists of each of the four navigate options, plus the pickup and putdown) as defined by Dietterich. For our purposes, state S = 500, given that there are 25 squares, 5 locations for the passenger (including the taxi), and 4 destinations. For the rewards, we want to maximize reward for a successful drop off – the goal – and a small penalty. We assign the standard penalty of -1 for each action (movement penalty as used in shortest task functions) and a reward of +20 for successfully delivering a passenger. Incorrect pickups or putdowns were given a 10-point penalty and the time penalty incurs an additional -1 cost for each turn. This should incentivize the agent to find the shortest way to successful drop off.



```
+---------+  +---------+  +---------+  +---------+  +---------+
|R: | : :G|  |R: | : :G|  |R: | : :G|  |R: | : :G|  |R: | : :G|
| : | : : |  | : | : : |  | : | : : |  | : | : : |  | : | : : |
| : : : : |  | : : : : |  | : : : : |  | : : : : |  | : : : : |
| | : | : |  | | : | : |  | | : | : |  | | : | : |  | | : | : |
|Y| : |B: |  |Y| : |B: |  |Y| : |B: |  |Y| : |B: |  |Y| : |B: |
+---------+  +---------+  +---------+  +---------+  +---------+
  (South)       (West)      (Pickup)     (North)      (North)
```

```
+---------+  +---------+  +---------+  +---------+  +---------+
|R: | : :G|  |R: | : :G|  |R: | : :G|  |R: | : :G|  |R: | : :G|
| : | : : |  | : | : : |  | : | : : |  | : | : : |  | : | : : |
| : : : : |  | : : : : |  | : : : : |  | : : : : |  | : : : : |
| | : | : |  | | : | : |  | | : | : |  | | : | : |  | | : | : |
|Y| : |B: |  |Y| : |B: |  |Y| : |B: |  |Y| : |B: |  |Y| : |B: |
+---------+  +---------+  +---------+  +---------+  +---------+
  (West)       (West)       (West)       (South)      (South)
                                                     Score  10
```

Fig. 3. Grid by grid representation of a successful taxi cycle

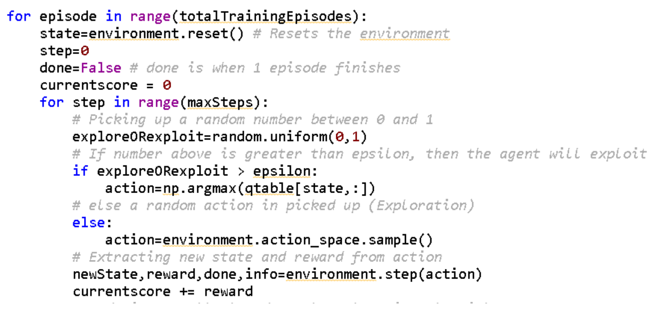
In order to visualise and explain this dynamic we show in the above figure an actual successful loop of a taxi cycle from our trained agent. In cell (a) the taxi is represented as yellow and therefore with no passenger. In (b) it moves to the blue grid square (represented by a capital B), (c) represents a successful pickup and the other grids represents actions chosen to get to the successful drop-off point at cell (j) – the yellow Y square - with the reward of 10 represented in the text. The score of 10 denotes the reward of 20 for successful pickup and delivery of the passenger -10 for cost of the steps taken to achieve this outcome.

1.3 Policy Selection and Details

We are seeking a policy that maximizes the total rewards per episode with the two separate sets of subtasks (the hierarchical element) involving the passenger states. Because of this we have decided that in our case an epsilon greedy policy with an exponential time decay should be sufficient to allow us to converge towards optimal performance in this exercise, because it is not a continuing (Sutton and Barto, 2018) or infinite horizon (Dietterich, 2000) task. In our code this appears as the following (exponential) formula:

```
epsilon=minEpsilon+(maxEpsilon-minEpsilon)*np.exp(-decay*episode)
```

Where epsilon is set at a starting value of one and iterates according to the below if statement that gradually manages down the chance of exploring versus exploitation as the agent learns the optimal paths through the environment over time. We also considered using a Boltzmann (softmax method) but opted to try and replicate Dietterich as closely as possible in the first instance (in that paper a GLIE – greedy in the limit with infinite exploration - policy was used with the MAXQ algorithm). We have opted for our policy applied to a standard Q-Learning table, enacted with code as presented in fig. 4:



```
for episode in range(totalTrainingEpisodes):
    state=environment.reset() # Resets the environment
    step=0
    done=False # done is when 1 episode finishes
    currentscore = 0
    for step in range(maxSteps):
        # Picking up a random number between 0 and 1
        exploreORexploit=random.uniform(0,1)
        # If number above is greater than epsilon, then the agent will exploit
        if exploreORexploit > epsilon:
            action=np.argmax(qtable[state,:])
        # else a random action in picked up (Exploration)
        else:
            action=environment.action_space.sample()
        # Extracting new state and reward from action
        newState,reward,done,info=environment.step(action)
        currentscore += reward
```

Fig. 4. Code for epsilon greedy policy

In Q learning, where the actions and states are defined as above and pi represents the policy, we can perform updates based on the following equation:

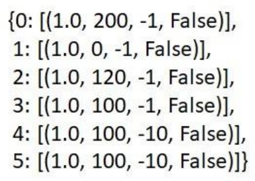
$$Q(S,A) := Q(S,A) + \text{alpha}(\alpha) * [R(S,A) + \text{gamma}(\gamma) * \text{Max } Q(S',A') - Q(S,A)]$$

1.4 Tabular Representation of P Table

Because of the above, we do not have an R-matrix defined mapping all the states with the correspondent reward values. The Q matrix consists of a dictionary object that represents

[3] Sutton and Barto, Reinforcement Learning, p.131

P outputs in each state. This dictionary can be represented in a P table where each action (0 south, 1 north, 2 east, 3 west, 4 pickup, 5 drop-off) is represented with a probability (always set to 1 in this environment), next state, reward, goal state boolean output). The full dictionary can be described as (Rows (States) * Cols (Actions)). Below is an example of the dictionary output for state 100 in our P table.



```
{0: [(1.0, 200, -1, False)],
 1: [(1.0, 0, -1, False)],
 2: [(1.0, 120, -1, False)],
 3: [(1.0, 100, -1, False)],
 4: [(1.0, 100, -10, False)],
 5: [(1.0, 100, -10, False)]}
```

Fig. 5. State 100 P output.

Because the full 500 states would be too large to show, below we have P table values for states 299 and 499 as an

P Matrix		Action	State	Reward
State	299	0	399	-1
		1	199	-1
		2	299	-1
		3	279	-1
		4	299	-10
	⇕			
	499	0	499	-1
		1	399	-1
		2	499	-1
		3	479	-1
		4	499	-10
		5	499	-10

example of this input.

Table 1. P table for selected states at 299 and 499

1.5 Q learning Parameter Value Selection

Our selected parameters for the Q-learning algorithm are as follows:

- Alpha: 0.5 (learning rate)
- Gamma: 0.5 (discounting rate)
- Epsilon: 1 (exploration/exploitation)
- maxEpsilon: 0.99 (maximum exploration)
- minEpsilon: 0.01 (minimum exploration)
- decay = 0.005 (decay rate for epsilon)

The values above were selected as a reasonable midpoint before analysis of results and trial and error test (discussed below). Of course, the epsilon will decrease over time given the time decay component of our epsilon policy. Accordingly, Alpha has been assigned 0.5 to balance efficiency and effectiveness in the learning rate. Meanwhile, gamma has been assigned 0.5, as we get closer to our goal we expect to have to increase the number (focuses on long term rather than short term reward). Epsilon in this iteration is initially set to full exploration, which reduces according to the decay amount. We set training and testing episodes to 2000 and 100, respectively. As the trials and episodes increase, epsilon will decrease automatically rather than us having to set this manually.

1.6 Q Matrix Creation and Learning Updates

The Q matrix for our project is created in the standard way, creating an array of zeros to be updated with Q values in each of the possible outlined above. By contrast, however, the Q value is calculated by updating an old Q value (from s) with the immediate reward from action a and adding the best Q value possible from the next state (gamma is the discount factor which controls importance of future rewards).

More simply, we can say that the agent feeds information back into the Q table to inform future decision making as shown in the following diagram:

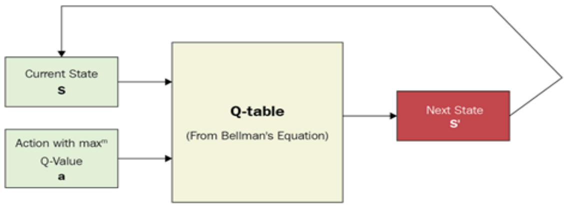


Fig. 6. Q learning update cycle, Source: Habib, 2019

Alpha is the learning rate applied above, setting it low will mean values are never updated, and high will mean that updates completely override the old information. The Q values are updated in accordance with the Bellman equation (rule of optimality). A cautious approach helps preventing overfitting or underfitting in the training process. In our case the method for applying our epsilon greedy training can be according to following algorithm:

```
For each episode:
While the goal state is not reached Do:

A. Set a random value for E between 0 and 1
B. If E > ε (exploit):
   1. The next state will be the one with the highest q
      value according to the argmax of the Q table
   2. Extract new state and reward from action
   3. Where the current state is greater than or equal to
      reward: Update Q-Matrix according to formula
      above:
C. Else (explore):
   1. Randomly chose the next state
   2. Update the Q-matrix as above
   3. Update state to new state
D. If Episode is finished:
   1. Update episode count
   2. Update value of ε according to exponential time
      decay:
```

```
epsilon=minEpsilon+(maxEpsilon-minEpsilon) * np.exp(-decay*episode)
```

Table 2 displays the updated values of the Q matrix at states 299 and 499 in one of our experimental runs, as an example of how these values change.

Table 2. Updated Q values based on our method for selected states

1.7 Results and Experimentation

The agent initially showed a slow and steady development towards convergence (in our case defined as a regular positive number for the cumulative rewards), reinforced by the effect of time decay on the agent learning process (a validation of our selection of the exponential epsilon with time decay policy).

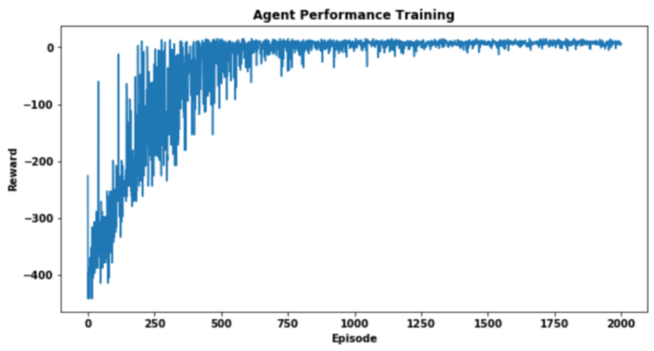


Fig. 7. Agent performance based on our initial parameters of alpha = 0.5 and gamma = 0.5, and exponential epsilon time decay policy

We can see in figure 7 a few negative reward spikes that were observed between episodes 500 and 1000, likely a result of the residual greedy epsilon policy. This was confirmed when we replaced the exponential policy with a linear one (as discussed below). The agent gained an average reward of -68.69 over the entire training cycle. For completeness, we present the results of our manual grid search over the parameter values below. In the first section, one can see that where alpha is constant higher gamma values reap higher average rewards in our training run. The same is true when the gamma is held constant for the Alpha value iterations, although the increase in average rewards slowed significantly between 0.75 and 1. In any case, the best results we found for both values together were at 0.75 for each parameter, with the exponential gaining over twice as many rewards on average at that point.

Gamma (Alpha@0.5)	Avg Reward – Linear	Avg Reward – Exponential
0	-166.47	-130.50
0.25	-78.00	-41.82
0.50	-68.69	-36.46
0.75	-64.11	-32.04
1	-64.03	-31.70
Alpha (Gamma@0.5)		
0	-166.81	-129.76
0.25	-91.55	-51.87
0.50	-69.22	-36.50
0.75	-61.77	-28.50
1	-59.36	-27.99
Gamma@0.75 Alpha@0.75	**-58.26**	**-27.92**

Table 3. Gamma and alpha values, training and test average reward results

The above table shows the results of our experimentation with the Q learning agent. All other variables have been kept constant. The gamma in our case can be understood as the weighting given to future rewards. The higher the gamma, the greater the long-term reward strategy of the agent's decision if it is low it gives a greater weight to immediate rewards (greedy), and if it is high greater importance is attached to the overall rewards as defined by Sutton and Barto in the previous section. From analysis of the results, we can observe that indeed, the higher gamma values generally perform better with the optimal value being reached at 0.75. The alpha, meanwhile, is the learning rate, with more frequent updates performed at higher numbers. In our case we can see that frequent updates do indeed perform best without significantly affecting the computational burden (the training cycle was still easily done on our home computers).

Analysis of the agent performance in the test phase shows that it converges almost instantly and rewards are positive throughout, with the variation restricted to a limited range in the positive reward band. It is generally consistent, which becomes clear when it is plotted against both training methods. We can clearly see the importance of policy selection for the performance of the agent. In the figure below, we have shown the erosion of the greedy epsilon number in both the exponential and linear case. In short as the exploration value reduces much earlier in our exponential case, the training reward accumulation has improved. This could mean that our environment was insufficiently complex and therefore more sophisticated exploration methods (policies) would be unnecessary. Our decay value of 0.005 within the prescribed ranges of an epsilon maximum (starting) value of 0.99 and epsilon minimum value of 0.01 seemed sufficient for the agent to reach the optimal state. In fact, we can see from Figure 8 that the performance of both agents reaches optimal well before episode 918, when the minimum epsilon was achieved.

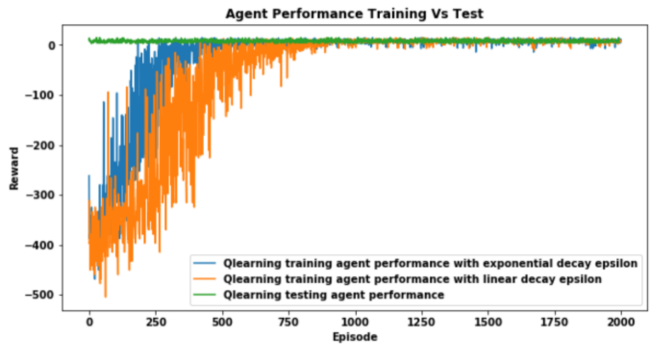


Fig. 8. Agent performance Comparison based on tuned parameters

Q Table		Action					
		(0) South	(1) North	(2) East	(3) West	(4) Pickup	(5) Drop off
States	0	0	0	0	0	0	0
	⇕	⇕	⇕	⇕	⇕	⇕	⇕
	299	-0.655	-1.416	-1.103	-0.280	-7.556	-9.506
	⇕	⇕	⇕	⇕	⇕	⇕	⇕
	499	-0.850	-0.320	-0.850	-0.750	-7.500	-5.000

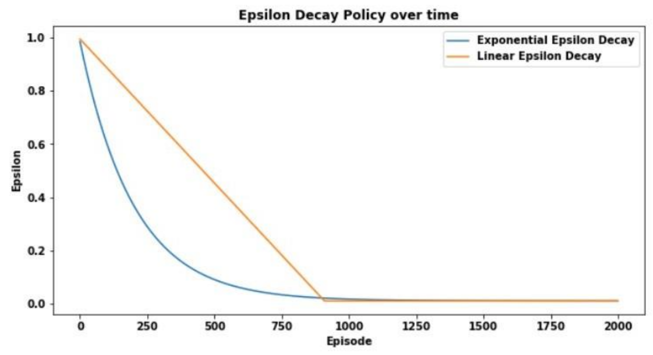


Fig. 9. Epsilon decay exponential vs linear

In conclusion, we can say that our Q-learning exercise was successful in training agents to learn and solve the environment in the optimal manner, but that our selection of mid-range parameters for gamma and alpha could have hindered the speed of agent learning. In the end, it is also possible that a less conservative epsilon reduction policy would also help improve the overall learning speed, albeit at an additional risk of further inflating positive (maximization) bias – a known risk of Q-learning and Sarsa-based control algorithms. We will proceed in the next section to examine more advanced methods of controlling this bias in a more complicated environment.

2. Advanced Exercise: Deep Reinforcement Learning Based Methods

2.1 Introduction to Deep Learning Algorithms

We decided to follow our analysis of standard Q-learning with analysis of Deep Q networks (DQN) in a more complex Atari environment on the OpenAI platform – the lunar lander. The DQN method was created by Silver et al. in 2013. The algorithm was created to apply deep learning and neural network techniques to Q-learning for agents to function in more complex environments. Rather than updating a Q table or dictionary of possible states, a neural network is used to infer or predict possible outcomes. There is therefore a training phase for the model rather than updating the table. This is used via regression calculated from the outputs of a neural network with in our case looks like the network represented below.

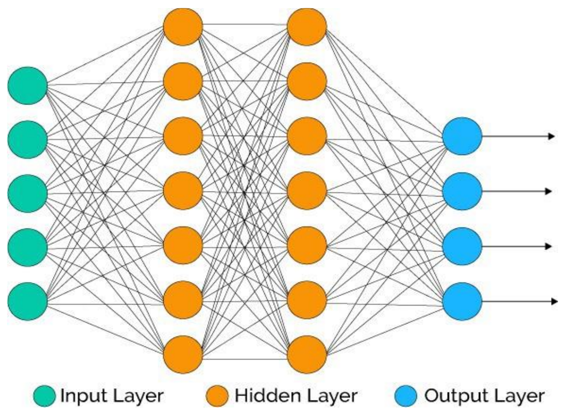
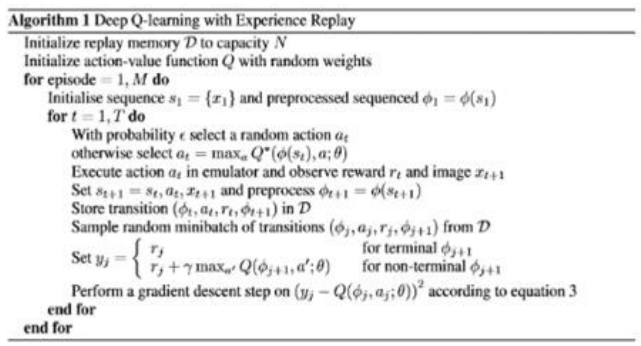


Fig. 10. DQN neural network architecture example

The learning rate is multiplied by the reward, discount factor and estimate of optimal future value to derive values for the updates. In addition, Mnih et al. (2013) pioneered the approach of using experience replay where the agent's experiences at each time-step are stored in a dataset pooled over many episodes into a replay memory. During the inner loop of the algorithm (below) mini-batch updates are used to take random samples of experience which the agent uses to select actions according to a chosen epsilon policy (in this case greedy). One limitation of this method is that the histories used must have a fixed length (to prevent computational issues).



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      end for
  end for
```

Fig. 11. DQN algorithm with experience replay (Mnih et. Al, 2013)

It is well established that Q learning methods suffer from overestimation bias as the same max and argmax functions are used to calculate and update Q values, creating an issue that accelerates in more complex environments. In 2015, Hasselt et al suggested an alternative method that requires two separate calculations for values (according to the equation below), one for action and one for value estimation. These will alternate in order to reduce the risk of overestimation based on the pure max functions displayed before, without being fully separated. Hasselt et al redefined the selection and evaluation process in Q learning in the following way, in order to create a comparison point with the different but related method in double Q-learning.

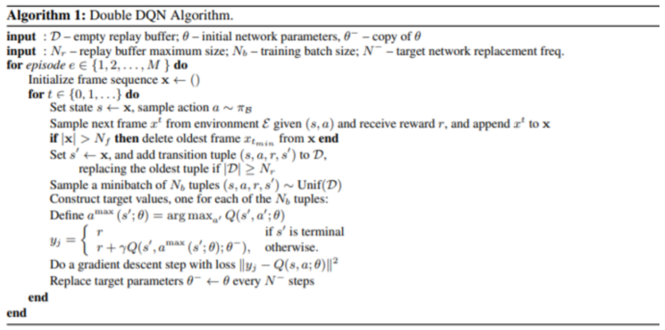
$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

The Double Q learning error was then presented as:

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)$$

In that paper, Hasselt et al commented that, "This version of Double DQN is perhaps the minimal possible change to DQN towards Double Q-learning. The goal is to get most of the benefit of Double Q-learning, while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead."

As a result, the double DQN is largely similar with the exception of the use of the target network, and can be expressed in the following way:



```
Algorithm 1: Double DQN Algorithm.
input : $\mathcal{D}$ – empty replay buffer; $\theta$ – initial network parameters, $\theta^-$ – copy of $\theta$
input : $N_r$ – replay buffer maximum size; $N_b$ – training batch size; $N^-$ – target network replacement freq.
for episode $e \in \{1, 2, \ldots, M\}$ do
    Initialize frame sequence $\mathbf{x} \leftarrow ()$
    for $t \in \{0, 1, \ldots\}$ do
        Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_B$
        Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append $x^t$ to $\mathbf{x}$
        if $|\mathbf{x}| > N_f$ then delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ end
        Set $s' \leftarrow \mathbf{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$,
            replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
        Sample a minibatch of $N_b$ tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
        Construct target values, one for each of the $N_b$ tuples:
        Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
        $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
        Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
        Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
    end
end
```

Fig. 12. Double DQN algorithm with experience replay (Wang et. Al, 2015)

Another alternative approach posited by Wang et al in 2015 is to create a so-called dueling architecture. This works creating value pairs for the states and actions that are individually calculated via the other methods. This is achieved by way of two separate calculations, one for the state value and advantage (the relative importance of each action). These can be calculated and merged in the following way:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$$

Where V is the state value and A is the advantage. Dueling architectures require two separate neural network outputs, one calculates a scalar and the other a vector, and the two values are merged. Visual representations of the two different networks are displayed below. Wang et al. state that, "As the dueling architecture shares the same input-output interface with standard Q networks, we can recycle all learning algorithms with Q networks…", in our case this means that there is a direct compatibility, and similarity, between the algorithms of our three chosen methods for training in our environment. We can expect a significant improvement sequentially as we apply the three methods.

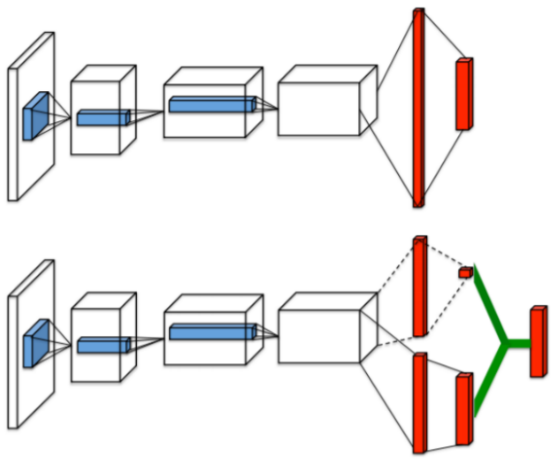


Fig. 13. DQN vs. Dueling DQN neural network architectures, Wang et al. (2015)

2.2 Environment, State and Reward Functions

For our (relative to the taxi exercise above) more complicated environment based on these methods we have selected the Lunar Lander v2 environment from OpenAI Gym. The lander is controlled by engaging thrusters at the cost of using fuel, and the goal is for the agent to land on randomly generated moon surfaces.

The state space can be defined as an eight-dimensional vector: horizontal and vertical position of agent, horizontal and vertical velocity of agent, orientation, angular velocity, and left and right leg binary flags indicating contact with the surface. There are four possible actions for the agent in the environment: no action, push up (main engine), push right (orientation) and push left (orientation). The lander starting position is at the top of the screen with a random velocity and should land at the same point (0,0). The episode ends when the lander crashes (represented in the code by another binary flag).

The reward policy grants or subtracts 100 points for a successful landing or crash respectively. Leg contacts give 10 points each, and a fuel consumption penalty of -0.3 points is incurred for each positive action (every time one of the engines is fired). We have calibrated our convergence threshold in a standard metric found in the literature (for example in Yu, 2019) of an average score of 200 over the last 100 episodes.

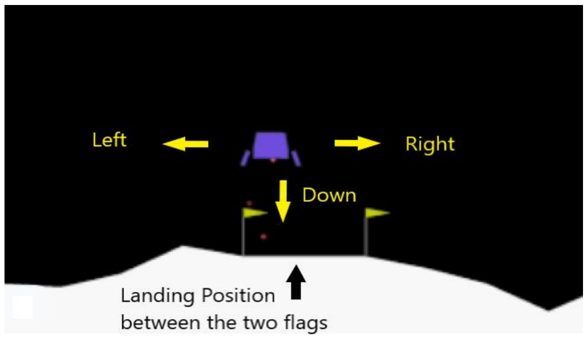


Fig. 14. Lunar Landing screenshot, landing and in flight, flags denote landing zone

2.3 Parameter Training

The values for the initial training runs for our DQN agents were set as follows: the maximum number of training episodes was set at 1500, maximum timesteps per episode set at 1000, the minimum and maximum range for epsilon set between 0.01 and 0.99 respectively, epsilon decay rate set at 0.005 and tau (soft target update calculation) set at 1e-3 (0.001), rather than the 0.01 used by Silver et al (2016) in a similar exercise. Soft updates (often expressed by the tau value) are used to negate weights and biases that stem from updates and resets of the original Q values (called the target network) to negate the effect of changes in the weight of the network and goals over time (possibly obstructing convergence), the idea behind a soft update is to assign a small number and update the target network more frequently, as shown below.

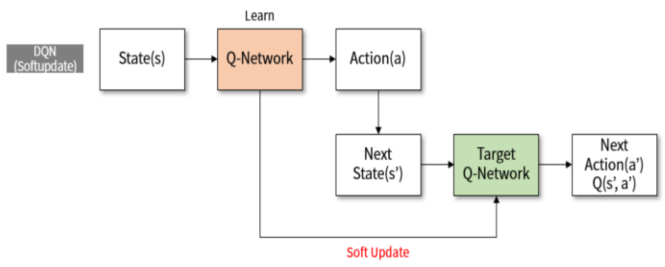


Fig. 15. Structure of learning to soft update target network in DQN Source : https://greentec.github.io/reinforcement-learning-third-en/

We have kept these consistent throughout the following experiments, varying them where explicitly described. In the context of other Atari playing environments, the Lunar Lander environment can be characterised as lower dimensional (8 states) and therefore rather than use the fully convolutional neural networks used in many other cases, we have opted for two fully (densely) connected layers set at 64 nodes generating four outputs approximating values from the eight inputs, using the Adam optimizer.[4] First developed in 2014 by Kingma and Ba, adaptive moment estimation (Adam) is an adaptive stochastic method for varying learning rates assigned to weights in a network in a manner similar to stochastic gradient descent with momentum. Learning rates are calculated for different parameters.

2.4 Results and Performance Evaluation

In the first instance, we created two separate training runs varying the epsilon policy between linear and exponential with time decay. Our hypothesis was that the additional calibration between exploration and exploitation would avoid local minima and create a more efficient training path to convergence (similar to what we observed in the taxi environment). The results of the exercise are displayed below:

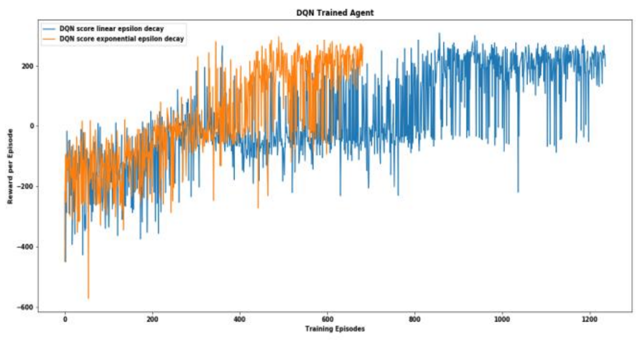


Fig. 16. DQN agent, linear and epsilon decay

We can see a dramatic difference in the training experience for the agent between linear and exponential decay, with the environment solved in 1136 versus 582 episodes respectively. The DQN agent with exponential epsilon decay solved the environment in just under half the time, outperforming our expectations given comments in Yu (2019) and others regarding the relative insensitivity of lunar lander experiments to changes in parameter and policy rates. This mirrors the effect and results of our experimentation in the Q-learning taxi environment.

Due to the lack of other hyperparameter tuning, we thought we might be witnessing a residual effect of maximisation bias in our model, and decided to test this out by applying the double DQN methodology to an agent trained on the same basis. The results of this exercise are shown in Figure 17.

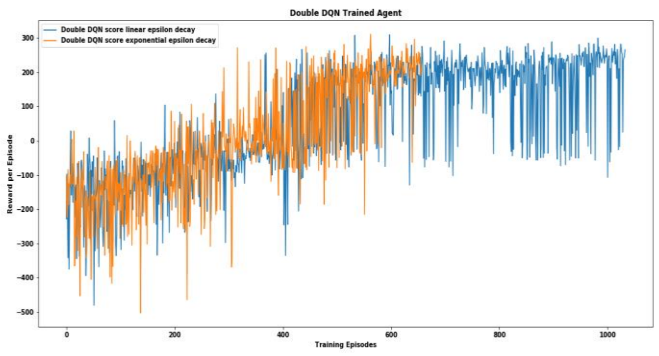


Fig. 17. Double DQN agent, linear and epsilon decay

In this case, the double DQN performed markedly better, as expected, solving the environment in 993 and 557 episodes. This is still below our success threshold of 500 episodes, so we will try using a dueling DQN architecture to decouple and then merge our state value and advantage values – allowing us to
only retain relevant performance metrics and increase learning efficiency beyond simply managing out our maximisation bias. This was expected to be our best performing variation and this was indeed the case:

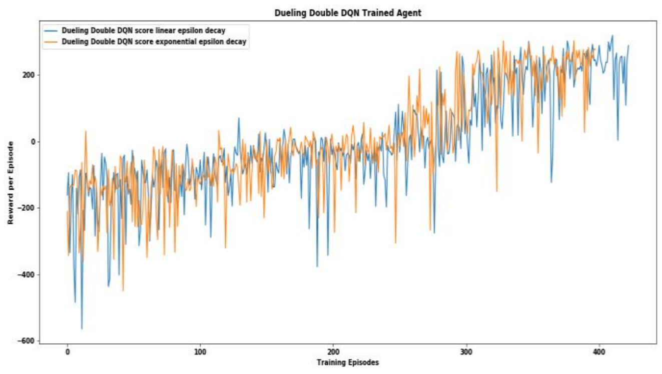


Fig. 18. Dueling Double DQN agent, linear and epsilon decay

The Dueling DQN solved the environment in 323 and 298 episodes in the linear and exponential policy agents.

4 P. 605, Jansen, S., n.d. Hands-On Machine Learning for Algorithmic Trading.

Interestingly, while we have managed to beat our expectations, the updated and latest also had the effect of drastically reducing the difference between our decay learning policies – bringing it more in line with our initial expectations. This could be further reduced in our estimation by refining gradient clipping or policy prioritization in our algorithms. Nevertheless, we can say with certainty that even for our relatively simple environment, the dueling architecture alone with minimal parameter tuning displayed by far the best performance in steps to our stated goal, and beating our reward threshold.

One other interesting conclusion is that in terms of average reward (where the double DQN only overtook our standard DQN agent at around 400 episodes) the dueling DQN agent learned much faster and consistently performed far better at a much earlier stage even with a linear epsilon policy (see Figure below).

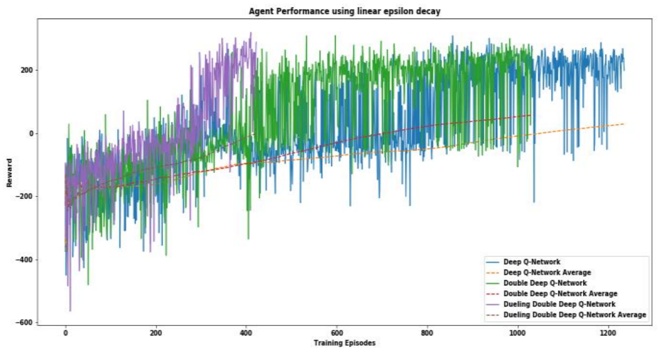


Fig. 19. Rewards and average rewards for selected algorithms for linear epsilon decay

It can also be stated that the relative performance of our selected algorithms (measured by steps to convergence) held true in both the linear and exponential policy states.

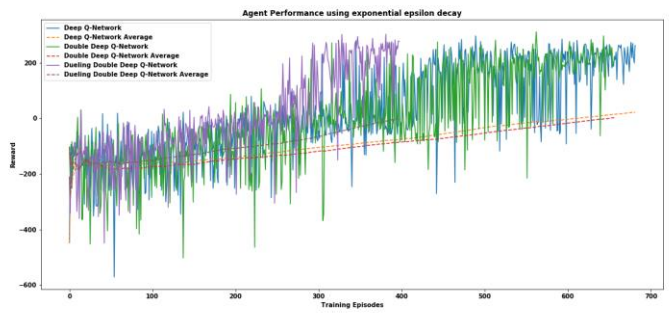


Fig. 20. Rewards and average rewards for selected algorithms for exponential epsilon decay

Analysing our results for the exponential epsilon alone, we observed that average rewards across all of our selected algorithms escalated much faster and performed at a smoother rate, justifying the use of our exponential formula. Once again, the difference in learning rate and consistency was more than we expected given the environment – we would expect this to increase greatly in more complex 3D environment with a greater range of actions.

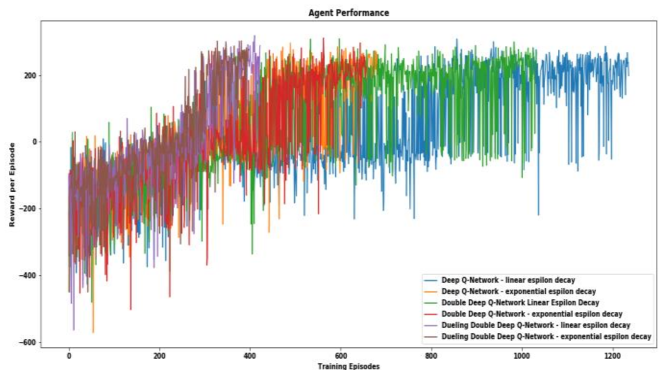


Fig. 21. All algorithms, all policies, absolute rewards per episode

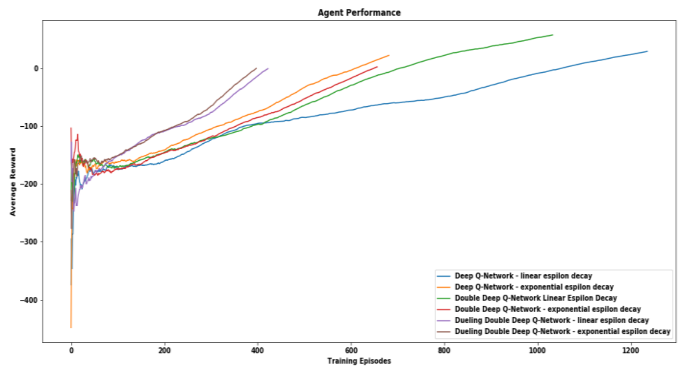


Fig. 22. Average reward for all algorithm combinations

Figure 22 above shows the average rewards reaped by the agents across the algorithms and different policies – despite the outperformance overall of our duelling architectures we can make some generalised observations on the performance of algorithms with certain policies attached. In general, the slowest learners and slowest to reach our convergence threshold were the linear policies (in line with our conclusions from the Q learning exercise). We can also see that in terms of rewards gained, there is an interesting dynamic in the earliest episodes, our double Q networks did have an early burst, even overtaking duelling before rewards gained declined. In both figures 21 and 22 one can easily see how much faster the duelling DQN agents were at reaching our optimal threshold than the alternatives (the point at which the lines stop).

Algorithm 1 Double DQN with proportional prioritization

```
1: **Input:** minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
2: Initialize replay memory $\mathcal{H} = \emptyset, \Delta = 0, p_1 = 1$
3: Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
4: **for** $t = 1$ **to** $T$ **do**
5:     Observe $S_t, R_t, \gamma_t$
6:     Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
7:     **if** $t \equiv 0 \mod K$ **then**
8:         **for** $j = 1$ **to** $k$ **do**
9:             Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
10:            Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
11:            Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
12:            Update transition priority $p_j \leftarrow |\delta_j|$
13:            Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
14:        **end for**
15:        Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
16:        From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
17:    **end if**
18:    Choose action $A_t \sim \pi_\theta(S_t)$
19: **end for**
```

Fig. 23. Algorithm for double DQN with prioritized experience replay

We could further optimize our agent's ability to solve the environment by applying prioritized experience replay as suggested by Schaul et al (2016). In this case, we woulld use the TD-error to categorize and rank experiences, using this memory dataset to generate minibatches to be sampled by the agent, according to the algorithm in figure 23. Relatively speaking, looking at both the actual episode rewards and average, the exponential greedy policy with time decay performed similarly for DQN and double DQN, meaning that there is little to choose between those two agents when trained with our optimal parameters. Some further experimentation with less optimal parameters might be warranted to see why this is the case. The observation does not stand in the linear case, where the standard DQN algorithm does noticeably lag behind all other options.

3. Conclusions and Future Work

We achieved our best results with Dueling Double DQN by solving the lunar landing environment in 298 episodes. If we were to take this work further, we would try to generalise our trained models in other Atari Learning Environments (ALE). In order to do this, we would take the approach taken by Mnih et al (2013) and many others in applying reward clipping and seeing how this would affect our agents. We would initially apply this experimentally to fully understand the effect this would have in our environment before widening it out further to the full list of 50 games commonly used for testing in Deep Mind papers.

Finally, we would add Rainbow DQN to our range of algorithms used to include a state-of-the-art application to our exercise. This uses elements of double DQN, duelling DQN and prioritized replay along with multi-step learning and noisy nets. We would be interested in testing the expected improved performance with the rainbow DQN with an ablation study of the type applied by Hessel et al. (2017).

4. References

Dietterich, T., 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. Journal of Artificial Intelligence Research, 13, pp.227-303.

Haykin, S., 2009. Neural Networks and Learning Machines, Pearson International Third Edition.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M., 2013. Playing Atari with Deep Reinforcement Learning, Deep Mind.

Hasselt, H., Guez, A., Silver, D., 2015. Deep Reinforcement Learning with Double Q-Learning, Deep Mind. https://arxiv.org/pdf/1509.06461.pdf

Kingma, D., Ba, J., 2015. Adam: A Method for Stochastic Optimization, ICLR, arXiv:1412.6980

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., 2015. Human-level control through deep reinforcement learning. Nature, 518(7540), pp.529-533.

Goodfellow, I., Bengio, Y. Courville, A., 2016. Deep Learning, MIT Press.

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D., 2016. Continuous Control with Deep Reinforcement Learning, ICLR.

Schaul, T., Quan, J. Antonoglou, I., Silver, D., 2016. Prioritized Expérience Replay, ICLR. https://arxiv.org/pdf/1511.05952.pdf

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. Freitas, N., 2016, Dueling Network Architectures for Deep Reinforcement Learning, Deep Mind, https://arxiv.org/pdf/1511.06581.pdf

Hessel, M., Modavil, J., Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D., 2017. Rainbow: Combining Improements in Deep Reinforcement Learning, Deep Mind. https://arxiv.org/pdf/1710.02298.pdf

Jansen, S., 2018. Hands-On Machine Learning for Algorithmic Trading, Packt Publishing.

Palanisamy, P., 2018. Hands-On Intelligent Agents with OpenAI Gym, Packt Publishing.

Sutton, R. and Barto, A., 2018. Reinforcement Learning: An Introduction, MIT Press 2nd Edition.

Habib, N., 2019. Hands-On Q-Learning with Python, Packt Publishing.

Yu, X, 2019, Deep Q Learning on Lunar Lander Game, ResearchGate.