Exploring the Q-learning Algorithm in Basic and Advanced Environments

Elnara Mammadova
Department of Computer Science
City, University of London
London, United Kingdom
Elnara.Mammadova@city.ac.uk

Jerome Titus
Department of Computer Science
City, University of London
London, United Kingdom
Jerome.Titus@city.ac.uk

***Abstract*— Q-learning is an off-policy reinforcement learning technique used to find an optimal policy through interaction with an unknown environment [1]. This paper aims to examine and critically evaluate a battery of Deep Reinforcement Learning (DRL) techniques in different scenarios, each increasing in order of complexity. We present and evaluate convergence results for (i) simple Q-learning algorithm using a custom-built environment, (ii) DQN, with Double DQN and Dueling DQN improvements in the Open AI Gym environment using a customized neural network built via PyTorch and (iii) DQN Trainer for the Atari 2600 game environment using the Ray RLlib library.**

Keywords—Deep Reinforcement Learning, Q-learning, DQN, Double DQN, Dueling DQN, OpenAI Gym, Atari, RLlib

I. INTRODUCTION

Q-learning provides a means for agents to act optimally through experiential learning. When an agent executes an action, it receives immediate feedback in the form of a reward. Through continuous interaction with its environment, it learns the optimal sequence of actions to implement, judged by long term discounted rewards [2].

The first part of this paper presents a verification of convergence of the Q-learning algorithm in a custom-built environment - using two separate policies for comparison. The second part of this paper will evaluate the results from a more complex Q-learning algorithm structure; Deep Q-Networks (DQN) with two improvements (Dueling DQN, Double DQN) in an OpenAI Gym environment (Cartpole-v1). The third part of the paper will experiment with a more complex Atari 2600 game environment, where we will utilize the Ray Tune RLlib library to produce and evaluate our results.

II. UBER POOL PROBLEM

A. Environment and Problem Definition

The first environment of study is a simplified version of the real world "Uber Pool" task using a custom built environment. The first task was to construct a city grid using a modified version of Prim's algorithm - a perfect maze generation technique [3]. The environment is a $x \times y$ grid (the first environment experimented on is a 25×25) consisting of obstacles (marked as "■") representing surrounding grid boundaries and city buildings within the grid space, and free cells (marked as " . ") representing roads (Fig 1). There are five designated locations in the Uber Pool environment indicated by two passenger locations (marked as "P"), two destination locations (marked as "D") and one car location (marked as "C")

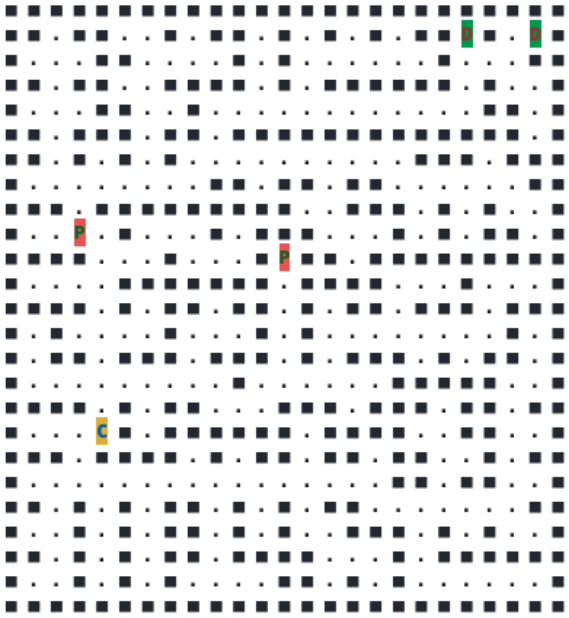

Fig. 1. Custom built Uber Pool environment (*P – Passengers, D – Destinations, C – Car location*).

The defining characteristic of the Uber Pool problem is for the agent (car driver) to pick up both passengers before dropping them off at their respective destination cells in the shortest time possible. While doing so, the agent needs to avoid hitting any obstacles, nor perform any illegal pick-up or drop-off actions (e.g. attempt to pick-up and drop-off at wrong cell, drop-off before both pick-ups were completed successfully etc.). Termination occurs either when both passengers are successfully dropped-off, or when the agent runs out of time ("$time\ limit$" = "$environment\ size$" $\times$ 2).

B. State and Action Space

This custom city grid environment is discrete, and its state space corresponds to the (x,y) coordinates on the grid space, coupled with the 7 scenarios which we will call sub-spaces defined below – corresponding to a total state space of "$environment\ size$" $(width \times height) \times 7$:

- Sub-space 1: (State indices [0, 624])
  Agent has not picked up either passenger
- Sub-space 2: (State indices [625, 1249])
  Agent has picked up passenger 1, but not passenger 2
- Sub-space 3: (State indices [1250, 1899])
  Agent has picked up passenger 2, but not passenger 1
- Sub-space 4: (State indices [1900, 2549])
  Agent has picked up both passengers
- Sub-space 5: (State indices [2550, 3199])

Agent has dropped passenger 1, but not passenger 2

- Sub-space 6: (State indices [3200, 3849])

Agent has dropped passenger 2, but not passenger 1

- Sub-space 7: (State indices [3850, 4499])

Agent has dropped both passengers at their respective destinations

Figure 2 below is the graphical representation of our states, with the possible transitions between sub-spaces.

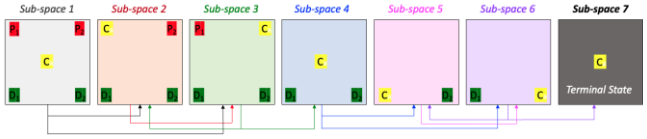

Fig. 2. State space with its sub-space components and the possible transitions between them represented by arrows.

The action space for this environment is defined by the agent's navigational ability in addition to the picking up and dropping off passengers:

['north, 'south, 'west, 'east, 'pick-up', 'drop-off']

$Action\ Space \longrightarrow Discrete(6)$

C. State Transition and Reward Functions

Uber Pool is a deterministic environment, with state transition probabilities defined by:

$$P(S_{t+1}|s_t, a_t) = \begin{cases} 1 \\ 0 \end{cases}. \quad (1)$$

The factors contributing to the 0 and 1 probabilities include the current and next states, as well as the action being executed by the agent. This transition function was implemented using embedded for-loops. The first loop iterated through the individual scenarios (sub-spaces) illustrated in the Fig. 2, with the last two iterating through the coordinate system within each scenario.

The reward function provides our agent with a numerical signal which it will aim to maximize to find the optimal policy. We want our agent to find the fastest route, by providing a reward of -1 per timestep. A reward of -5 will be obtained if the agent 'bumps' into a wall or boundary. If the action executed by the agent corresponds to 'pick-up' it will receive a reward of $+"environment\ size"\ (w \times h)/2$ if it is in the correct passenger location and scenario. Otherwise, it will receive a -10 reward for executing this action. In a similar fashion, the agent will receive a reward of $+"environment\ size"\ (w \times h)$ if the agent executes the 'drop-off' action in the correct position, and -10 if not.

D. Setting up Q-Learning

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning defined by Bellman's equation [1][4]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]. \quad (2)$$

where;

α - learning rate [0-1] – controls how large are the updates to the Q-values

R – reward value – the immediate reward acquired after performing an action (A_t)

γ – discount factor [0-1] – determines how much an agent considers rewards in the distant future relative to those in the immediate future

(S_t, A_t) – action state pair for current state

(S_{t+1}, a) – action state pair for next state

As our agent navigates the city grid environment and observes the next state S_{t+1} and reward R_{t+1}, it will update the Q-value matrix. The Q value function directly approximates the optimal state-action value independent of the policy being followed. The Q-learning algorithm is shown below in procedural form (Fig. 3).

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal

Fig. 3. Procedural form of Q-learning algorithm [5].

Following the procedural form, we first initialize our parameters for the learning rate (α) to 0.1 and discount factor (γ) to 1. A higher learning rate means we update our Q-values in bigger steps, while lower learning rates correspond to smaller updates. The magnitude of the discount rate on the other hand will determine how far sighted we want our agent to be (i.e., how much it cares about rewards in the distant future relative to those in the immediate future). We then initialize the environment, iterating through each step of an episode and choosing actions from a given policy (we will be using both Decaying Epsilon Greedy and UCB (Upper Confidence Bound) policies). We take an action proposed by our policy and observe the reward and the next state, which we then use in our td-update rule (equation 2). The Q-learning algorithm then updates a state–action pair, maximizing over all actions available in the next state. Finally, we set the current state as the next state and continue the loop until a terminal state is reached.

One of the simplest policies used in reinforcement learning is the Epsilon Greedy policy, where an agent takes actions using a greedy policy with a probability of $(1 - \epsilon)$ and a random action with a probability of ϵ (epsilon). We apply a decay rate which slowly decays epsilon over time which ensures the agent moves from exploration to exploitation, ultimately converging to an optimal policy. We set the initial value of both epsilon (parameter that controls the exploitation-exploration trade-off), and decay to 0.9.

UCB policy on the other hand choses an action with a probability that remains constant rather than performing exploration by simply selecting an arbitrary action. It is a deterministic algorithm that changes its exploration vs exploitation balance based on a confidence boundary that the algorithm assigns to each action on each round of exploration, as it gathers more knowledge of the environment. For actions that have been tried the least, the agent explores instead of concentrating on exploitation, selecting the action with the highest estimated reward.

$$A_t = \underset{a}{\operatorname{argmax}}\left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}\right]. \quad (3)$$

where;

$Q_t(a)$ - is the estimated value of action 'a' at timestep 't'.
$N_t(a)$ - is the number of times that action 'a' has been selected, prior to time 't'.
c - is a confidence bound that controls the level of exploration.

The first part of the equation 3, $Q_t(a)$ by itself represents exploitation where the action that currently has the highest estimated reward will be the chosen action. The second part of the equation $c\sqrt{\frac{\log t}{N_t(a)}}$ is where we add exploration, with degree of exploration being controlled by the hyper-parameter c. $N_t(a)$ is the number of times that action a has been selected, prior to time t. This value will be small for scenarios where an action hasn't been tried enough times or not at all. The confidence in our estimate increases as $N_t(a)$ increments, as time progresses the exploration term gradually decreases (since $N_t(a)$ goes to infinity, $\log t \,/\, N_t(a)$ goes to zero, until eventually actions are selected based only on the exploitation term - $Q_t(a)$. The UCB only requires one parameter c which we will set to 1.

E. Training and Performance Evaluation

The initial environment chosen to conduct experiments and training was a 25x25 city grid environment. First, we conduct a single experiment without training our agent. Then we run both policies and evaluate their performances in terms of the timesteps taken and total rewards accumulated. As seen in the table 1 below, both policies converge at optimal policy in the shortest timesteps (73) possible, acquiring the maximum reward value available for the environment given the initial state (1802.0).

TABLE I. PERFORMANCE EVALUATION RESULTS

	Timesteps taken	**Total accumulated reward**	**CPU times**
Untrained	1250 *(time-limit)*	-7247.5	850 ms.
Epsilon-Greedy	73	1802.0	29 sec.
UCB	73	1802.0	38.3 sec.

At closer inspection however, when evaluating the training performance for both policies, we can see that UCB policy converges at around ~700 episodes while our Epsilon Greedy policy takes a little longer, around ~900 episodes (Fig. 4 and Fig. 5).

F. Hyper-parameter tuning

Grid search was implemented using the hyper-parameter ranges indicated in table 2. For the Epsilon Greedy policy, we perform hyper-parameter tuning on the learning rate, discount factor, epsilon, and decay. For UCB policy the hyper-parameters chosen for tuning are the learning rate, discount factor and confidence.

For Decayed Epsilon Greedy policy five hyper-parameter combinations (table 3) and for UCB policy three hyper-parameter combinations (table 4) resulted in the maximum cumulative rewards with varying TD-error values. Results are shown in Figures 6 and 7.

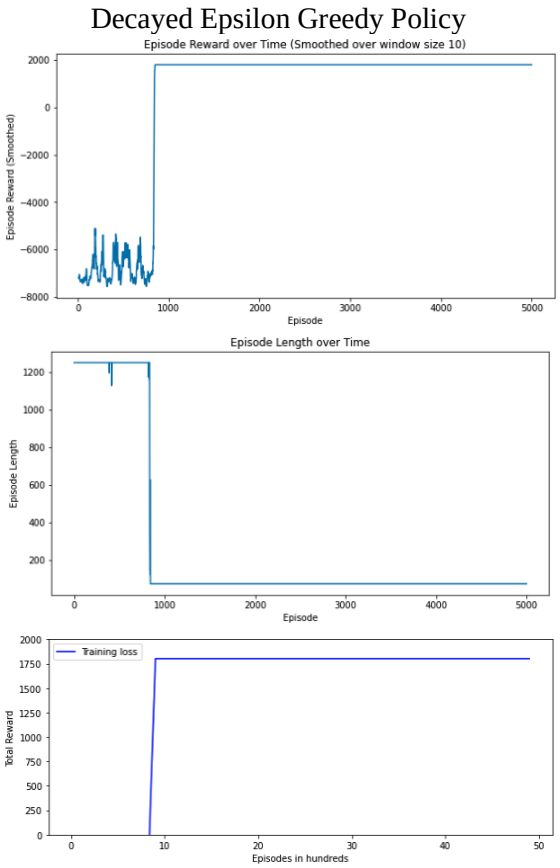

Fig. 4. Training Results for Decayed Epsilon Greedy Policy.

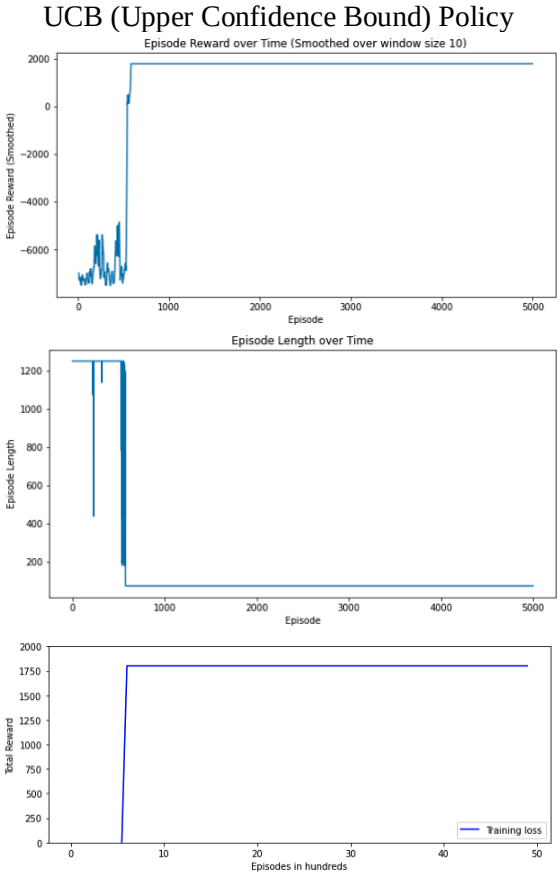

Fig. 5. Training Results for UCB Policy.

TABLE II. CHOICE HYPER-PARAMETER COMBINATIONS FOR GRID-SEARCH

Decayed-Epsilon-Greedy Policy	UCB (Upper Confidence Bound) Policy
learning-rates (α) = [0.01, 0.1] discount-factors (γ) = [0.9, 1.0] epsilon (ϵ) = [1.0, 0.9, 0.8, 0.7] decay-rates (d)= [0.9, 0.99]	learning-rates (α) = [0.01, 0.1] discount-factors (γ) = [0.9, 1.0] confidence (c) = [0.5, 1.0, 1.5, 2.0]

TABLE III. BEST HYPER-PARAMETER COMBINATIONS FOR DECAYED EPSILON GREEDY POLICY

α	γ	ϵ	d	Reward	TD-Error
0.1	1.0	1.0	0.9	1802.0	77291.90
0.1	1.0	1.0	0.99	1802.0	187646.58
0.1	1.0	0.9	0.9	1802.0	64701.36
0.1	1.0	0.8	0.9	1802.0	74835.64
0.1	1.0	0.8	0.99	1802.0	93951.59

TABLE IV. BEST HYPER-PARAMETER COMBINATIONS FOR UCB POLICY

α	γ	c	Reward	TD-Error
0.1	1.0	1.0	1802.0	72404.40
0.1	1.0	1.5	1802.0	81961.79
0.1	1.0	2.0	1802.0	92375.05

Hyper-parameter tuning for Epsilon Greedy Policy

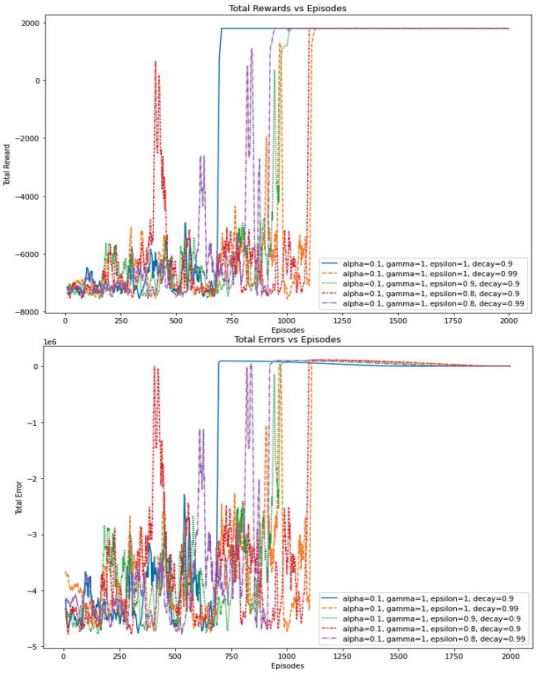

Fig. 6. Hyper-parameter tuning for Decayed Epsilon Greedy policy based on the five best performing hyper-parameter combinations acquired through grid search.

Hyperparameter tuning showed optimal results for learning rate and gamma values of 0.1 and 1.0, respectively. The lower learning rate suggests smaller updates to the Q-values. This is important as larger rates may converge to sub-optimal policies. For example, it was observed that larger learning rates corresponded with the agent picking up the passenger to the left of the car first (see Fig. 1), and then the passenger on the right. While this completes the task, it does not do so in the shortest time possible. This also explains the gamma value of 1.0. To ascertain the quickest route, the agent must consider all future rewards with equal weightage to consider the time factor.

Hyper-parameter tuning for UCB Policy

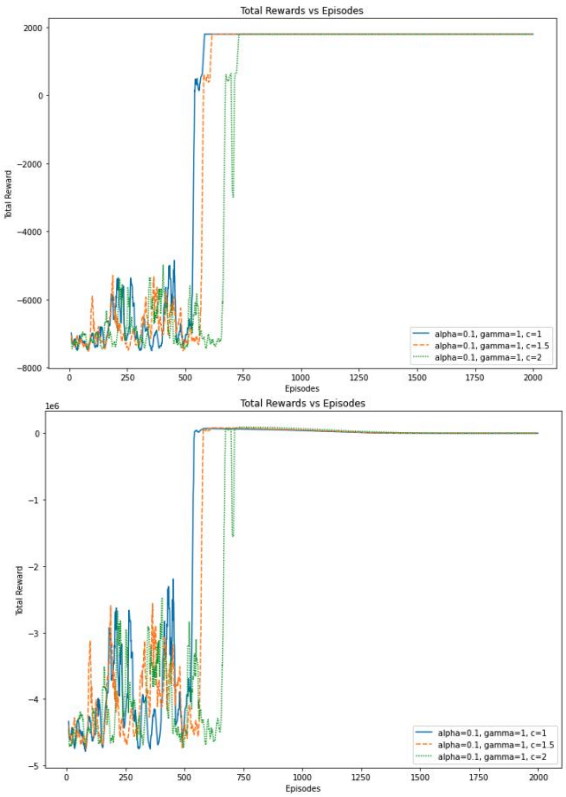

Fig. 7. Hyper-parameter tuning for UCB policy based on the three best performing hyper-parameter combinations acquired through grid search.

Final Model Training Results (UCB)

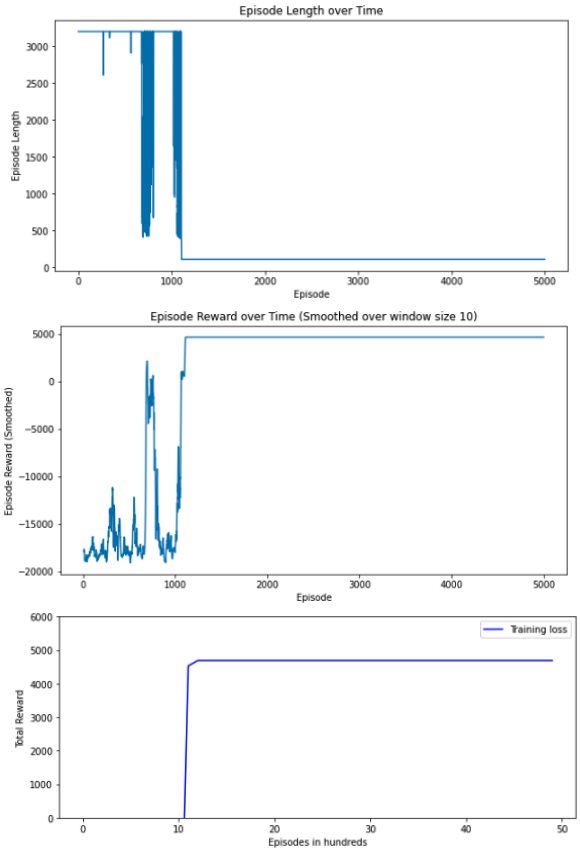

Fig. 8. Final model training results on a 40x40 city-grid environment

In Fig. 6, we can see that our q-learning algorithm run with epsilon greedy policy converges faster (~700 epochs) with epsilon value of 1 and decay of 0.9 in comparison to the other epsilon-decay hyper-parameter combinations. For UCB policy, however, Fig. 7 shows convergence at around ~500 epochs, with confidence bound ("c") parameter value of 1.0 (compared to 0.5, 1.5 and 2.0). When comparing both policies, we can see that the UCB policy outperforms the Epsilon Greedy policy. In the next section we will test our policy on a bigger environment and state-space, testing to see how well our q-learning algorithm will converge given the hyper-parameters acquired from our grid-search.

G. Final model on a bigger environment

Since UCB policy shows to converge the fastest we choose to run our final model using a UCB Policy based Q-Learning algorithm using the best hyper-parameter combinations acquired through our grid-search in the previous section ($\alpha = 0.1, \gamma = 1.0, c = 1.0$), but this time on a bigger environment (40x40 grid). We ran total of 5000 episodes which lasted only 2 minutes and 3 seconds, while our policy converging at around ~1300 episodes (as seen in Fig. 8). Total reward cumulated being 4687.0 and total timestep taken 113. This shows that our UCB based Q-learning algorithm will converge in any state-space and environment size.

III. DEEP Q-NETWORK AND IMPROVEMENTS

A. Motivation

The traditional Q-learning method in a tabular setting breaks down in more complex environments. However, the central ideas behind the algorithm can be generalised and adapted to enable learning in settings approaching real-world complexity. These adaptations are implemented within a neural network using a method introduced by [6], hereby termed as Vanilla DQN. Neural networks are proven to be universal function approximators and lend themselves nicely to the task of approximating Q-values within complex settings.

B. Environment and Problem Definition

The environment of choice is the Cartpole (v-1) task obtained from the Open AI gym [7]. The task involves the agent balancing a pole on a cart moving along a frictionless surface.

The states correspond to the following parameters:

1. Cart Position
2. Cart Velocity
3. Pole Angle
4. Velocity at the tip of the pole

The actions available for the agent include the application of a +1 or -1 force on the cart - to move it in the right and left directions respectively. A reward of +1 is awarded to the agent for every time step that it can keep the pole balanced. An episode terminates when the pole's angle is greater than 15 degrees from the vertical, or if the cart's distance from the centre exceeds 2.4 units. In this case, the agent will receive a reward of 0.

C. Vanilla DQN

1. Introduction

The objective of the Deep Q-Network is to predict the Q-values required to learn an optimal policy. The inputs of the neural network will correspond to an environment's state(s), with the outputs corresponding to the Q-values associated for each action. For the Cartpole problem, the state and action spaces have sizes of 4 and 2, respectively. For the hidden layers, we will use 2, consisting of 8 nodes each.

Traditional neural network infrastructures are built on independent and identically distributed data. Given the sequential nature of an agent's navigation through an environment (and its subsequent update of Q-values), algorithm performance will be hindered and yield unstable results. This is due to the correlations between samples and moving target values. [6] suggests two ways to deal with this:

Experience Replay: The process works by building a store of memory consisting of an agent's transitions through an environment. A transition can be represented by the tuple (state, reward, action, next state). Once a store of sufficient transitions has been collected, the agent can randomly sample batches from memory to use for training. This random sampling reduces the correlation between samples.

Addition of target network: This target network will share the same architecture as the evaluation network described above. It will be used for target generation in the update rule and its parameters will be periodically updated with those from the evaluation network to improve convergence properties.

The target value for the network is given by:

$$y_i^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta^-) \qquad (4)$$

where θ^- corresponds to the parameters for the target network.

2. Problems with DQN

The DQN approach significantly overestimates Q-values [8]. This can be explained by the max operator in equation 4. The target network is used for both action selection and evaluation, which will yield over-approximated Q-values.

3. Base Run for DQN

Fig.9 shows the results of running DQN on the Cart-pole problem with the following hyperparameters: learning rate=0.001, gamma=0.9, decay=0.9999, epsilon=1.0 and minimum epsilon=0.01. We use a memory capacity of 2000 and batch size of 200. The existing literature tends to favour lower learning rates and higher values for the discount factor. The average reward significantly increases, followed by a drop in performance - with significant gyrations following. These can be explained by the problem of overestimation

described above. Rewards decrease when overestimations begin [8].

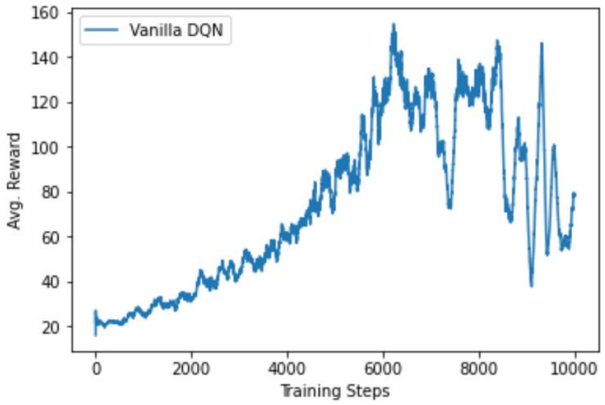

Fig. 9. Vanilla DQN base run results in terms of avg. reward per timestep.

D. Double DQN

1. Introduction

As mentioned in previous section, Q-learning overestimates action values. There is an argument that if this estimation is uniform for all action values, then the algorithm does not suffer. However, there exist certain environments where the over-estimation of Q-values is non-uniform, significantly hindering algorithm performance [8].

Double Q-learning was first implemented in a tabular setting, but its central ideas can be generalised to more advanced environments [9]. The central reason for the overestimation of action values revolves around the argmax operator in equation 4. The target network is used for both action selection and evaluation. The central tenet of Double DQNs is to decouple these processes. In this new setting, the selection of the action is determined using the evaluation (or online) network, whilst action evaluation is performed on the target network. This decomposition allows minimal changes to be made to the existing DQN infrastructure, whilst showing significant improvements in certain environments.

The target value is now given by:

$$y_i^{DDQN} = r + \gamma Q(s', \arg\max_a Q(s', a, \theta); \theta^-) \qquad (5)$$

where θ and θ^- are the parameters of the online and target networks respectively.

2. Motivation and Expectations

For the Cartpole environment, overestimation of Q-values can lead to detrimental effects. The environment terminates if the cart moves beyond a certain boundary or if the pole's angle is greater than 15 degrees. Over-approximations of action values for actions leading to either of these states will result in termination and reduced reward. Implementing a Double DQN should prevent this from occurring. Therefore, it is expected that the average rewards will be higher and also more uniform. That is, the high level of gyration observed in Figure 9 should not be present.

3. Base Run for Double DQN

Figure 10 shows the results of running Double DQN on the Cart-pole problem using identical parameters to the base run implemented for DQN. Greater stability can clearly be observed, implying that decoupling of the max operator for action selection and evaluation produces more stable results.

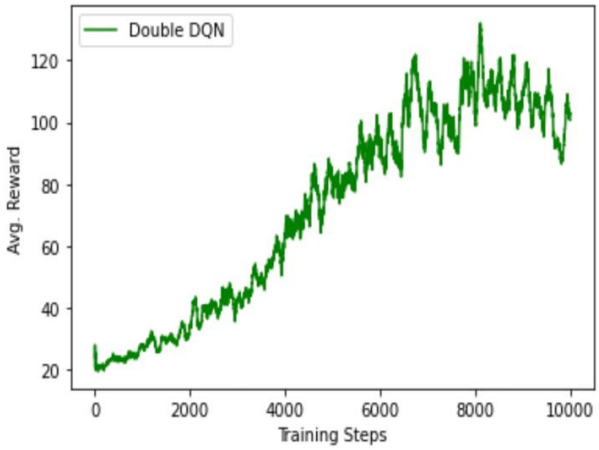

Fig. 10. Double DQN base run results in terms of avg. results per timestep.

4. Hyperparameter analysis

Hyperparameter tuning was performed on the learning rate and gamma. The values tested will correspond to 0.01 and 0.0001 for the learning rate and 0.8 and 1.0 for gamma.

The learning rate controls the extent to which our model adapts to new information. Higher rates correspond to less training time, but the model may converge to a suboptimal solution. For lower rates, training progresses slower as limited updates are being made to the weights of the neural network.

Gamma, or, the discount factor is utilised to discount future rewards. If the agent is only concerned with immediate rewards (lower gamma), it is termed "myopic". If all rewards are given equal value (higher gamma), it is termed "far-sighted".

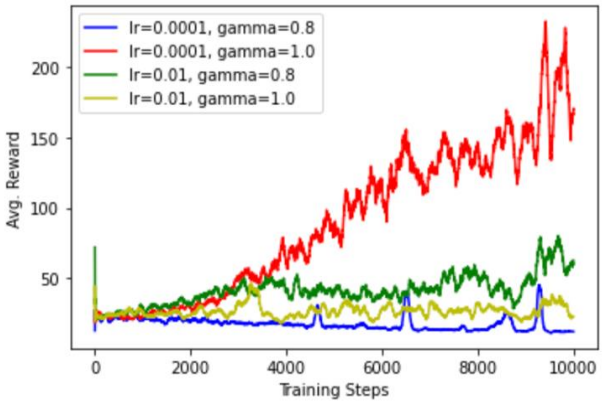

Fig. 11. Double DQN hyper-parameter tuning results

Figure 11 shows the results of hyperparameter tuning. Across all iterations, we observe a greater stability of average rewards in comparison to the implementation of Vanilla DQN. Optimal performance is observed when the learning rate and discount factor are 0.0001 and 1.0 respectively. The higher value for gamma implies that the agent should look at all future rewards equally. In the Cartpole environment, the agent

receives a reward of +1 per timestep it can manage to balance the pole, and a reward of 0 if the pole falls. Given the sequence of 1s and 0s the agent will receive, it is imperative that it considers all rewards equally in order to ascertain when it receives a reward of 0 (so it can factor this into the Q-value update). For learning rates corresponding to 0.01, we notice sub-par performance. The lower level of gyration observed implies that over-estimation does not occur. However, the low values imply that the algorithm converges to suboptimal values – as can be expected when using higher learning rates. These results are in line with the existing literature – where lower learning rates are preferred. For example, [8] uses a learning rate of 0.00025.

E. Dueling DQN

1. Introduction

Dueling Deep Q-Networks were introduced as a novel approach compared to traditional methods [10]. Their utility lies primarily in the realm of model free RL. It separates two estimators: state value functions (action independent estimates of the value of a given state) and action advantage functions (which are state-dependent). These two streams are then combined in an aggregate layer to produce an estimation of the Q-values. Please note here that these values cannot simply be added together to produce Q. This will lead to the issue of identifiability - where there exist no unique solutions to the equation. Therefore, the combination of the state and advantage values will need to include an average over the advantage values of the next state in order to ensure a closed solution. Once completed, the traditional methods of action selection and evaluation can proceed.

This approach learns valuable states without examining the effect of the actions available in each state. This method is especially useful in states where actions do not affect the environment in any significant way. It also has the added benefit of being compatible with existing and future reinforcement learning algorithms.

2. Motivation and Expectations:

In the Cartpole environment, there exist certain states whose inherent value is independent of any action taken. For example, if the cart is situated in the centre along with a pole angle of 0 and minimal cart and pole velocity, exerting a force (action) on the left or the right will not significantly alter the environment. The Dueling DQN will be able to recognise this. Additionally, any states where the car position is significantly off-centre with high velocity and pole angle will be recognised as important by the agent. Therefore, we expect to see improved average rewards. Given that the traditional methods of action selection and evaluation will be implemented as in DQN, we can expect greater instability in these rewards – when compared to the Double DQN approach.

3. Base Run for Dueling DQN

Figure 12 shows the results of running Dueling DQN on the Cart-pole problem using identical parameters to the base run implemented for vanilla DQN. A higher average reward can be noticed, in addition to the significant gyrations as expected.

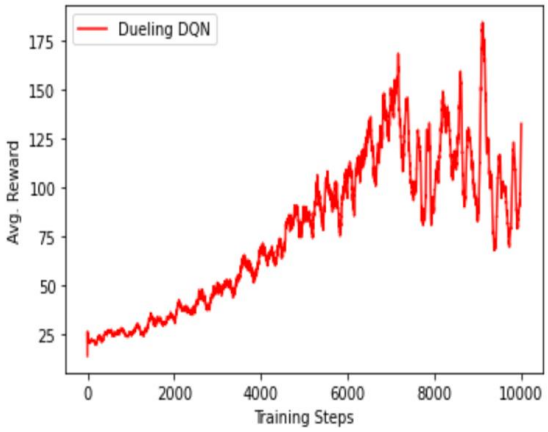

Fig. 12. Dueling DQN base run results in terms of avg. reward per time step.

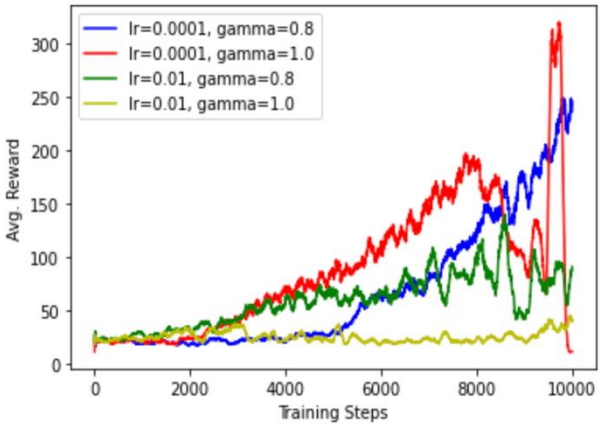

Fig. 13. Dueling DQN hyper-parameter tuning results

4. *Hyperparameter analysis*

Hyperparameter tuning was performed on the learning rate and gamma. The values tested will correspond to 0.01 and 0.0001 for the learning rate and 0.8 and 1.0 for gamma. Further explanation on these parameters can be found in section D, part d.

Figure 13 shows the results of hyperparameter tuning. Optimal performance is observed when the learning rate and discount factor are 0.001 and 1.0 respectively. A higher average reward can be observed for two of the iterations. For the evaluation of higher discount factors and lower learning rates, please refer to section D, part d. Those concepts translate to the Dueling DQN as well. The primary takeaways from the figure are the significant gyrations in average rewards, especially towards the end of training. While this can be attributed to over-approximation, there are certain improvements that can be made otherwise. For example, a higher batch and memory size could be used. Due to computational restraints however, we were unable to implement this. To deal with over-approximation, the Dueling network architecture can utilise the action selection/evaluation decoupling implemented in Double DQNs.

IV. Atari Learning Environment

A. Introduction

This section examines the use of advanced RL algorithms on the suite of Atari games available on the Open AI gym environment – using the RL-Lib library.

1. *Environment/Game Selection*

The game of choice is Breakout - it includes a wall of bricks, a bat and a ball. Upon hitting a block, a reward is received, and the block is removed. The bat has to be moved along the bottom of the screen to prevent the ball from going out of play – which would cause the agent to lose a life. The agent's actions correspond to movement along a horizontal axis at the bottom of the screen.

In the Atari learning environment, two versions of this game are provided:

- Observations provided as pixels
- Observations provided as positions in RAM - version of choice, chosen as the literature experiments on the former version. It will be seen if similar results hold.

2. *Algorithm Choice*

Breakout was chosen as experiments in [6] and [10] showed marginal improvement in performance for double deep Q-networks and *decreased* performance for dueling deep Q-networks. Research will be conducted to see if these results can be improved. An ablation study will be implemented, examining performance of a Vanilla DQN on the game, adding the improvements and analysing performance as a result of their addition. Performance Experience Replay will also be considered as in improvement.

3. *Algorithm Presentation and Justification*

a) Deep Q-network (DQN): Given the experimental results presented by [6] and [7], this will be included in analysis to see if the results hold. While DQNs suffer from the problem of overestimation, [11] argues that optimism in uncertain environments can be a useful explorational technique. For a more detailed presentation on DQNs, please refer to Section III, part C.

b) Double Deep Q-networks: This network will be examined to see if the experimental results illustrated in the previous section can be improved. Improvements include using a reduced learning rate and higher gamma value. The learning rate used by [6] is 0.00025. Lower values will be implemented and results observed. Higher values for the learning rate will not be examined – see Analysis for explanation. For a more detailed presentation on Double DQNs, please refer to Section III, part D.

c) Dueling Deep Q-networks: The poor performance of Dueling DQNs on Breakout evidenced in the literature is surprising. The game contains multiple states where any action executed by the agent will not significantly alter the environment. One such state includes the moment right after the bat hits the ball, sending it on an upward trajectory towards the wall of bricks. Any action the agent executes will not create significant alterations to the environment. The architecture provided by Dueling DQNs should allow the agent to recognise such states and improve learning. Reasons for underperformance may be due to the fact that model-free models generally require involved hyperparameter tuning, otherwise results may falter [12]. We will experiment with this using the lower learning rates mentioned in the previous paragraph. For a more detailed presentation on Dueling Deep Q-networks, please refer to Section III part E.

d) Prioritised Experience Replay (PER)

This improvement is related to Experience Replay (described in section III, part c). It aims to prioritise certain experiences over others – as they may contain more valuable information which the agent can prioritise learning over [13]. Higher priority experiences will have a greater probability of being sampled, in comparison to the uniform sampling performed in the standard experience replay paradigm. This priority can be calculated by examining the magnitude of the TD error. Importance sampling weights are added to counterbalance the bias introduced by this method. In the Breakout environment, prioritisation of certain experiences can enhance learning. For example, if this environment can prioritise transitions where "tunnel" are dug into the wall (which will increase the overall score), more learning can be allocated to these memories. PER will be used across all iterations in the ablation study due to the explanations above.

B. Results

The ablation study was conducted in tandem with hyperparameter tuning using Ray Tune. A summary of results is presented in the table and figure below.

TABLE V. OPTIMAL HYPERPARAMETER COMBINATIONS FOR ABLATION STUDY

Double DQN	*Dueling DQN*	γ	α	Mean Reward
True	False	1.0	0.00001	10.9
True	False	0.9	0.00001	10.86
False	True	1.0	0.00001	10.17
False	True	0.9	0.00001	10.1
True	True	1.0	0.00001	9.76
False	False	1.0	0.00001	9.07

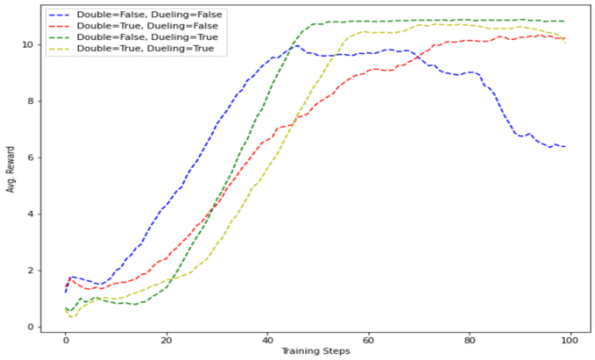

Fig 14. Dueling DQN hyper-parameter tuning results

C. Analysis

Hyperparameter tuning was run for 100 iterations on the Breakout environment. Whilst this is not ideal for thorough analysis, computational restrains limit a broader study.

The neural network consists of two hidden layers with 64 units each. [14] implemented training on Breakout with two hidden layers consisting of 128 nodes each, and noticed high variance in results, due to overfitting. This motivates the choice for a lower amount of nodes in the hidden layers (64). Values for the learning rate corresponded to 0.00001 and 0.0001, with values for the discount factor being 0.9 and 1.0. Optimal results were obtained for the lower learning rate, with results varying for gamma. Lower learning rates are preferred as higher values result in larger steps in the parameter space of the neural networks, which can lead to significant changes in the policy space and hinder results.

It can also be noticed that the dueling implementation actually performs *better* than the Vanilla DQN, in contrast to results observed by [10]. This improvement can be explained by the lower learning rate explained in the previous paragraph. Figure 14 also shows that this algorithm converges the fastest (the green line). We also notice greater stability for the dueling implementation. This is due to the initiation of Prioritised Experience Replay [13].

Observing table 8, optimal results were shown for the Double DQN. This falls in line with the experiments conducted in the existing literature [8].

REFERENCES

[1] Watkins, C. J. C. H. (1989) *Learning from delayed rewards*. University of Cambridge.

[2] Watkins, C.J. and Dayan, P., 1992. Q-learning. *Machine learning*, *8*(3), pp.279-292.

[3] Wikipedia contributors (2022) *Prim's algorithm*, *Wikipedia, The Free Encyclopedia*. Available at: https://en.wikipedia.org/w/index.php?title=Prim%27s_algorithm&oldid=1083605811

[4] Munro, P. *et al.* (2011) "Bellman Equation," in *Encyclopedia of Machine Learning*. Boston, MA: Springer US, pp. 97–97.

[5] Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction*. MIT press.

[6] Mnih, V. *et al.* (2015) "Human-level control through deep reinforcement learning," *Nature*, 518(7540), pp. 529–533.

[7] OpenAI (no date) *Gym: A toolkit for developing and comparing reinforcement learning algorithms*, *Openai.com*. Available at: https://gym.openai.com/envs/CartPole-v1/ (Accessed: April 23, 2022).

[8] van Hasselt, H., Guez, A. and Silver, D. (2015) "Deep reinforcement learning with Double Q-learning," *arXiv [cs.LG]*.

[9] H. van Hasselt. 2010. Double Q-learning. *Advances in Neural Informa- tion Processing Systems*, 23:2613–2621.

[10] Wang, Z. *et al.* (2015) "Dueling network architectures for deep reinforcement learning," *arXiv [cs.LG]*.

[11] L.P. Kaelbling, M.L. Littman and A.W. Moore. Reinforcement Learning: A survey. *Journal of Artificial Intelligence Research*, 4:237-285, 1996

[12] Haarnoja, Tuomas, et al. "Soft actor-critic algorithms and applications." *arXiv preprint arXiv:1812.05905* (2018).

[13] Schaul, Tom, et al. "Prioritized experience replay." *arXiv preprint arXiv:1511.05952* (2015).

[14] Sygnowski, Jakub, and Henryk Michalewski. "Learning from the memory of Atari 2600." *Computer Games*. Springer, Cham, 2016. 71-85

Summary of Individual Contributions to Team Tasks

1. **Define an environment and a problem to be solved**
   Given that a custom environment was created, both individuals worked in tandem to create its underlying infrastructure.
2. **Define a state transition function and reward function**
   Jerome – Definition of state transition function
   Elnara – Definition of reward function
3. **Set up Q-Learning parameters (gamma, alpha) and policy**
   Jerome – Defined Q-Learning with E-greedy policy and relevant hyperparameters
   Elnara – Defined Q-Learning with UCB policy and relevant hyperparameters
4. **Run the Q-Learning algorithm and represent its performance**
   Jerome – Implementation of Q-learning with UCB policy
   Elnara – Implementation of Q-learning with E-greedy policy
5. **Repeat the experiment with different parameter values and policies**
   Jerome – Ran hyperparameter tuning for Q-learning with E-greedy policy
   Elnara – Ran hyperparameter tuning for Q-learning with UCB policy
6. **Analyse the results quantitatively and qualitatively**
   Jerome – Performed analysis for Q-learning with E-greedy policy
   Elnara – Performed analysis for Q-learning with UCB policy
7. **Implement DQN with two improvements, presented during the Lectures/Labs or from the literature (if from the literature, please consult first with Michael Garcia-Ortiz). Motivate your choice and your motivations for choosing a particular improvement. Apply it in an environment that justifies the use of Deep Reinforcement Learning**
   Jerome – Implementation of Dueling Deep Q-network
   Elnara – Implementation of Double Deep Q-network
   Both – Implementation of Vanilla Deep Q-network
8. **Analyse the results quantitatively and qualitatively**
   Jerome – Analysis for Double Deep Q-network
   Elnara – Analysis for Dueling Deep Q-network

Link to Github Repo

All code implemented for this project can be found at the following Github repo: https://github.com/JeromeTitus/INM707-Coursework.git