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")

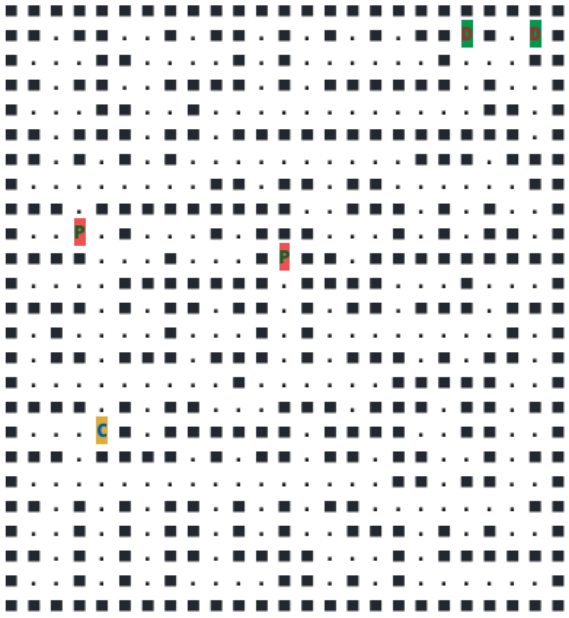


Fig. 1. Custom built Uber Pool environment (*P – Passengers, D – Destinations, C – Car location*).

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 - width, y - height) 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 below is the graphical representation of our states, with the possible transitions between sub-spaces.

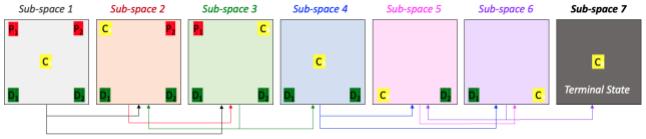


Fig. 2. State space with its sub-space components and the possible transitions between them represented by arrows.

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 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).

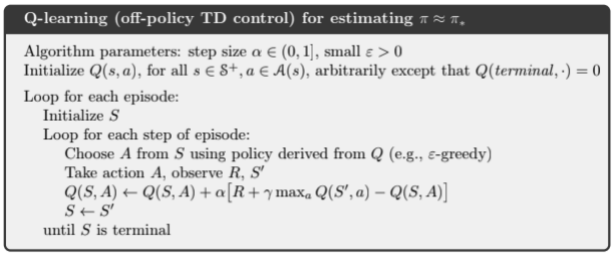


Fig. 3. Procedural form of Q-learning algorithm [5].

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 in the shortest timestep possible. 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 and 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.

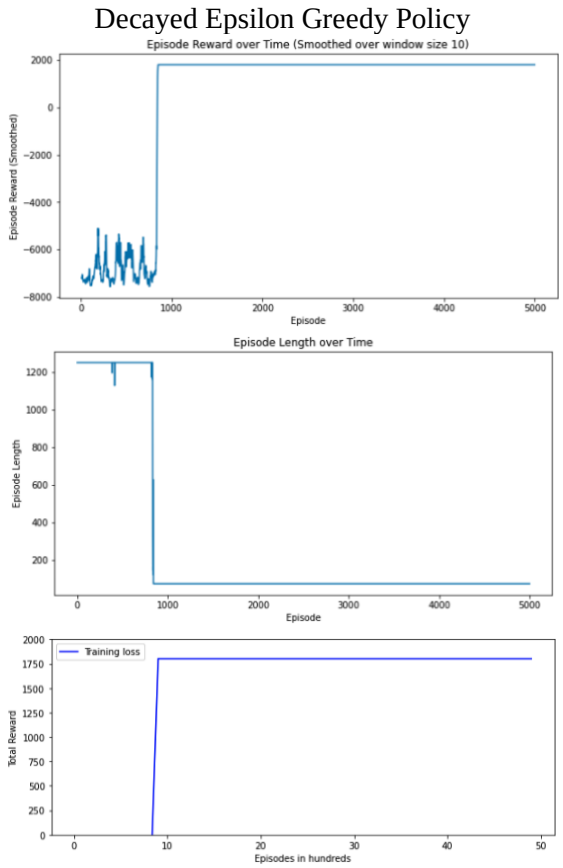


Fig. 4. Training Results for Decayed Epsilon Greedy Policy.

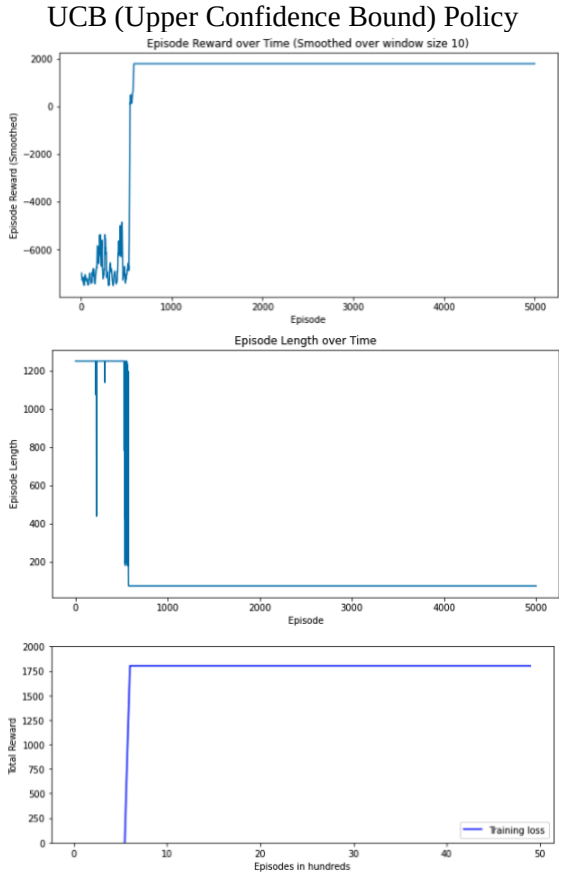


Fig. 5. Training Results for UCB Policy.

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

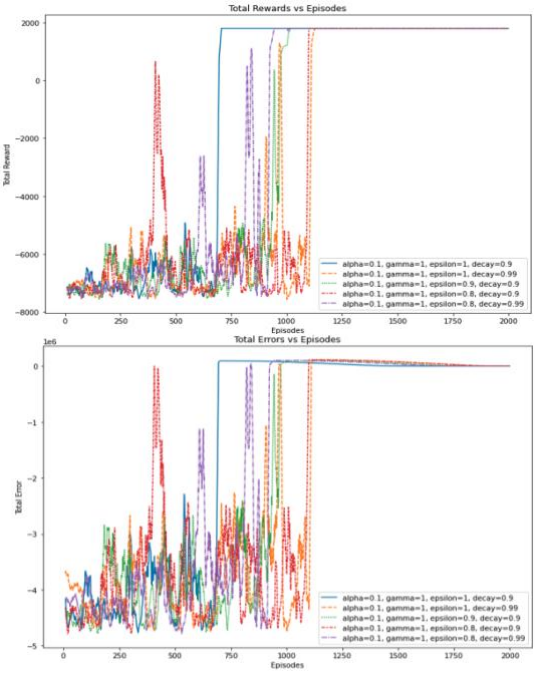


Fig. 6. Hyper-parameter tuning for Decayed Epsilon Greedy policy based on the five best performing hyper-parameter combinations acquired through grid search.

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

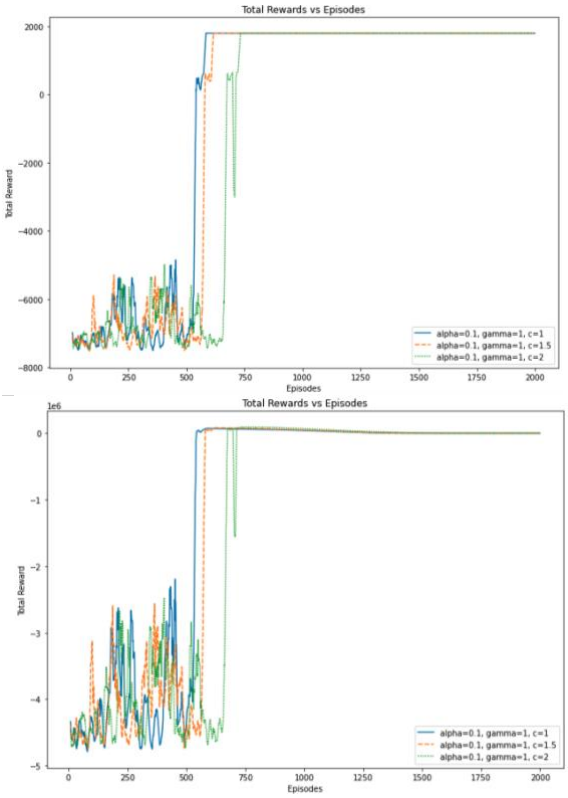


Fig. 7. Hyper-parameter tuning for UCB policy based on the three best performing hyper-parameter combinations acquired through grid search.

Final Model Training Results (UCB)

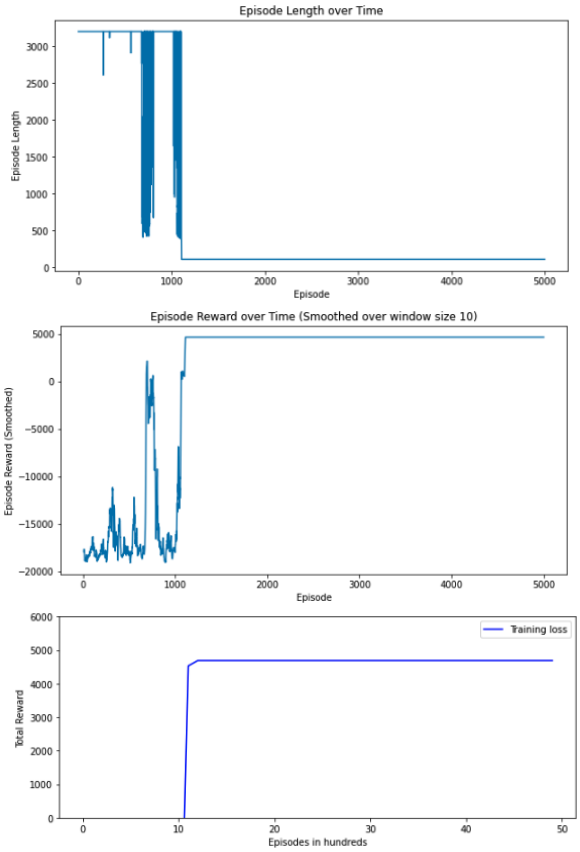


Fig. 8. Final model training results on a 40x40 city-grid environment

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:

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

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].

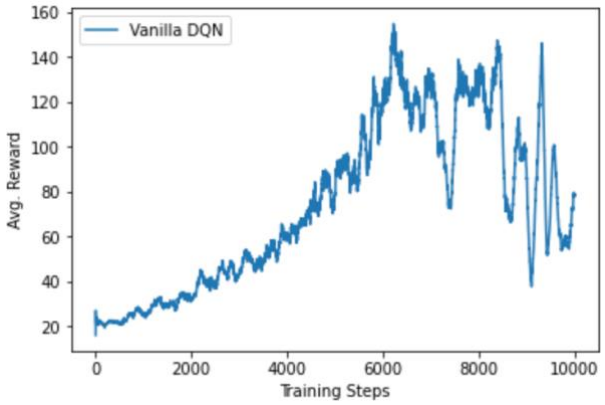


Fig. 9. Vanilla DQN base run results in terms of avg. reward per timestep.

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.

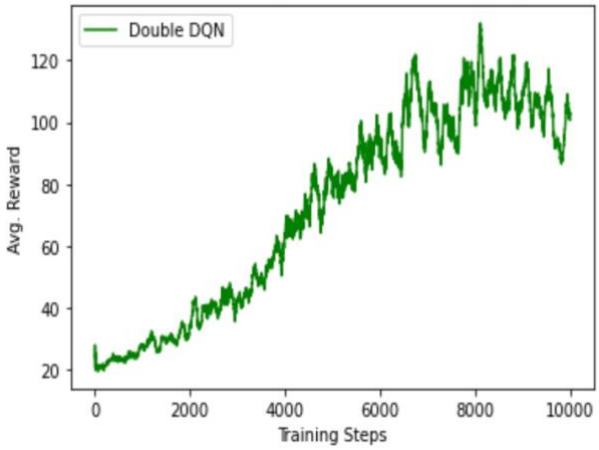


Fig. 10. Double DQN base run results in terms of avg. results per timestep.

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".

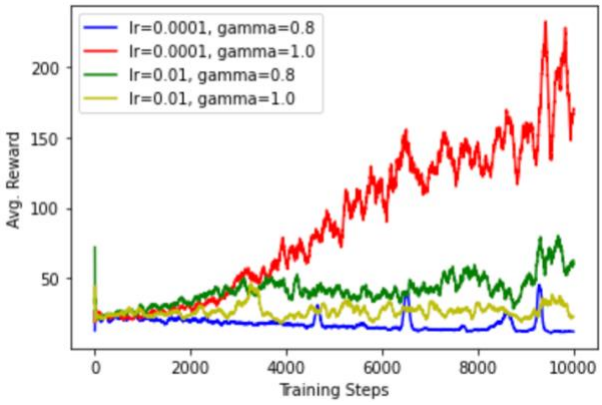


Fig. 11. Double DQN hyper-parameter tuning results

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.

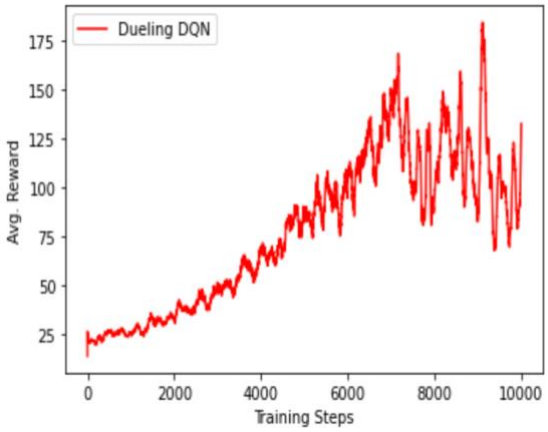


Fig. 12. Dueling DQN base run results in terms of avg. reward per time step.

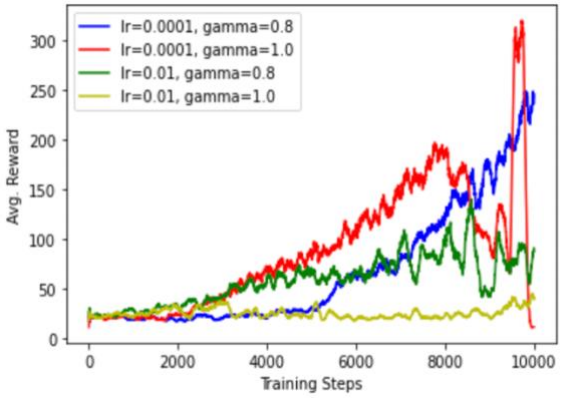


Fig. 13. Dueling DQN hyper-parameter tuning results.

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. Q-Learning Implementation in Atari Environment

A. Introduction

In this section, we will apply DQN and some variants on Atari 2600 Pong game using OpenAI's gym library [7] to simulate the environment. Additionally, we will also be using Ray's RLlib library (an open-source library for reinforcement learning) where we will create custom neural network policy using PyTorch [11].

B. Environment and Problem Statement

Pong a table tennis–themed twitch arcade sports video game, featuring simple two-dimensional graphics, manufactured by Atari, and originally released in 1972 [12]. It is a simple tennis-like game featuring two paddles and a ball, the goal being to gain 20 points (a player gets a point every time the opponent misses a ball). This game presents a classic reinforcement learning problem, where we have an agent within fully observable environment, executing actions that yield different rewards. The state here is defined as the position of the agents and the ball within the game space (Fig. 14).

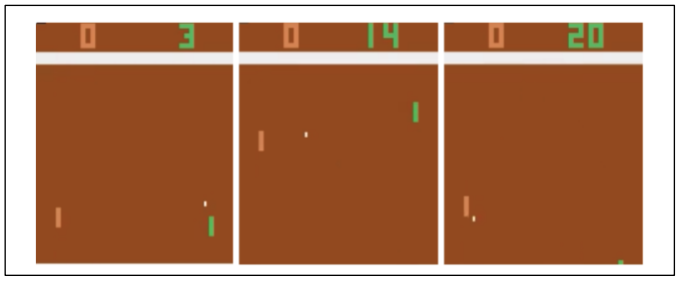


Fig. 14. Snapshot of Atari Pong game.

The Pong environment we chose for our task is a graphics version which yields the observation in terms of pixels on the screen, instead of ram (yields observations in terms of what is going on in the computer memory). The environment names for OpenAI's gym environments include a version and suffix. We will be working with v4 where agent performs the chosen actions. v0 - has a repeat action probability of 0.25 meaning the quarter of the time the previous action will be used instead of the chosen action.

For suffix we will be working with Deterministic where the frames skip 4 frames at each step. In other words, the agent sees and selects actions on every 4th frame instead of every frame, and its last action is repeated on skipped frames. This allows the agent to play roughly 4 times more games without significantly increasing the runtime. Another available option is NoFrameskip, where the frame skip is equal to zero at each step. In essence, the Deterministic environment would achieve same results as NoFrameskip but with smaller buffer.

Our Pong environment has 6 discrete action space, corresponding to ['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']. Even though Open AI's Pong Environment has six actions, three of the six are redundant (FIRE is equal to NOOP, LEFT is equal to LEFTFIRE and RIGHT is equal to RIGHTFIRE). So, in essence there are only three actions an agent (player) can take within the Pong Environment: remaining stationary, vertical translation up, and vertical translation down. Generally, each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from {2,3,4}. However, since we are working with a deterministic environment, k is sampled every 4 frames instead.

The observation space is an RGB image (red, green, and blue values for each pixel) of the screen, displayed at a resolution of 210 by 160 pixels.

We will build a deep network by feeding pixel data as input and predict actions in order to learn to win the game for the network. The deep reinforcement learning netweok used in this project is Deep Q-Network (DQN).

C. Policy

The policy is the basic DQN network in a torch framework, with implementations of Double DQN, Dueling DQN and Prioritized Replay (default configurations for DQNTrainer in RLlib library). As previously discussed in the third section, Double Q-learning helps minimize the overestimation bias introduced by the conventional Q-learning, while Dueling networks uses two different neural networks (one outputting value of the state and the other the advantage of each action) to mitigate the overoptimistic value estimates created by Double DQN. The prioritized experience replay built on top of DDQN leads to both faster learning and to better final policy quality.

The values of all the hyperparameters and optimization parameters were adapted from (Mnih et al., 2015) [6]. We did not perform a systematic grid search owing to the high computational cost. The values and descriptions of all hyperparameters are provided below:

- `"lr"=1e-4`
- `"learning_starts"=10000` – which controls when the first optimization step happens
- `"explore" = True` – we will keep the default value for explore to enable exploration schedule; this way we make sure the agent can take an action chosen at random with a probability that decays over time, instead of just taking the action that the agent might deem best. This can help avoid over-fitting.
- `"discount_factor"=0.99` – discount factor gamma used in the Q-learning update
- `"hiddens"=[512]` – rectifier units for final fully connected layer. *since we enabled dueling architecture, we modify this parameter for dense-layer setup for each of the advantage branch and value branch.*
- `"exploration_config"` - parameters for the exploration class constructor:
  - `"type"="EpsilonGreedy"` – linear annealed epsilon-greedy policy which updates epsilon based on steps.

- "epsilon_timesteps" = 200000 – timesteps over which to anneal epsilon, i.e., how many steps to play
- "final_epsilon"=0.01
- "initial_epsilon"=1.0

```
if epsilon > final_epsilon:
      epsilon -= (initial epsilon-
                  final epsilon)/
                  epsilon timesteps
```

- "buffer_size"=50000 – size of the replay buffer. The larger the experience replay, the more stable the training of the network is. However, a large experience replay also requires a lot of memory and it might slow training. Since we will be using PongDeterministic-v4 environment we can keep the buffer small.
- "model" – "dims" = 84 – final resize frame dimensions
- "training_batch_size" = 32 – how many transitions to sample each time experience is replayed.

D. Network Architecture

The architecture of our Q-network is as follows. The input to the neural network consists of a 84 x 84 x 4 image. The first hidden layer convolves 32 filters ("train_batch_size") of 4 x 4 with stride 4 input image ("rollout_fragment_length"). Ray divides episodes into fragments of this many steps during rollouts. Sample batches of this size are collected from rollout workers and combined into a larger batch of "train_batch_size" for learning. These fragments are concatenated and we perform an epoch of SGD (refer to ray documentation for ray.rllib.agents.trainer source code [13]).

The final hidden layer is fully connected and consists of 512 rectifier units. The output layer is a fully connected linear layer with a single output for each valid action. The number of valid actions varying between 1 and 6. The behaviour policy during training is ε-greedy with ε annealed linearly from 1.0 to 0.01 over the first 200000 frames and fixed at 0.01 thereafter.

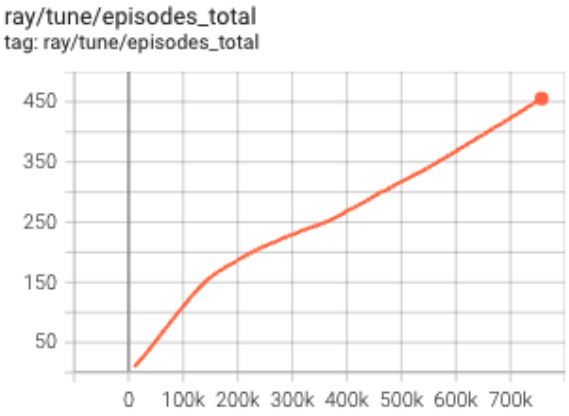


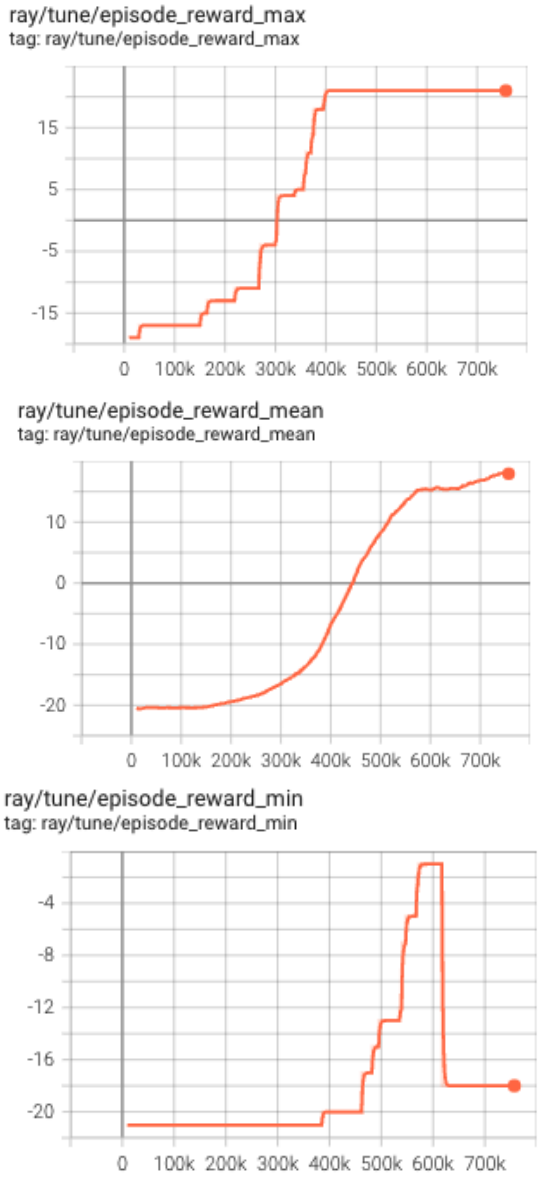
Fig. 15. Trained DQN model episode_total vs. timesteps.

E. Training Results

To evaluate our results within the confinement of the Colaboratory environment, we record an entire episode and display its results within a tensorboard.

We trained for a total of 757K time steps (that was equivalent to 456 episodes (Fig. 15) with around 2hrs of game experience in total) and used a replay memory of 50000 most recent frames (DQNTrainer default config).

As seen in Fig 16. our episode_reward_mean has improved over the training period although it's still a long way off from being capable of defeating the benchmark system (max reward = 21.0). All in all, our results are acceptable given the small training times (due to computational cost), and we can safely assume that our q-learning model would converge given enough training time. A study on the Pong environment by Phon-Amnuaisuk suggested that roughly 10000 episodes of run time is needed to generate an agent capable of achieving a win [14]. The final episode_mean_reward in our training is 18. However, at around 600K timestep we can see a plateau followed by a dip in episode_reward_min which may be caused by



overestimation. Same plateau was also observed in episode_mean_reward. As a future work, running a PPO algorithm and comparing it with our DQN policy might yield interesting results for further discussion.

Fig. 16. Rewards per training timestep acquired for PongDeterministic-v4 Atari Environment via RLlib's DQNTrainer.

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] *RLlib: Industry-grade reinforcement learning — ray 2.0.0.Dev0 Ray.io*. Available at: https://docs.ray.io/en/master/rllib/index.html

[12] Wikipedia contributors (2022) *Pong*, *Wikipedia, The Free Encyclopedia*. Available at: https://en.wikipedia.org/w/index.php?title=Pong&oldid=1083384511

[13] *Ray.RLlib.Agents.Trainer — ray 1.12.0* (no date) *Ray.io*. Available at: https://docs.ray.io/en/latest/_modules/ray/rllib/agents/trainer.html

[14] Phon-Amnuaisuk, S. (2018) "Learning to play Pong using policy gradient learning," *arXiv [cs.LG]*. Available at: http://arxiv.org/abs/1807.08452 .