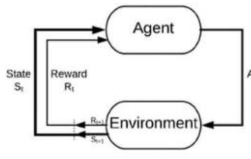
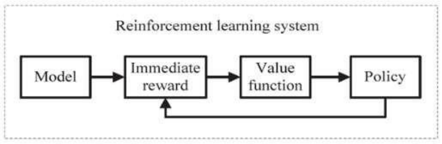
Basic Task – Q-Learning

Reinforcement learning is a machine learning technology that seeks to find the best policy by rewarding favourable behaviours and penalising detrimental ones. There are two categories of reinforcement learning methods: value-based and policy-based. The policy-based approach attempts to learn the stochastic policy directly, whereas the value-based approach indirectly learns the policy by selecting the most advantageous action in each state that maximises the cumulative reward.





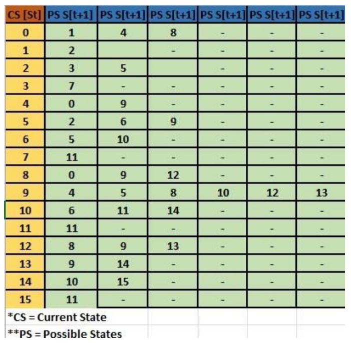
i1

Q-learning is a reinforcement learning value-based approach that entails learning an action-value function that determines the expected utility of a certain action. The value-based set of rules update the function using mainly the Bellman equation. In comparison, the policy-based approach uses a greedy policy carried from the previous policy improvement to estimate the value function.

For the first (Basic) task of our assignment, our environment will be focussed on the London Underground Tube Map. Q-Learning methodologies will be applied in order to understand the best possible path in our environment from any given station to our target station. We will concentrate on a portion of the tube map around the Central London Zone 1 area. As most will be aware certain stations can be reached via only limited or singular tube lines whereas certain stations are accessible via multiple lines. Our environment is deterministic and static, and the next state of the environment can always be predicted based on the current state and the agent's activities and the distinct states will be represented by the stations. Our goal is to use the techniques available to find the shortest most efficient route to Holborn station.

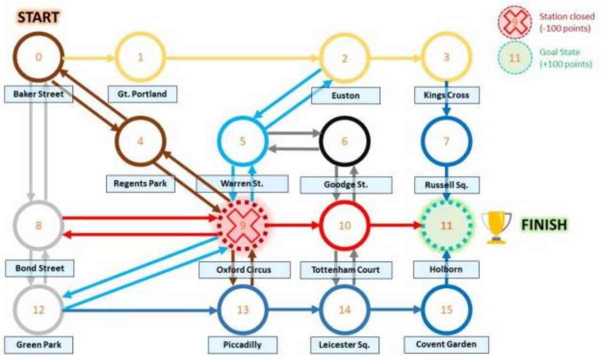
State Transition Function & Reward Function

The state transition function defines moving from one state to another. The agent can move from its current state S[t] to its potential future states S[t+1]. The table below lists the current states on the LHS and the possible movements across 16 states from 0 to 16. Agents remain static in their current state if in their goal state.



CS [St]	PS S[t+1]	PS S[t+1]	PS S[t+1]	PS S[t+1]	PS S[t+1]	PS S[t+1]
0	1	4	8	-	-	-
1	2		-	-	-	-
2	3	5		-	-	-
3	7	-	-	-	-	-
4	0	9	-	-	-	-
5	2	6	9	-	-	-
6	5	10	-	-	-	-
7	11	-	-	-	-	-
8	0	9	12	-	-	-
9	4	5	8	10	12	13
10	6	11	14	-	-	-
11	11	-	-	-	-	-
12	8	9	13	-	-	-
13	9	14	-	-	-	-
14	10	15	-	-	-	-
15	11	-	-	-	-	-

*CS = Current State
**PS = Possible States



The network map is made up of a series of connections and intersections. Each connection denotes a station, and each intersection denotes a station with multiple connections (i.e. oxford circus, Tottenham court road etc). There are a total of 36 possible actions, which was sufficient for training our model.

Our aim is for the agent to arrive at its destination station, Holborn, or state 11. A reward function has been used to enable our agent to collect the highest possible reward by reaching its goal state and avoiding states that provide 'NaN' or negative rewards. The R Matrix below depicts the immediate reward that can be obtained for performing an action in a state. We assumed that 'Oxford Circus' station was closed and so an alternative path must be learned by the agent. We chose this station specifically because it has numerous connections and intersections. As a result, state 9 has negative rewards of -100, as shown by our R-Matrix, and the agent should avoid going to this state. The agent will be rewarded for reaching state 11, as we can see that the rewards are +100 here.

There are no other incentives for the agent to relocate to any other states. NaNs should be avoided because they provide no reward. The reward paths with a '0' indicate that there are possible paths to the goal state, but the reward collected is '0'.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	nan	0	nan	nan	0	nan	nan	nan	0	nan	nan	nan	nan	nan	nan	nan
1	0	nan	0	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
2	nan	nan	nan	0	nan	0	0	nan	nan	nan	nan	nan	nan	nan	nan	nan
3	nan	nan	nan	nan	nan	nan	nan	0	nan	nan	nan	nan	nan	nan	nan	nan
4	0	nan	nan	nan	nan	nan	nan	nan	nan	-100	nan	nan	nan	nan	nan	nan
5	nan	nan	0	nan	nan	nan	nan	nan	nan	-100	nan	nan	nan	nan	nan	nan
6	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	0	nan	nan	nan	nan	nan
7	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	100	nan	nan	nan	nan
8	0	nan	nan	nan	nan	nan	nan	nan	nan	-100	nan	nan	0	nan	nan	nan
9	nan	nan	nan	nan	nan	0	nan	nan	nan	nan	0	nan	0	0	nan	nan
10	nan	nan	nan	nan	nan	nan	0	nan	nan	nan	nan	100	nan	nan	0	nan
11	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	100	nan	nan	nan	nan
12	nan	nan	nan	nan	nan	nan	nan	nan	nan	-100	nan	nan	nan	0	nan	nan
13	nan	nan	nan	nan	nan	nan	nan	nan	nan	-100	nan	nan	nan	nan	0	nan
14	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	0	nan	nan	nan	nan	0
15	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	100	nan	nan	nan	nan

Setting up the Q-learning parameters and policy

Epsilon Greedy Policy is used to determine a good balance between exploration with probability Ɛ and exploitation with probability 1- Ɛ. Exploration allows the agent to enhance its current knowledge about each action that it can take with the aim to improve its accuracy on estimated action-values. Exploitation chooses the greedy action to get the most reward by exploiting the agent's current action-value estimates. We initially set the exploration rate Ɛ=1 as is with the random policy. This means that the agent will start by exploring the environment randomly (with 100% likelihood) rather than exploiting it. As time goes on and the agent's knowledge of the environment grows, Ɛ begins to decay λ at a rate that we specify, reducing the need to explore. The agent therefore starts to exploit more. The trade-off between exploration and exploitation has a value between 0 and 1 with a value closer to 1 meaning leaning more towards exploration and vice versa. The decay factor is set to 0.9999 when epsilon>=0.5 and 0.99995 otherwise. This makes sure that the agent explores more at first, and then begins to exploit as it learns. We will experiment with a range of epsilon values of [0.1, 0.5, 0.75 and 0.9] with the initial value set to 0.9.

The learning rate, or alpha, α defines the magnitude the agent would use new information rather than old information to update the Q-Matrix (i.e how much Q-Values are updated with each episode). This can take values ranging from 0 to 1, with 0 indicating that the Q-Values are never updated, implying that no learning occurs and that the model relies solely on past knowledge. Setting the alpha closer to 1 means that learning occurs faster, using the most recent information, but relies less on prior knowledge. We must strike a balance so that our agent learns from previous Q-Values, and we can decide how much past knowledge to keep for our state-action pairs. We'll be experimenting with three different values [0.01, 0.1, 0.5, and 0.9]. The initial value is set to 0.9.

The discount factor, also known as gamma, γ, determines whether future rewards are more valuable than immediate rewards. This can have a value ranging from 0 to 1. Setting the gamma to 1 indicates that the agent seeks to maximise rewards over the long term, whereas a value of 0 indicates that the rewards will be maximised immediately. As we become near to the deadline, immediate rewards will be more appealing because we are already nearing maturity and thus longer-term rewards will be less relevant, causing our gamma to decrease. We'll test with gamma values of [0.01, 0.1, 0.5,0.8 and 0.9]. Our initial gamma value of 0.8 indicates that our agent will seek to maximise the long-term reward structure.

Running the Q-learning algorithm and performance evaluation

The below steps show how the Q-Learning Algorithm is run and subsequently how the Q-Matrix is updated (using the formula below), from the beginning using some initial parameters.

We say that the episode has finished once the agent reaches its goal state which for us is state 11.

$$Q_{new}(s,a) = Q_{old}(s,a) + \alpha[(r(s,a) + \gamma \max Q(next.s, all.a)) - Q_{old}(s,a)]$$

ii

We chose the parameters of Alpha = 0.9, gamma = 0.8, epsilon = 0.9 and greedy decay of 0.99999.

Starting state is 12 and the available actions in this state are [8, 9,13]. Q values for this current state are 0 and the since the policy is random, and there are no rewards the agent picks the new state randomly. Our agent selects 13 and therefore 13 is our new state. The current q value and reward for action 13 is 0.

Q value is updated below with the following formula

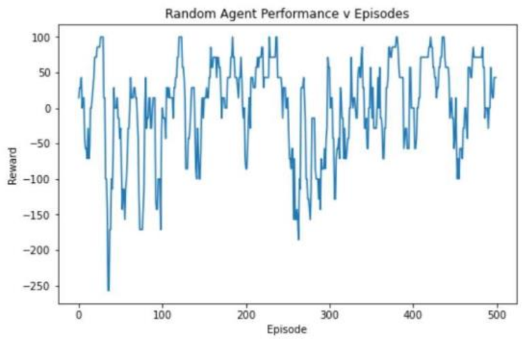
Q value update: Q(12,13) = Q(12,13) + alpha*(r(12,13) + gamma*max(Q[13,:]) - Q[12,13])
Q matrix update: Q(12,13) = 0.0 + 0.9*(0.0 + 0.8*0.0 - 0.0) = 0.0

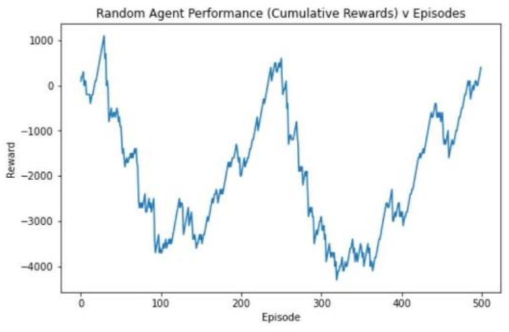
The above is repeated until the goal state is reached. The final state in our code shows that when we reach sta te 15, the possible next states are 11. Selecting the greedy action 11 with current q value 0 and reward of 100. The new state is 11 (Our goal state) and the Q matrix is updated as follows.

Q value update: Q(15,11) = Q(15,11) + alpha*(r(15,11) + gamma*max(Q[11,:]) - Q[15,11])
Q matrix update: Q(15,11) = 90.0 + 0.9*(100.0 + 0.8*0.0 - 90.0) = 90.0

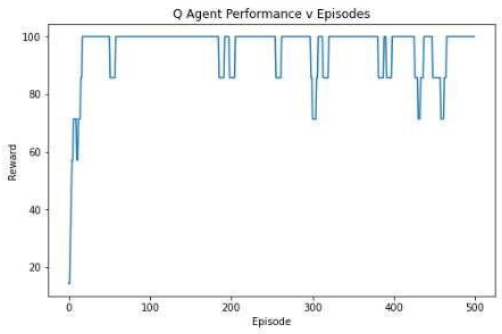
We ran some preliminary performance tests to compare the random and greedy policies.

We began our test with a random agent in our environment. The graphs below show the rewards in relation to the number of episodes. The random agent is picking at random, as evidenced by the high peaks and trouts in the first graph. As a result, we can conclude that the random agent does not learn and, despite earning some rewards by randomly reaching the goal state over 500 episodes, the cumulative graph is not consistent with a learning algorithm. The maximum cumulative reward is round 6000 here.

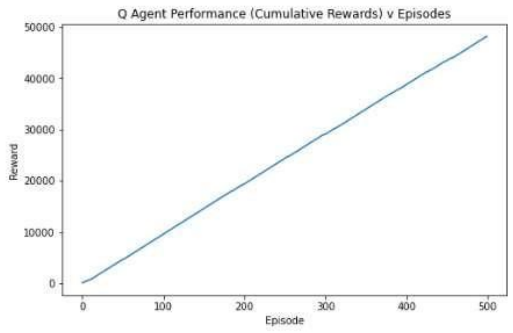




We now run our Q-Learning algorithm over 500 episodes with the parameters alpha = 0.9, gamma = 0.8, epsilon = 0.9, and epsilon decay = 0.99995. We can see that our agent initially explores more with a high epsilon value, which gradually decays as it learns. As shown in the first graph, the agents' learning curve is quite steep. As seen by the linear relationship in the cumulative graph, the maximum reward is consistently achieved as the number of episodes increases. When compared to the random, the maximum cumulative



rewards are also much higher here at 50,000.

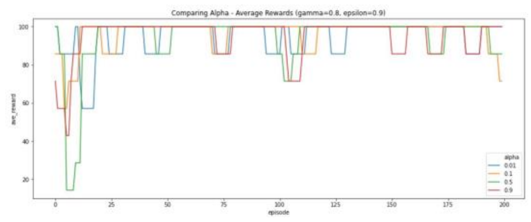


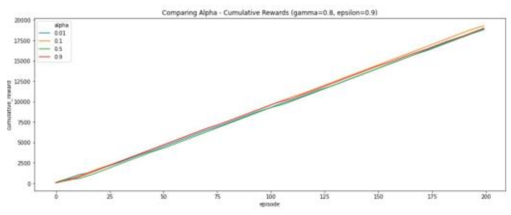
Repeat the experiment with different parameter values, and policies

Experimenting with different parameter values

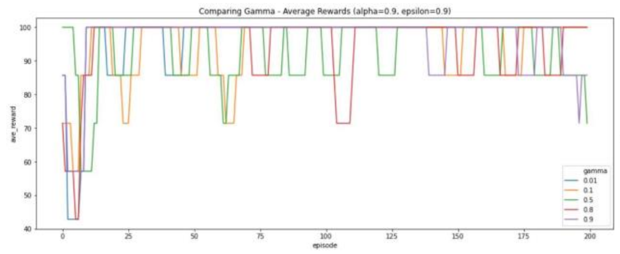
In this section, we experiment with various parameter values and analyse the changes in the reward structure per episode.

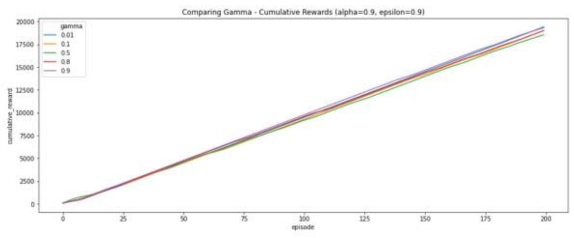
Static: Gamma=0.8, Epsilon=0.9 and Varying: Alpha [0.01, 0.1, 0.5, 0.9] – This value determines how quickly learning occurs, i.e. the learning rate. The graphs below demonstrate how changing the learning rate affects our rewards per episode. As we can see, an alpha of 0.1 yields the best results over 200 episodes, and this line yields the highest cumulative reward. An Alpha of 0.01 results in a higher line initially, indicating that the agent has higher rewards at the start, but a learning rate of 0.1 outperforms the learning rate of 0.01 towards completion.



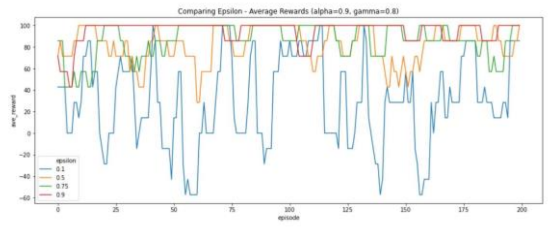


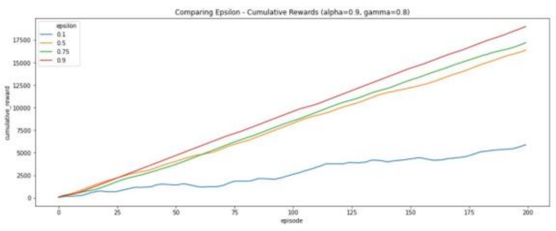
Static: Alpha=0.9, Epsilon=0.9 and Varying: Gamma [0.01, 0.1, 0.5, 0.8, 0.9] – This value determines whether future rewards (higher discount rate) are more valuable than immediate rewards (Lower discount rate). Here we change the gamma/discount factor to see its effect on rewards per episode. We notice the cumulative rewards per episode are pretty similar, this maybe due to our model converging more quickly with only 1 blocker and 15 states. However we notice that a gamma of 0.01 performs pretty well with higher rewards as the model converges after 200 episodes.





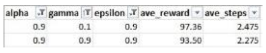
Static: Alpha=0.9, Gamma=0.8 and Varying: Epsilon [0.1, 0.5, 0.75, 0.9] –Here we change the epsilon to see its effect on rewards per episode. We notice the cumulative rewards per episode are highest for an epsilon of 0.9 as it achieves higher rewards than lower epsilon values. This means our agent works best when it explores the environment more as oppose to lower rates of epsilon.





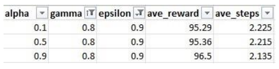
Quantitative & Qualitative Analysis

We notice that as we increase the gamma, it takes less steps for convergence, increases the avg reward per episode for the shortest path (i.e. rewards are recognised immediately).



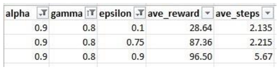
alpha	gamma	epsilon	ave_reward	ave_steps
0.9	0.1	0.9	97.36	2.475
0.9	0.9	0.9	93.50	2.275

High values of alpha give us a high reward for a smaller number of epsiodes meaning learning is taking place quickly and convergence is improving. This means the agent focuses on the latest information and generates a deterministic environment. The best value for alpha here is 0.9 which gives the highest rewards for the least amount of episodes.



alpha	gamma	epsilon	ave_reward	ave_steps
0.1	0.8	0.9	95.29	2.225
0.5	0.8	0.9	95.36	2.215
0.9	0.8	0.9	96.5	2.135

Here we expect epsilon to do better when close to 1 (allows for more exploration of the environment initially). We notice from the below that a higher epsilon value gives us a higher reward as learning is taking place but takes a higher number of episodes to complete.

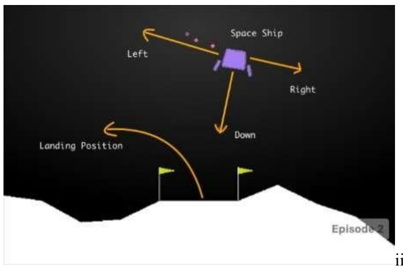


alpha	gamma	epsilon	ave_reward	ave_steps
0.9	0.8	0.1	28.64	2.135
0.9	0.8	0.75	87.36	2.215
0.9	0.8	0.9	96.50	5.67

In comparison, we saw that when assigned to a random policy, no learning takes place and therefore no convergence to optimal path occurs. The most optimal path occurs with a high value of epsilon and gamma (0.9) and an alpha of 0.5. This gives us a high average reward in the lowest number of episodes.

Advanced Task

In this part of the exercise, we aim to solve the LunarLander-V2 environment from Open AI Gym for a continuous control task in the Box2D simulator. The purpose of the game is for the agent to make movements in order to land between the two flags successfully.



Q-learning is often used to solve issues with discrete states and actions, and therefore we do not anticipate this game to run successfully because it includes 6 continuous variables and 2 discrete variables. This is due to the fact that the Q table would be enormous and would take an extraordinarily long time to converge to the accurate q-values.

In this task we will be learning the environment using an algorithm/showing improvements and comparing them. These are the DQN with a soft update target, DQN with a Fixed Target and a Deulling DQN. The lunar lander problem is a well solved problem which shows the lander descending form the top of the screen as the episode begins. The current state (co-ordinates & speed both horizontally and vertically, direction, angular velocities and of course the flags between which it must land) of the lander is provided to the agent in each episode which is an 8-dim vector of real values. The actions that it can take are [Do nothing, fire engine to left, fire engine to right, fire main engine]. If the agent can stabilise these controls, then it can land successfully.

The scoring system is mentioned below which is taken directly from the environment on the open AI website.

"Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine."[iv]

Implementation Details

DeepQN with soft update

As discussed earlier, Q learning begins with the identification of agents at random and the assessment of their impact on the environment. The Q function is then updated using the Bellman equation. DQN is an extension to this. As we start hitting that max q tables with large number of states, the Q learning becomes costly and is limited by its own model limitations. we can apply neural networks. This trains the network to use gradient descent to minimise the temporal difference residual and update the neural network weights. We say this is a better approximate for the Q value as it accounts for the rewards. Here we do a soft update by retaining the existing weights by some Tau which is less than 1. Then (1-tau) * online networks weights are added. The max function in DQN with fixed target has evaluation and action selection values which are the same. Hence we can get overestimated values and causes DQN to be quite unstable. The big disadvantage here is serial correlation in that the samples depend on the current state/action to get to the next state

DeepQN with fixed target

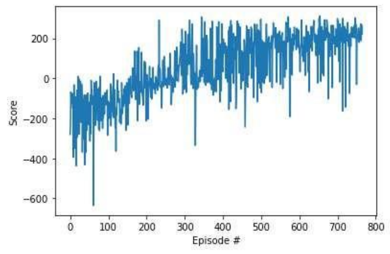
The Q target is generated using a different target network. This is the neural network training's temporal difference target. The weights of the target network, which are used to calculate the value of the state reached as a result of an action, are fixed, and only copied over from the online network on a regular basis. The training becomes more stable as a result, and convergence occurs more quickly. We want to fix the target so we are not chasing our own tail.

Duelling Deep QN (DuellingQN)

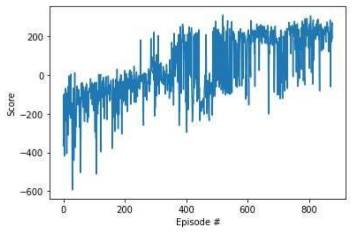
There are two estimators in the duelling network. The first is for the state value function, while the second is for the action advantage function that is state dependent. Duelling networks eliminate the need to learn each and every action for each state by determining which states are worthwhile. Q(s,a) implicitly calculates the total of V(s), the value of being present at a state, and A(s,a), the benefit of doing a specific action at the state. DDQN therefore efficiently learns value of a state by using these two estimates to in turn estimate the Q value. For example if we have an agent in a particular state but it cannot add any value being this state to the final goal then we can eliminate calculating a value of that action. For example in our lunar lander example if the lander is on the right side of the box then firing to the right side would be pointless. This improvement should encourage a faster learning and training. It is our prediction that duelling will be the best performing algorithm here.

Analyse the results quantitatively and qualitatively

We will aim to run for 1000 episodes and get an average score every 100 episodes and to stop training once the target average score of 200 is reached. It was a real time consuming and difficult task to change hyper parameters and running the algorithms overall (through test and final runs). We ran multiple experiments with different combinations of Learning rate, epsilon start, end, decay, target model update, batch size etc. We found our DQN model to have performed pretty well with the soft update. Finding the right parameters were abit of a trial and error. Convergence of our DQN model took 665 episodes and achieved an average score of 200.45.



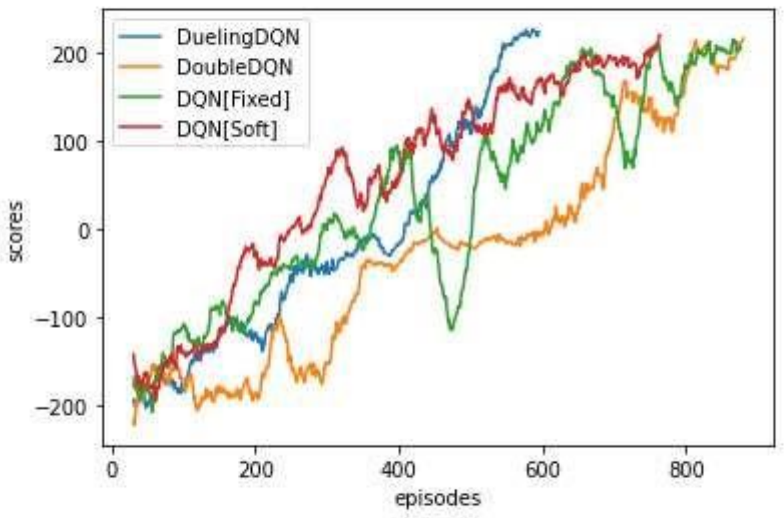
We then ran the tests for DQN by fixing the target and the expectation was that we should achieve an improvement on the DQN with soft update however the training took longer converging at 777 episodes with a perfect score of 200. Therefore it was more accurate but by fixing the weights, gave rise to a longer computation time.



We next moved onto duelling and expected to have by far the quickest convergence and here were able to converge after 496 episodes with a average score of 200.08.

We also ran the Double DQN for good measure but this performed the worst converging in 781 episodes with an average score of 200.66.

Here is what all our results from above looked like. We tried to keep the parameters the same so we can have a good comparison however doing a more extensive hyperparameter search can vastly improve the results. We can see that Duelling rose the most and finished the fasted followed by a soft DQN model. The DQN Fixed Target did well but did have a large drop after the 500 episode. The simplicity of our environment maybe the reason the basic DQN model was enough to perform well.



References

1. Sutton, R. S., & Barto, A. G. (2018). A Beginner's Guide to Q-Learning. *Towards Data Science.* [Online]. Available at: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c (Accessed: [Access Date]).

2. Sahoo, S. (2020). Bellman Equation and Dynamic Programming. *Medium.* [Online]. Available at: https://medium.com/analytics-vidhya/bellman-equation-and-dynamic-programming-773ce67fc6a7 (Accessed: [Access Date]).

3. Verma, S. (2021). Solving Lunar Lander (OpenAI Gym) - Reinforcement Learning. *Medium.* [Online]. Available at: https://shiva-verma.medium.com/solving-lunar-lander-openaigym-reinforcement-learning-785675066197 (Accessed: [Access Date]).

4. OpenAI. (n.d.). Lunar Lander - OpenAI Gym Environment. *OpenAI Gym.* [Online]. Available at: https://gym.openai.com/envs/LunarLander-v2/ (Accessed: [Access Date]).

5. OpenAI. (n.d.). Space Invaders - OpenAI Gym Environment. *OpenAI Gym.* [Online]. Available at: https://gym.openai.com/envs/SpaceInvaders-v0/ (Accessed: [Access Date]).

6. Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529-533. [Online]. Available at: https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf (Accessed: [Access Date]).

7. DeepMind. (n.d.). Deep Q-Networks (DQN). *Papers with Code.* [Online]. Available at: https://paperswithcode.com/method/dqn#:~:text=A%20DQN%2C%20or%20Deep%20Q,each%20action%20as%20an%20output (Accessed: [Access Date]).

8. Ray Team. (n.d.). RLlib Algorithms Overview. *Ray Documentation.* [Online]. Available at: https://docs.ray.io/en/latest/rllib/rllib-algorithms.html (Accessed: [Access Date]).

9. Ray Team. (n.d.). DQN - Ray RLlib Algorithm Documentation. *Ray Documentation.* [Online]. Available at: https://docs.ray.io/en/latest/rllib/rllib-algorithms.html#dqn (Accessed: [Access Date]).

10. Dharmendra, B. (n.d.). A Brief Introduction to Proximal Policy Optimization. *GeeksforGeeks.* [Online]. Available at: https://www.geeksforgeeks.org/a-brief-introduction-to-proximal-policy-optimization/ (Accessed: [Access Date]).

Additional References:

11. PyTorch. (n.d.). Reinforcement Q-Learning Tutorial. *PyTorch Tutorials.* [Online]. Available at: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html (Accessed: [Access Date]).

12. Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602.* [Online]. Available at: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf (Accessed: [Access Date]).

13. Kaptan, V., Pechenizkiy, M., & Puuronen, S. (2020). Multi-step deep Q-networks with experience replay. *Neural Networks, 129,* 273-286. [Online]. Available at: https://www.sciencedirect.com/science/article/pii/S0896627320308990 (Accessed: [Access Date]).

14. California State University, San Bernardino. (n.d.). Space Invaders. *CSUSBDT GitHub.* [Online]. Available at: https://github.com/csusbdt/441-2015/wiki/Space-Invaders (Accessed: [Access Date]).

15. Ray Team. (n.d.). RLlib Colab Notebook. *Google Colab.* [Online]. Available at: https://colab.research.google.com/github/rayproject/tutorial/blob/master/rllib_exercises/rllib_colab.ipynb#scrollTo=i3FVvEJRNlhy (Accessed: [Access Date]).

16. deeplizard. (Year). Exploration vs. Exploitation - Learning the Optimal Reinforcement Learning Policy. *YouTube.* [Online]. Available at: https://www.youtube.com/watch?v=hCeJeq8U0lo (Accessed: [Access Date]).

17. Torres, J. (Year). The Bellman Equation. V-function and Q-function Explained. *Towards Data Science.* [Online]. Available at: [https://towardsdatascience.com/the-complete-reinforcement-learning-dictionarye162

30b7d24e#:~:text=Greedy%20Policy%2C%20%CE%B5%2DGreedy%20Policy,Agent%20to%20explore%20at%20all](https://towardsdatascience.com/the-complete-reinforcement-learning-dictionarye16230b7d24e#:~:text=Greedy%20Policy%2C%20%CE%B5%2DGreedy%20Policy,Agent%20to%20explore%20at%20all) (Accessed: [Access Date]).