INM707: Deep Reinforcement Learning

Coursework

Anthony Hoang
Priyanka Velagala

Table of Contents

Part 1 - Basic Task

1.1 Defining the problem and the environment

The environment chosen is a 6x6 grid world, where the objective of the agent is to find the optimal path from location (0,0) to (5,5) as seen in Figure 1.1. The terrain is designed with additional components the agent must learn about to maximize rewards along its path:

1. **Bitcoins -** on collecting this resource at t = 0, the agent receives a reward of +25. This reward depletes linearly with each time step.
2. **Oil spills -** on traversing a cell with an oil spill, this causes the agent to slip and fall. The time taken for the agent to get back up is reflected by a penalty of -100.
3. **Walls -** are obstructions within the boundaries of the environment that limit the set of actions available to an agent.
4. **Tunnels** - allows a one-way teleportation of the agent from one cell to a non-adjacent cell in one time step where the darker end of the tunnel represents the side with restricted entry.

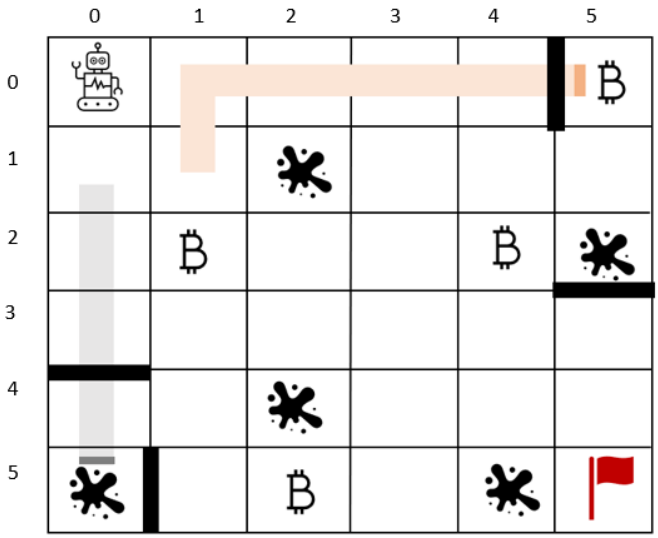


Figure 1.1: 6x6 Grid world

1.2 State transition function

The environment has 36 states,where the states are defined sequentially with S_0: agent at location (0,0), S_1: agent at location (0,1) … S_{35}: agent at location (5,5).

At each location, the set of actions available to an agent are defined by $\mathcal{A}$ = {up, down, left, right} which allow the agent to move to an adjacent cell on the grid. The actions available to an agent at a given state can be restricted by components of the environment such as boundaries or walls. For example, when the agent is at S_{18} (i.e. at location (3,0) as seen in Figure 1.2), the set of actions available to the agent, $\mathcal{A}$ = {up, right}. If the agent chooses the action up, the new state is S_{12}, alternatively, if the agent chooses the action right, the new state is S_{19}.

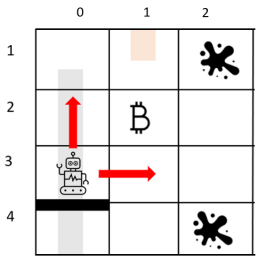


Figure 1.2: Actions available to an agent at S_{18}

To generalize this for all states, we define the state transition function as:

$$s' = t(s, a)$$

Where t is a function which takes the current state s, and the action, a, as inputs and maps them to s' which reveals the next state of the environment.

1.3 Reward function

Each state transition in the environment has a non-zero reward. To ensure the agent finds an optimal path to the destination while disincentivizing traversal of non-optimal paths, each time step has at least a -1 reward. Exceptions to this reward structure vary by component:

1. **Terminal state -** the terminal state, S_{35}, *has* a reward value of +1000. With each time step, the reward depletes linearly with time until it reaches zero. An episode is complete if the agent either reaches this state or the maximum number of steps are reached, whichever comes first.
2. **Bitcoins -** bitcoins can be collected when the agent executes actions that result in state transitions to the following states - S_5 , S_{13} , S_{16} and S_{32} . Similar to the reward structure of the terminal state, these states are initialized with a reward value of +25 and set to be depleted by a unit with each time step until reaching zero.
3. **Oil spills** - when the agent executes actions that result in state transitions to the following states - S_8 , S_{17} , S_{26} , S_{30} and S_{34} , the agent is awarded a penalty of -100 to disincentivize traversal to the destination along that path.

To generalize this structure for all states, we define r, the reward transition function which takes as input the current state s, and action executed by the agent, a, and awards the agent a reward of r' associated with the state-action pair.

$$r' = r(s, a)$$

1.4 Q-learning and its parameters

Q-learning is a model-free reinforcement learning algorithm that allows an agent to learn the value of an action in a given state. Given a problem that satisfies the conditions of a Markov Decision Process (MDP), this algorithm allows the agent to learn to act in an optimal manner by learning the consequences of actions through reinforcements offered in the form of rewards. The objective of the agent is to maximize rewards accumulated.

For our grid world environment, we start by initializing our Q matrix to 0's and then use the following update rule to allow the agent to learn the optimal (action-selection) policy (where the optimal policy is one that yields maximum rewards for the agent):

$$Q^{new}(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \Upsilon * max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

1.4.1 Learning Rate: Alpha

Alpha is the learning rate parameter. This determines the extent to which new information is used to calculate the Q-value of a state-action pair. When $\alpha = 0$, the agent retains the old Q value and disregards the error in estimation as it exclusively relies on exploiting prior knowledge. Whereas when $\alpha = 1$, the agent completely disregards the old Q value, and the Q-value of the state-action pair is updated solely based on information acquired from the most recent state transition (Alonso, E. 2022) .

1.4.2 Discount Factor: Gamma

Gamma is the discount factor which determines the importance of estimated future rewards. This parameter is typically set to a value between [0,1]. When $\Upsilon = 0$, this makes an agent short-sighted, meaning the Q-values are updated based on the existing Q-value and immediate reward only. Whereas $\Upsilon = 1$ allows estimation of future rewards to impact the new Q-value of a state-action pair(Alonso, E. 2022).

1.5 Policies

In this section, we will describe the two action selection (behavior) policies that were implemented and evaluated. Each policy varies in how the agent addresses the exploration vs. exploitation trade-off. Exploration of the state space allows an agent to expand its knowledge base and can potentially lead to better rewards in the long-term, however exploitation allows an agent to maximize immediate rewards. Exclusively relying on a greedy approach can result in the agent being trapped in sub-optimal solutions.

1.5.1 Decaying epsilon-greedy policy

The epsilon-greedy policy addresses the exploration-exploitation dilemma by conditionally choosing a strategy by defining a value for epsilon (ε) between 0 and 1. The policy requires a random number between 0 and 1 to be generated and if the value is greater than epsilon, the agent exploits (i.e. selects the action that results in the highest reward from prior knowledge) and if not, the agent explores.

The decaying epsilon-greedy policy takes this one-step further by initially allowing an agent to explore and over time decays epsilon such that the probability of the agent selecting exploitative actions increases over time. For our problem we initialized epsilon at 0.9 and explored different decay rates.

The moving average of rewards and steps per episode were evaluated (over a 20 episode period) to identify the optimal parameters for the problem. For analysis purposes, during trials $\alpha = 0.5$ and $\Upsilon = 0.5$ were used.

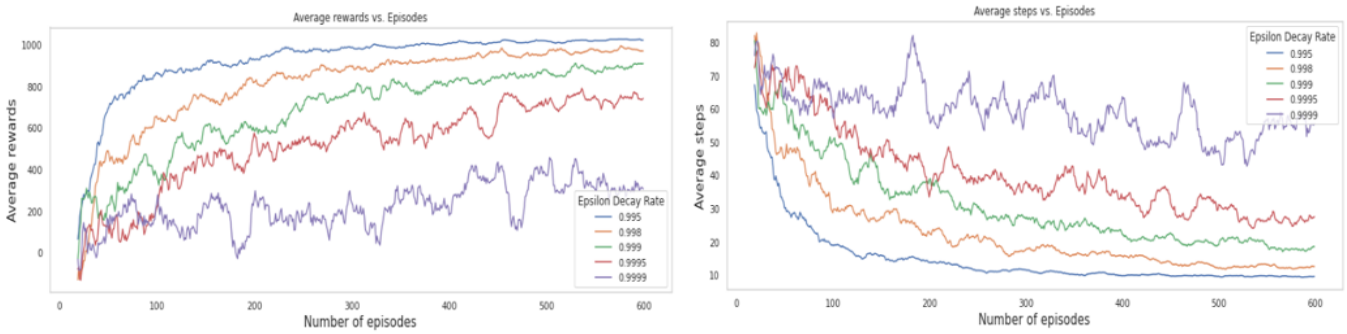


Figure 1.5.1 - Rolling Average of Rewards and Steps by Episode and Epsilon decay rate

All evaluated decay rates were successful at finding the terminal state; however not all of them produced optimal solutions. For example, with a decay rate of 0.9999, on the 600th episode, the agent was still selecting exploratory actions more frequently than exploitative action and thus failed to find an optimal solution. Alternately, with a decay rate of 0.995, the agent uses the episodes early on to explore the state space; however, by the 600th episode, it only makes exploratory actions ~5% of the time and thus is successful at finding the optimal solution. We consider the decay rate of 0.995 to be optimal and will be using that for all further analysis of this policy.

One disadvantage of the decaying epsilon greedy strategy is that when it chooses to explore, it will choose equally among the available actions. This makes it so an action with the worst outcome is as equally likely to be chosen as an action with a best outcome. To address this problem we implement the softmax policy described below.

1.5.2 Boltzmann (softmax) exploration policy

In this policy, actions are assigned a probability based on the Q-values using the Boltzmann distribution. The hyperparameter tau (τ > 0) influences how strongly the relative Q values affect the calculation of probabilities or to what extent an action is chosen randomly. When tau is high, the agent is more likely to choose a random action and when low more likely to pick an action with the highest (expected) reward.

The moving average of rewards and steps per episode were evaluated (over a 5 episode period) to identify the optimal parameters for the problem. For analysis purposes, during trials $\alpha = 0.5$ and $\Upsilon = 0.5$ were used for values of tau between 6 and 14.

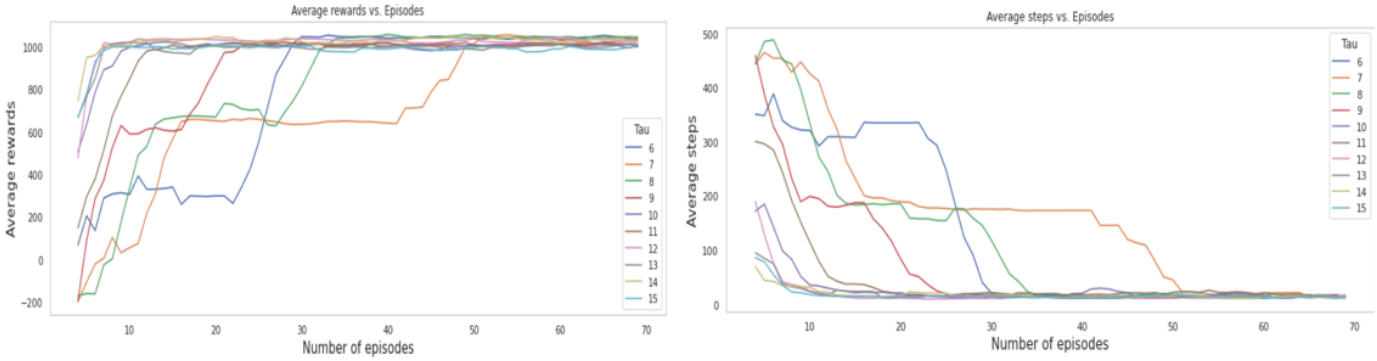


Figure 1.5.2: Rolling Average of Rewards and Steps by Episode and Epsilon decay rate

For this action selection policy the algorithm converges fairly early in the training period with the slowest tau (tau = 7) finding the optimal solution around the 50th episode. In general, higher values of tau appear to be more successful at always finding the terminal state, however for some values of tau we observe some sporadic behavior in terms of which convergence (e.g. tau = 6 converges faster than tau = 7,8). By analyzing both metrics, we choose tau = 14 and use this for all further analysis.

1.6 Results

1.6.1 Metrics used for evaluation

To evaluate convergence of each policy (and its optimal parameter values), for each episode, 3 metrics were tracked:

1. Average number of steps per episode

2. Average rewards per episode
3. If terminal state was reached within the maximum number of steps

Averages were calculated over 3 runs. For graphs, rolling averages were used for easier readability where for the epsilon-greedy policy a 20 episode period was used and for the Boltzmann Softmax a 5 episode period (due to faster convergence).

1.6.2 Q-learning parameters

To evaluate how the Q-learning parameters affect convergence of the algorithm, the epsilon-greedy action-selection algorithm policy was used with a decay rate of 0.995 and alpha and gamma were each set to 0.5 while the other was being evaluated.

By evaluating values of alpha between 0 and 1, we observe that the Q-learning algorithm converges for all values of alpha with some values converging to the optimal solution faster than others. The exception to this is when alpha is 0, the Q-value updates simply involve retaining the old value. As the values are all initialized to zero, regardless of whether the agent selects an exploratory or exploitative approach, the action selection will be random at each iteration as the agent is unable to update its knowledge based on experience and as a result finds the terminal state in only about half the trials (as a result of chance).

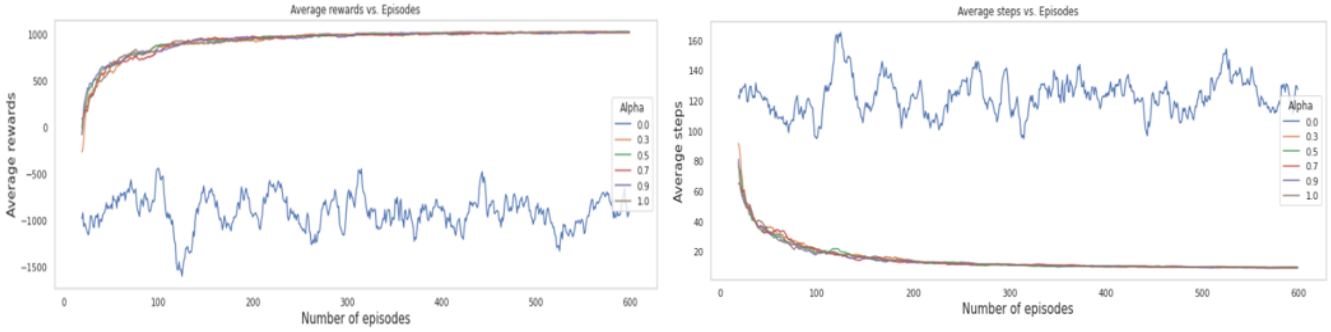


Figure 1.6.1 - Rolling Average of Rewards and Steps by Episode and Alpha

When evaluating values for gamma, we first note that when the agent is myopic, it updates the Q-matrix based on immediate rewards. As a consequence, the agent fails to meaningfully learn about the notion of time penalty while pursuing suboptimal paths. When analyzing gamma =1, this can be interpreted as the agent valuing the estimated future rewards with the same degree of confidence as the immediate rewards. While in the initial set of trials, this approach appears to be promising yielding non-negative rewards, in the long-term, inspection of the metrics show to maximize rewards and find the shortest path, future rewards must be discounted to some degree and that the optimal parameter value for our environment lies between [0.5, 0.9].

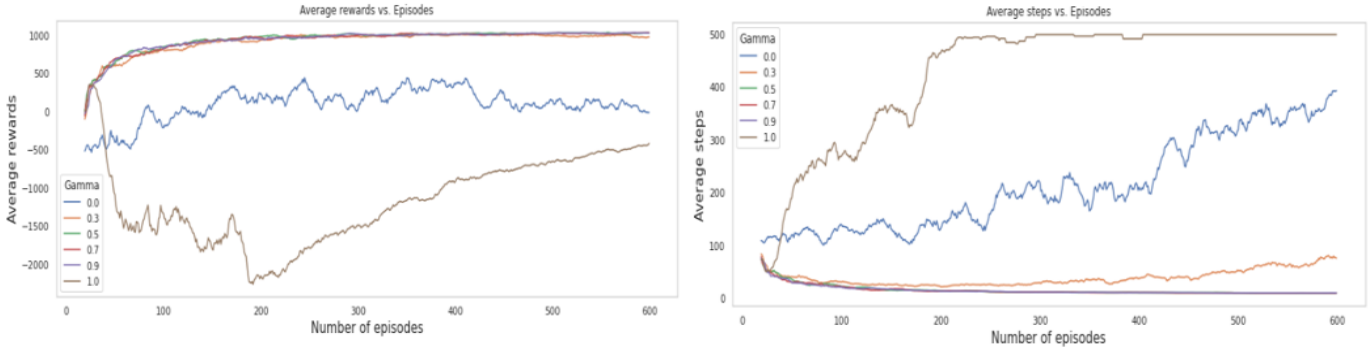


Figure 1.6.2 - Rolling Average of Rewards and Steps by Episode and Gamma

1.6.3 Finding optimal parameter values

To compare the two behavior policies, we used Bayesian optimization to find parameter values for our environment that maximizes rewards. Table 1.6.1 and Table 1.6.2 shows the best parameter values found by the optimizer in executing the algorithm for 100 iterations, and the results obtained in using those parameter values respectively.

Policy	**Reward obtained**	**Epsilon decay/tau**	**alpha**	**gamma**
Epsilon-Greedy	933.82	0.995	0.3052	0.5386
Boltzmann Softmax	1053	8.369	0.8392	0.4185

Table 1.6.1: Optimal values from Bayesian optimization

Policy	Path found	Total Reward	Path length	Average reward	Average steps
Epsilon-Greedy	[6,7,5,11,10,16,22,23,29,35]	1027	10	904.26.2	16
Boltzmann Softmax	[1,7,13,7,13,7,5,11,5,11,10,16,22,23,29,35]	1069	16	951.2	20.1

Table 1.6.2 : Results from using parameter values found by Bayesian optimization

In general, both policies appear to be successful at learning about the different components of the environment that assist them in their goal such as taking the tunnel from S_7 *to* S_5 and avoiding the oil spills at S_{17} *and* S_{34}. The softmax policy accumulates higher mean rewards over the last 100 episodes of training, however the path found is sub-optimal as it revisits the cells with bitcoin several times to collect the reward several times. The epsilon-greedy algorithm appears to converge relatively slower and accumulates a slightly lower mean reward in the last 100 episodes but reaches the destination in fewer time steps.

Over many iterations, we observed that very rarely do the agents collect the bitcoin at S_{32}, which suggests that parameters of both policies need to be amended such that the agent more thoroughly explores the state space in the early episodes. A drawback of the "optimal" parameters found using Bayesian optimization is that the set of parameter values found might not be a true optimal as the optimizer was run for a set number of iterations.

Part 2 - Advanced Task

2.1 DQN

2.1.1 Defining the problem and the environment

For the advanced task we chose the Lunar Landing V2 environment provided by OpenAI. At the start of the environment there is a ship that spawns at the top center of the screen and has an initial velocity that is randomly set. The goal of the

game is to safely land the ship on the landing pad marked by two flags. The game ends if the ship either crashes (body touches the ground), goes off screen, or comes to rest on the ground.

2.2 State transition function

The environment provides eight observations for the state of the environment: The ship's x and y coordinates, the linear velocities x and y, the current angle, the angular velocity, and a boolean for each of the two legs to signify if it is in contact with the ground. Each state represents one frame of the environment.

$$S = \{x, y, v_x, v_y, a, v_a, l, r\}$$

There are four discrete actions available for the agent.

$$\mathcal{A} = \{\text{do nothing, fire the right engine, fire the left engine, fire the main engine}\}$$

A state transition will consist of a set of the eight observations (state) paired with one possible action to result in another state of the environment. The game reaches a terminal state when the ship either crashes (body touches the ground), goes off screen, or comes to rest on the ground.

Rewards are assigned based on how close the ship is to the landing pad and if it reaches a resting state without crashing. The ship has unlimited fuel but there is a negative reward every time an engine is fired. If the body of the ship touches the ground, it is considered a crash and results in a large penalty. Landing the ship inside the flags has a large positive reward whereas safe landing of the ship also has a positive reward.

2.3 Deep Q-Learning

With a state that involves eight different variables, it is not realistic to keep track of a Q matrix because of the curse of dimensionality. In order to train an agent to learn how to solve this environment, a deep Q network will be used. In the previous task we would look at the Q look up table to determine the best action for any given state but will now feed the eight dimensional state into a deep Q network to see the Q values associated with each action from the given state.

We will be performing an ablation study of four different configurations of the DQN to see how well they learn to solve this environment. Since the training for each of these is not always fast, we chose parameter values found from others' work online to guide us.

2.3.1 Deep Q Network with Experience Replay (DQN)

For our base model we will be implementing a standard DQN with experience replay. This lets the DQN store a short term memory of transitions and samples a batch from it periodically to train the network. This is done to break any correlation between sequential inputs of states. This increases the efficiency of learning in the network. To calculate loss, this implementation uses the same DQN to calculate the target value by feeding the next state into the DQN and using the highest value as the target.

2.3.2 Double Deep Q Network (DDQN)

The first improvement upon the vanilla DQN is by adding a second DQN so that there is a separate policy network and a target network. This allows the model to use a second network as a target instead of the same one. This avoids the scenario where the target is always changing while training. The target network is considered frozen and would be used

to calculate the loss for the policy network and update it. Periodically the policy network would be copied over onto the target network.

2.3.3 DQN with Prioritized Experience Replay (PER)

The second improvement upon the model we implemented is the prioritized experience replay. This adds another level of complexity to the experience replay that is already used in the vanilla DQN. The network learns the most from transitions of states where the loss is high and thus causes the biggest updates in the network's parameters. Probabilities are calculated for each of the transitions in the replay buffer where transitions with the highest loss get assigned the highest weights. Sampling from this buffer uses these weights so that each batch will have a greater effect on the training of the network compared to a normal experience replay.

2.4 Results

We trained each configuration of the model until the average score of the last hundred episodes was at least 200. This score signifies the ship has learned to land safely near the launch pad. As expected, the vanilla DQN took the longest to achieve this score. The model took over 1400 episodes to learn how to solve this environment. When adding a prioritized experience replay to the vanilla DQN, the model was able to finish training in almost half the time as a standard DQN. This is comparable to the training length of the model when a second DQN was added to form a DDQN. Though the training lengths are similar for the DQN + PER and DDQN, the graph reveals that the DDQN was much more stable in its performance near the end of training. The DDQN is consistently getting positive reward values between roughly (100, 250) while the range of the rewards for the DQN + PER is about (-25, 250).

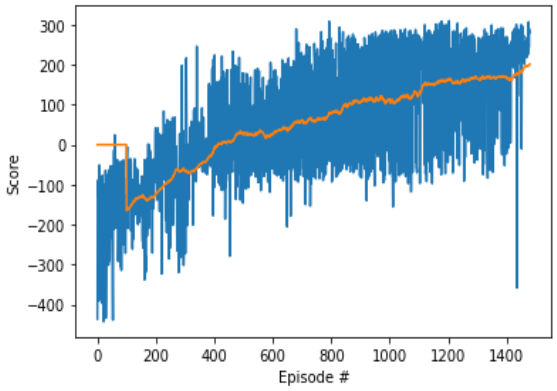


Figure 2.4.1 DQN Training

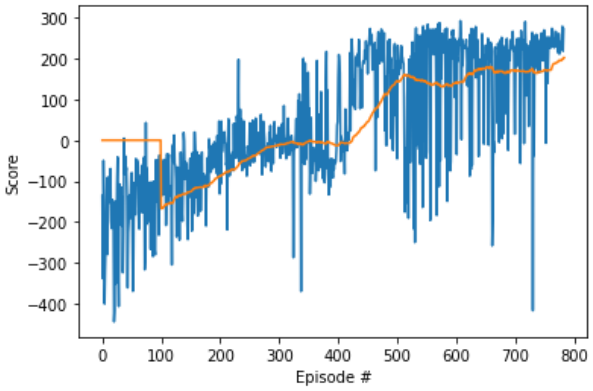


Figure 2.4.2 DQN + PER Training

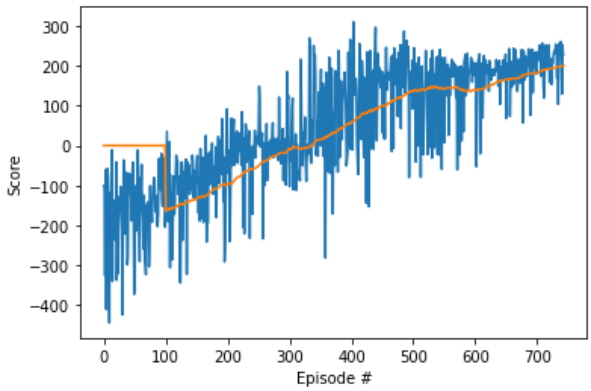


Figure 2.4.3 DDQN Training

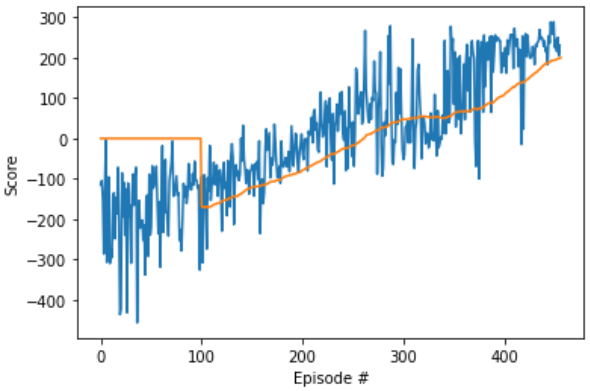


Figure 2.4.4 DDQN + PER Training

The fastest configuration to reach the stopping condition was when we combined the DDQN and PER. It finished training in about 450 episodes, about 1000 episodes faster than the vanilla DQN. The DDQN + PER appears to be stabilizing near the end, however they appear to crash the ship a few times around episode 400.

3.0 Atari Learning Environment

3.1 Background

The game chosen for this task was Breakout from the Atari Learning Environment. The 128 byte RAM from the console was used for training. This observation space stores both the state of the environment as well as the call stack among other components. At any given state, the actions available to the agent are:

A = {do nothing, fire(starts the game), move right, move left}

The objective of the game is to destroy the wall of bricks on the top of the screen by hitting the ball against them. On hitting a brick, the ball bounces towards the bottom of the screen and the player must adjust the paddle such that the ball bounces back towards the bricks or the ball disappears to the bottom of the screen and the player loses a life. The game reaches a terminal state when the player either successfully hits all the bricks or the player loses all 5 lives. The reward structure is set so the higher up the wall a brick is, higher the reward, where bricks on the bottom-most layer have a reward of +1.

3.2 RL Algorithms

The RL Algorithm implemented from rllib was Rainbow DQN. This algorithm works in discrete action spaces as is the case for Breakout. It also uses off-policy learning that allows it to learn from a set of transitions in memory. This allows the algorithm to have good sampling efficiency as the agent can learn from the same state transition multiple times before discarding from memory.

3.2.1 Rainbow Deep Q Network (DQN)

This algorithm uses a neural network to learn the best action to take in a given state by calculating the Q-value of each action and then uses an action selection policy (e.g. epsilon-greedy) to select an action. To set hyperparameters, ray.tune was used to perform a grid search over 4 extensions of DQN introduced in Rainbow DQN (Hessel et al., 2017) and find

the optimal combination of hyper-parameters for this environment. The following 4 enhancements were evaluated on a vanilla-DQN with a single 128 fully connected layer with ReLU activation:

1. Double DQN - separates the policy and target network, tackles the issues observed in DQNs such as the moving target problem and overestimation bias(Hessel et al., 2017)(see section 2.4.2)
2. Prioritized Experience Replay - helps the network identify transitions from which the learning potential is higher. More recent state transitions generally have higher priority compared to transitions that have existed in the buffer for longer periods(Hessel et al., 2017) (see section 2.4.3)
3. Dueling Network - this feature separates approximation of the Q-values to two parts - the value function and the advantage function. Unlike DQN where Q-values are only updated for actions an agent has executed in a state, this network facilitates faster learning by estimating the state values even when only a single action has been taken. This feature shows faster convergence in environments where the action space is relatively large and the the value of different actions are similar.
4. Noisy Nets - this enhancement introduces a linear layer with a noisy stream. The purpose of the noise serves to allow different rates of exploration in different parts of the state space. As the agent learns to ignore the noise, the rate of exploration from those states declines. This type of network performs better than epsilon-greedy in environments with sparse reward settings(Hessel et al., 2017).

Grid-Search of each of the 4 hyperparameters revealed that a combination of PER, Dueling and Noisy Net produced the best results (see Table 3.2). A total of 10 episodes per permutation of each hyperparameter were performed. The trials were run 3 times) to ensure results produced were reliable and counter the effects of randomness of initial state. Also, exploration was turned off during the evaluation period.

Noisy Net	Double-Q	Dueling	Prioritized Replay	Mean Reward
True	False	True	True	5.00

Figure 3.2: Best performing DQN hyperparameters after tuning

3.3 Training and Evaluation

Once the best hyperparameters were found, the agent was then trained until a reward threshold of 5 was reached.

To compare the performance of Rainbow DQN to a SOTA algorithm, the agent was also trained on Proximal Policy Optimization (PPO). PPO is an off-policy, policy gradient method that typically shows fast convergence in a range of Atari games(Tewari, 2020). The default configuration parameters provided by the rllib trainer class for this algorithm was used, with exception to the number of workers which was reduced to 1 to accommodate computational limitations of the hardware.

Figure 3.3 belows shows how the mean reward changes with training for each algorithm. By observing the graph on the left we note that until the 70k episodes mark, the performance of PPO and DQN follow quite closely, with DQN lagging slightly behind. However after that, where PPO continues to increase mean rewards linearly at the same rate, DQN shows periods of increasing and decreasing mean rewards before reaching the set reward threshold of +5. DQN achieves this in ~250k episodes whereas PPO achieves the same results (without any hyperparameter tuning) after ~90k

episodes. This shows that both algorithms are successful at converging to the set reward threshold; however PPO is a faster learner.

The graph on the right shows the mean episode length during training. For DQN we observe that the episode length is particularly high early in the training period which suggests the agent is alive for longer. However as the mean reward during the same training period is relatively low, this can be interpreted as the period it takes for the agent to learn to start the game, that is, picking the fire operation. Once the DQN agent learns this around the 40k episode mark, we quickly note that episode length drops as it hasn't learned to meaningfully operate the paddle based on the location of the ball. PPO on the other hand appears to learn to start the game relatively quickly.

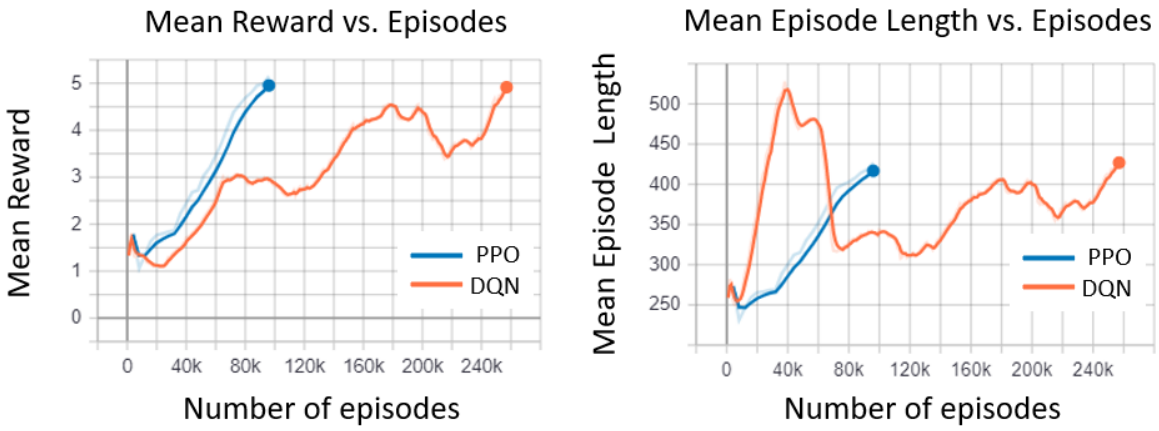


Figure 3.3: Mean Reward vs Number of episodes

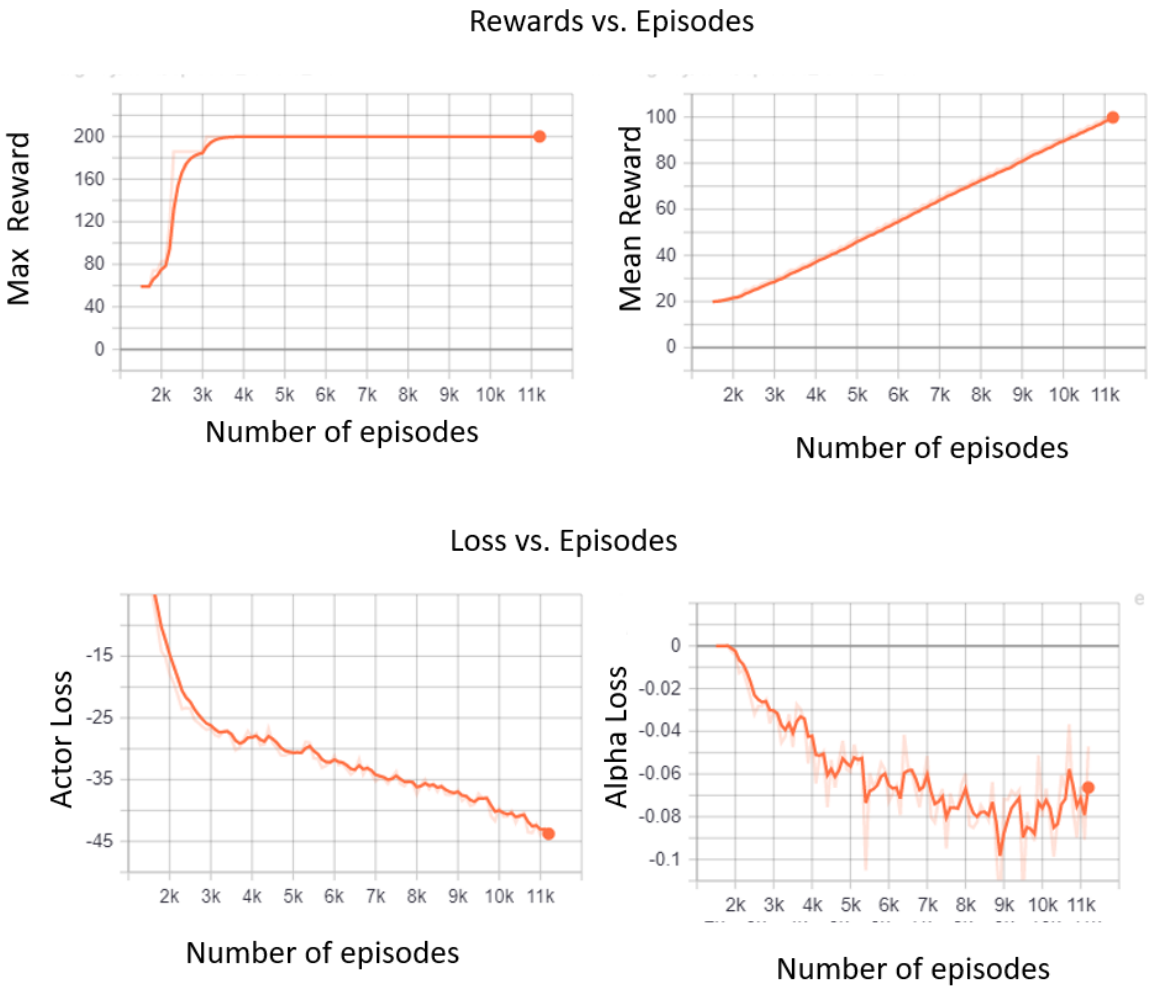
An extension to this would be to conduct a more exhaustive search of other hyperparameters for Rainbow DQN such as experimenting with various network configurations, target network update frequency etc. as it's possible DQN could show improved performance. It would also be interesting to investigate if learning from the console's RAM is just as effective as using the game screen as input.

Part 3 - Extra

Soft Actor Critic (SAC)

Soft Actor Critic is one of the newer reinforcement learning algorithms and claims to be more efficient in terms of samples needed to learn than the traditional RL algorithms. The main characteristic of SAC is that while it maximizes rewards for the environment, it also attempts to maximize the entropy of the policy. The term entropy refers to the randomness of the action selection of the policy. By maximizing this randomness, the algorithm encourages maximum exploration of the environment space. It does so by limiting the probabilities assigned to the actions with higher Q values and keeps weights closer to other actions with similar Q values (V.Kumar, 2019). Many other algorithms try to push the policies to select the actions with higher Q values, but SAC tries to limit this. It tries to find a new balance between exploration and exploitation. This algorithm addresses the problem where an algorithm heavily favors some actions early on and rarely explores the other actions.

This policy was trained on the CartPole-v0 from OpenAI Gym. Hyperparameters were set based on benchmark configuration settings provided by the rllib for this trainer. The graphs below show evaluation metrics over the training period. Training was set to stop once the agent reached a mean reward of 100 which was after around 11k episodes.



Reflection

Our strategy for working on the coursework was to collaborate on the coding aspects of the module which allows us both to gain exposure and hands-on implementation of the underlying concepts.For the written report, each section had a lead writer and was edited by the other person. The following table shows the breakdown of tasks:

	Coding	Written Report Lead	Written Report Editor
Basic Task	Anthony + Priyanka	Priyanka	Anthony
Advanced	Anthony + Priyanka	Anthony	Priyanka
Extra	Anthony + Priyanka	Anthony	Priyanka

References

Github Repository link: https://github.com/PriyankaVelagala/INM707_CW

Watkins, Dayan. Technical note q,-learning - UCL – university college london. . Retrieved April 24, 2022, from https://www.gatsby.ucl.ac.uk/~dayan/papers/cjch.pdf

Barto, Sutton. (2015). Reinforcement learning: An Introduction (2nd edition) . MIT Press

Alonso, E. *(2022, February 16). INM707 Reinforcement learning: the Solution[Lecture Notes]. Moodle.*

V.Kumar, V. (2019, January 09). Soft actor-critic demystified. Retrieved April 24, 2022, from https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665

Tewari, U. (2020, April 25). *Which reinforcement learning-RL algorithm to use where, when and in what scenario?* Medium. Retrieved April 24, 2022, from https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Silver, D. (2017, October 06). Rainbow: Combining improvements in deep reinforcement learning. Retrieved April 24, 2022, from https://arxiv.org/abs/1710.02298

Dueling DQN. . Retrieved April 24, 2022, from https://intellabs.github.io/coach/components/agents/value_optimization/dueling_dqn.html