Assignment 2: Deep Q Learning
Course: Reinforcement Learning 2023

Yuze Zhang s2819368 - Louis Siebenaler s3211126 - Peter Breslin s3228282

1. Introduction

While tabular reinforcement learning (RL) algorithms like Q-learning and SARSA were explored in the first assignment, the problem cases examined were of a low dimensionality. The small size and discrete nature of the state/action space meant it was possible for the agent to learn the optimal policy by simply visiting and updating the Q-values for every state-action pair. However, for high-dimensional problems, the number of entries required in the table grows exponentially and it becomes increasingly difficult to store and update all possible state-action pairs. Therefore, alternative approaches that break away from standard tabular-based methods must be employed to address problems with a high-dimensionality. This has been achieved through the application of deep learning to create the field of deep RL. The first successful demonstration of this is credited to the Deep Q-Network (DQN) algorithm introduced in 2013 at Google DeepMind (Mnih et al., 2013) which combined the Q-learning algorithm with a deep neural network. In this assignment, we examine the DQN algorithm and its application to high-dimensional problems.

2. Methodology

2.1. DQN

Unlike tabular RL which represents the value function as a table with one entry for each state-action pair, DQNs describe the value function with a deep neural network. States are fed as input into the network to produce an output which gives the expected Q-value for each possible action. The vector of Q-values returned can then be used to determine the best action to take in the current state.

The general structure of a DQN follows that of a basic feed-forward neural network. This type of network flows information from the input to the output and propagates the error backwards to update the model iteratively during training. The connections among the neurons are weighted to define the most (or least) meaningful features of the input so that the network is encouraged to learn the most important information. The weighted sum of the inputs is passed through an activation function which enables the model to learn more complex features by applying a non-linear function to the inputs of each layer. Following this, the output error is computed and, through backpropagation, the network is updated by adjusting the weights using an optimizer which works to minimize the loss. In the context of RL, the loss function defines how well the network performs at accurately predicting the Q-values of state-action pairs by minimizing the difference between the predicted rewards and the actual rewards received by the agent.

The DQN algorithm uses the Q-learning update rule to compute the estimates of the Q-values, as given by:

$$y = r_{t+1} + \gamma \max_a Q_{\theta_t}(s_{t+1}, a), \tag{1}$$

where y represents the target value or expected return that the agent is trying to optimize. The equation is composed of the immediate reward r_{t+1} received at timestep $t+1$ after taking an action, and the discounted estimate of future expected rewards given by the product of the discount factor γ (which determines the importance of future rewards) and $\max_a Q_{\theta_t}(s_{t+1})$, the maximum Q-value of all possible actions a in the next state s_{t+1} (where Q_{θ_t} is the estimate of the Q-function at the current timestep for the network, θ).

2.2. Experience Replay

In its most basic form, DQN uses a deep neural network to approximate the Q-values. However, neural networks in their simplest forms all suffer similar pitfalls, including the likes of overfitting and instability. In the context of deep RL, overfitting can occur if the network is trained on a limited set of experiences (i.e. when an agent observes a state, takes an action, receives a reward, and transitions to a new state) and is not able to generalize to new, unseen experiences. On the other hand, instability refers to unpredictable fluctuations in the network parameters, which may cause its performance to become unstable.

Both overfitting and instability can arise if the agent learns from correlated experiences, which is related to the Markovian nature of the environment. Environments with this Markov property are fully described by the current state of the system such that the next state solely depends on the current state and action taken by the agent, meaning it is independent of past states. Experiences can therefore become correlated as the current state-action pair will often be highly

correlated with the previous one. Introducing experience replay into the network can help mitigate these issues. Rather than updating the agent after each experience, this technique stores the agent's experience in a data structure called the replay buffer. Periodically, the agent will then update based on a batch of experiences randomly sampled from the buffer. This enables the agent to learn from a less correlated set of experiences which improves the stability of the learning process. Furthermore, the risk of overfitting is reduced since the agent becomes less biased towards specific sequences of experiences due to the random sampling.

It is worth noting that storing experiences can require an increasingly large amount of memory. Hence the replay buffer does not store them indefinitely but rather has a finite capacity. This is achieved by discarding the oldest experiences when the buffer becomes full, which also encourages a balance between learning from recent and old experiences, thereby reducing the bias that would have existed otherwise. The capacity of the replay buffer is a hyperparameter, and tuning is required to balance the trade-off between memory usage and the quality of the learned policy.

2.3. Target Networks

While experience replay helps reduce the correlation between consecutive experiences, it does not address the remaining risk of correlation between the target and predicted Q-values. As discussed, the target Q-values are used to update the network parameters during the training process. However, the target Q-values are computed using the same network that is being updated, leading to a circular dependency between the target and predicted Q-values. This can lead to poor overall performance with the network becoming unstable and failing to converge to the optimal policy.

To account for this, a separate "target" network is used to compute the target Q-values. This is simply a copy of the main DQN network but with parameters that are held fixed for N iterations before updating to match those of the main network. Hence, there will be a second update rule identical to that given by Equation 1 but for the target network. The target Q-values are computed independently of the main network, mitigating the risk of correlation between the target and predicted Q-values. Fixing the target network parameters for N iterations allows the main network to learn with increased stability due to the target Q-values no longer being influenced by changes in the main network predictions. The target network is updated periodically after N iterations to ensure it remains accurate and up-to-date with the main network. Updating too frequently may introduce instability, while not updating enough can make the target network outdated and inaccurate. Hence, the update frequency is a hyperparameter that requires tuning to balance the trade-off between stability and accuracy in the learning process.

2.4. Environment

For our primary investigation, the `Cartpole` environment from the OpenAI Gym toolkit was used. This is modelled as a 2-D world consisting of a cart able to move along a horizontal track with a pole connected to it that is free to rotate about its joint. Certain physical laws are obeyed such that the motion of the pole is influenced by the movement of the cart. In the context of RL, agent's task is to learn how to keep the pole upright at all times (starting from the upright position). The state of the environment is described by four continuous parameters; the cart's velocity and position, the angle of the pole, and the angular velocity of the pole as it rotates. Conversely, the action space consists of two possible discrete movements; moving the cart to the right or left. Since the goal is to keep the pole upright, the agent receives positive rewards (+1) for each timestep that the cart is within the track bounds and the pole is upright. If either the poll tips beyond a threshold angle of $\pm 12^{\circ}$ or the cart position becomes greater than ± 2.4 (i.e. the center of the cart reaches the edge of the display), the episode terminates.

To further examine the application of DQNs, we also experimented with the `MountainCar` environment, again provided by the OpenAI Gym toolkit. In this environment, a car is stochastically placed at the bottom of a valley for which the agent must learn how to move out of to reach a target position. To climb the steep mountain of the valley, the agent must overcome the force of gravity by using the momentum of the car, making the dynamics of the environment to be of a higher complexity than those in `Cartpole`. The state of the environment is defined by two parameters; the position of the car and its velocity, both continuous variables which give the state space a high-dimensionality. The action space is discrete but unlike `Cartpole`, it consists of three possibilities: the car can be moved to the right, to the left, or the agent can choose not to apply any force to it.

It is worth noting that the reward function differs between both environments. In `MountainCar`, the agent learns the sequence of actions that best accelerates the car up the mountain towards its goal. This is encouraged by giving a negative reward (-1) to the agent for each timestep, thereby motivating it to achieve its goal in fewer and fewer timesteps. The episode terminates if it exceeds a certain number of timesteps or if the car reaches the mountain top, for which it receives a positive reward (+100). It can be noted that this reward function is much more sparse than that of `Cartpole`, which makes it more challenging for the agent to learn the optimal policy since it is forced to rely on trial-and-error to find which actions lead to a positive reward.

2.5. Exploration strategies

Exploration encourages the agent to explore its environment and take actions that it has not yet tried before. This directly

affects the agent's ability in learning the optimal policy and hence is a crucial aspect to consider in any RL model. If the agent did not have an exploration strategy, it would be limited to only those actions it has already tried, meaning that it would be unable to find new, potentially better actions that could maximize its reward function.

2.5.1. ϵ-GREEDY EXPLORATION

One of the most popular exploration strategies is the ϵ-greedy algorithm. This uses a careful balance between exploration and exploitation such that the agent chooses new actions when it explores, while choosing past actions found to be successful when it exploits. The algorithm introduces a tunable exploration parameter, ϵ, into the action selection process to encourage the agent to randomly visit new states or explore different actions. ϵ defines the probability, p, of exploration and typically takes a value between 0 and 1, with higher values encouraging more exploration. With this strategy, a random value between 0 and 1 is determined and compared to the value of ϵ used. If ϵ is greater, the agent is encouraged to explore by taking a random action. If ϵ is smaller, the agent opts for the greedy action given by $\mathrm{argmax_a Q(s, a)}$, i.e. the action with the maximum Q-value for a given state. The algorithm is summarized below:

$$a = \begin{cases} \text{random action}, & \text{with } p = \epsilon \\ \text{argmax}_a Q(s,a)\,, & \text{with } p = 1-\epsilon \end{cases} \tag{2}$$

where the agent selects a random action with probability ϵ, or opts to exploit the greedy action with a probability $1-\epsilon$. Excessive exploration can lead to large amounts of time spent in sub-optimal states or taking sub-optimal actions, which is why it is important to maintain a random chance of exploration. In the previous assignment, it was found that ϵ-greedy with an annealing schedule leads to better results compared to constant ϵ. As a result, we decide to implement ϵ-greedy with a decay constant $\tau < 1$, which modifies the degree of exploration at any episode n as $\epsilon_n = \tau\epsilon_{n-1}$.

2.5.2. UCB EXPLORATION

Upper Confidence Bound (UCB) exploration is another approach for balancing exploration and exploitation so as to enable the agent to better learn its environment. Unlike ϵ-greedy, the UCB strategy does not rely on selecting random actions chosen with a constant probability for exploration. Instead, it shifts its primary focus back and forth between exploration and exploitation based on the knowledge it gains from the environment. The algorithm chooses the action a_t at the timestep t through the following equation:

$$a_t = \text{argmax}_a \left[Q_t(s,a) + c \cdot \frac{\log t}{N_t(a)} \right], \tag{3}$$

where c is a 'confidence' parameter that controls the level of exploration, and $N_t(a)$ is the number of times that a has been selected prior to timestep t. The two factors inside the square brackets represent different terms for exploitation and exploration, respectively. Without the second factor, we see that the algorithm takes the form of the greedy action and chooses to exploit. The second term gives the exploration, where $N_t(a)$ will be small if an action has not been tried very much, meaning that the second factor (known as the uncertainty term) will become large. In this case, the second factor will dominate over the first and the action chosen will tend towards exploration. On the other hand, the confidence associated with the estimate of an action grows as it is chosen more and more often. Hence, $N_t(a)$ will become large, meaning that the uncertainty term will decrease and this action will less likely be chosen. In this case, the first factor will dominate and the agent opts for exploitation. It can be noted that the confidence parameter will control the width of the uncertainty term, with a high c value giving more importance to exploration and a low c value giving priority to exploitation.

2.5.3. NOVELTY BASED EXPLORATION

The novelty search algorithm represents an alternative exploration method. Typically in RL, the agent is rewarded for maximizing a predefined objective, such as balancing a pole in the `Cartpole` environment. However, in novelty exploration, the agent will be given an extra reward if it diverges from prior behaviour, thus creating a constant pressure to do something new. This introduces the novelty metric, which is a measure of the uniqueness of a state. There are different ways to quantify novelty, but typically it is computed with respect to an archive of past states that have been visited by the agent. Novel states are expected to be found in low density regions of the archive state space, and so the sparseness at any point in the behaviour space is a good novelty metric (Lehman & Stanley, 2011). We can compute the sparseness of a state $\rho(s)$ as the average distance to its k-nearest neighbors in state space. Mathematically, this gives

$$\rho(s) = \frac{1}{k}\sum_{i=0}^{k} \text{dist}(s, s_i), \tag{4}$$

where s_i is the i^{th}-nearest neighbour of state s based on the distance measure 'dist', defined as the L2 norm of $s - s_i$. If the novelty of a state $\rho(s)$ is sufficiently high, i.e above some threshold value $\rho_{\min}$, the state is regarded as novel and the agent receives an extra/intrinsic reward. For the `Cartpole` environment, we decided to a give an extra reward of +1 if a novel state is visited. The ideal values of the hyperparameters $\rho_{\min}$ and k are problem dependent and will be investigated in this work in relation to the `Cartpole` environment. The main advantage of novelty based explo-

ration is to encourage exploration of uncharted regions in the state space, which can lead to diverse and creative solutions. Furthermore, it discourages the agent to revisit states, thus helping the agent to escape local optima. Methods involving intrinsic reward are typically very useful when the reward of the environment is sparse.

We note that in novelty based exploration, the action selection itself is still performed by another method, such as ϵ-greedy. The novelty algorithm merely encourages the agent to select an action leading to a novel state.

2.5.4. Curiosity Based Exploration

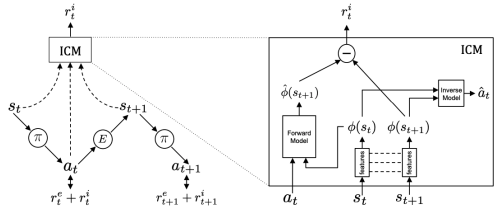

Figure 1. The network architecture of the ICM (Pathak et al., 2017).

The main idea of curiosity-based exploration is to force the agent to explore underexplored states. When the agent arrives at a state, it will first assess whether the state is "familiar". Suppose the state is not "familiar" to the agent, the algorithm designed for curiosity-based exploration will encourage the agent to prioritize the exploration of this particular state over other states. As a consequence, the agent would be able to better understand the environment instead of blindly exploring other states with an assigned probability, such as the ϵ-greedy only approach. To realize the curiosity-based exploration, we need to build a neural network called the intrinsic curiosity model (ICM) with three inputs and two outputs to "test" the agent's knowledge of the current state, followed by a mechanism to set up the exploration policy (Pathak et al., 2017).

To be more specific, Fig. 1 illustrates the exploration strategy of the ICM. When the agent is in a state, s_t, it first takes an action based on the traditional ϵ-greedy action selection policy and returns the next state, s_{t+1}, and an extrinsic reward, r_t^e. Here, r_t^e is the reward provided by the environment itself. The current state, s_t, and the next state, s_{t+1}, will be the first two inputs of the ICM, and after three dense layers ($24 \times 12 \times 4$ neurons), the network returns two feature maps, with the same shapes as the inputs, corresponding to the two states, $\phi(s_t)$ and $\phi(s_{t+1})$. The combination of these two feature maps is used to predict the action which the agent is going to take at s_t, $\hat{a}_t$, using two dense layers (24×2 neurons), where the output shape of $\hat{a}_t$ is the same as all the actions. This predicted action would be compared with the actual action taken at s_t for training the ICM. Our third input is the actual action taken at s_t, a_t, and it is combined with the feature map of the current state, $\phi(s_t)$, to predict the feature map of the next state, $\hat{\phi}(s_{t+1})$. After two dense layers (24×4 neurons), the second output of the ICM has the same shape as $\phi(s_{t+1})$ and is compared with $\phi(s_{t+1})$ for the subsequent training.

To make the training and prediction of the ICM model easier as well as speed up the process, we encode input actions using one-hot encoding. `ReLU` activation functions are used for all layers except for the output layers and the feature map layers. For the two feature map layers as well as the second output layer between $\hat{\phi}(s_{t+1})$ and $\phi(s_{t+1})$, a linear activation function was chosen and for the first output layer between $\hat{a}_t$ and a_t, a `sigmoid` activation function was used (a `softmax` activation function for the first output layer was also tested).

The idea of the curiosity exploration policy is to add an intrinsic reward from the ICM, which is based on the performance of the network in predicting the chosen action and the next state (i.e. $\hat{a}_t$ and $\hat{\phi}(s_{t+1})$). To achieve this, we need to first calculate the loss between the predicted feature map of the next state, $\hat{\phi}(s_{t+1})$, and the actual feature map of the next state, $\phi(s_{t+1})$. The loss function is defined as

$$L_F(\phi(s_{t+1}), \hat{\phi}(s_{t+1})) = \frac{1}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2, \quad (5)$$

which is the half of the norm squared between the two vectors $\hat{\phi}(s_{t+1})$ and $\phi(s_{t+1})$. By contrast, the loss function between $\hat{a}_t$ and a_t is `CategoricalCrossentropy`. $L_F(\phi(s_{t+1}), \hat{\phi}(s_{t+1}))$ is then used for assigning an intrinsic reward, $r_t^i = \eta L_F(\phi(s_{t+1}), \hat{\phi}(s_{t+1}))$, by adding this intrinsic reward to the extrinsic reward, r_t^e. Here, η is the discount factor for the curiosity exploration. The total reward $r_t^e + r_t^i$ is employed for the DQN to calculate the Q-values instead of the original r_t^e.

To implement the ICM inside a DQN, we should first encode the action selected by the ϵ-greedy policy. Then, we need to fit the ICM network with the current state, the next state, and the encoded action as inputs. The two references for outputs of the ICM network are the encoded action, which is the same as the input, and the feature map of the next state, which is generated by the ICM network itself. Immediately afterwards, the fitted ICM network is used to predict the intrinsic reward, followed by adding the predicted intrinsic reward to the extrinsic reward, which acts as the total reward. The rest of the DQN is not altered in any way.

As illustrated, the ICM uses the existing knowledge to check if the agent actually understands the state it is in. If the agent does not have enough knowledge of the state, a "bonus" reward will be added, which consequently changes the Q-value, so the agent can explore this state more often in the future. During training, the agent not only learns to solve

the problem but learns to understand the environment and the process itself.

3. Experimental Procedure

To determine a DQN model which efficiently learns the `Cartpole` game, we tuned several hyperparameters. This includes parameters inherent to the neural network, such as the activation function of the output layer, the number of hidden layers (N_{hidden}), the optimizer, and the learning rate (λ). This will affect the networks ability to approximate and generalize the value function. We did not investigate the impact of the number of neurons per layer, given that this influences the number of trainable parameters and thus, has a similar impact to the number of layers. Furthermore, we choose `ReLU` as the activation function for the input and all the hidden layers given that it prevents vanishing gradients during training, hence making it a popular choice. Additionally, we tuned parameters relevant to Q-learning. This includes the discount factor (γ), the size of the batch of the replay buffer (batchsize), the target neural network update frequency (ν), and the decay constant (τ) associated to the annealing of the ϵ-greedy exploration method. Furthermore, we investigated several alternative exploration methods (see Section 4.2), including UCB, novelty-based, and curiosity-based and investigated several hyperparameters associated to them. During the experiments, we always investigated hyperparameters individually and not their combined effect on the model performance, due to limited computer resources. For all the experiments, the fixed hyperparameters are the following: output activation function = `linear`, N_{hidden} = 3[1], optimizer = `adam`, $\lambda = 0.001$, $\gamma = 0.95$, batchsize = 64, $\nu = 10$, $\tau = 0.995$. We call this setting our baseline model which uses an initial $\epsilon = 1$.

To accelerate training and thus the hyperparameter exploration, we decided to pre terminate a learning episode once the agent has obtained a total reward of 200. In other words, we defined the `Cartpole` game to be won once the agent has managed to balance the pole for 200 consecutive actions. Due to the stochastic nature associated to the exploration of the environment during training, we repeated each training run 5-10 times for 750 episodes, in order to obtain statistically significant results.

For the ablation study, we employed the baseline architecture with ϵ-greedy, with a single modification of $\tau = 0.98$. Furthermore, we raised the condition of wining the `Cartpole` game to a reward of 500 and trained the network for 1000 episodes to ensure that convergence is achieved. For `MountainCar`, we used the same architecture as the baseline model and decided to pre terminate an episode once the agent takes more than 200 actions.

[1]with 512, 256, 64 neurons per layer. When increasing N_{hidden} a layer with 256 neurons is added to the middle of the architecture.

4. Results

4.1. Hyperparameter Exploration

To assess how different parameters influence the training performance of the DQN with replay buffer and target network, we changed parameters within the DQN agent as well as its neural network one at a time, as mentioned in the last section. Performances of different parameters are shown in the first two columns of Figure 2, and the qualifier for a relatively optimal parameter is how fast the agent learns to complete the task required for the `Cartpole` environment. These performances, along with their fundamental causes, will be elaborated on in this section.

We first explored activation functions used for the output layer. Since the network is employed for predicting the Q-values given the agent's current state, it is aimed to tackle a regression problem, and a linear activation function at the output layer is the most commonly used activation function under this condition. As can be seen from the upper-left panel, the linear activation function indeed learns faster, contrary to the others, because of its simple derivative and a lower computation requirement, making network optimization easier. On the other hand, a more complicated activation function is unnecessary for a network with only three hidden layers and might slow down the learning process. Subsequently, a different number of hidden layers was also explored, as illustrated in the left panel of the middle row. Provided we are solving a simple `Cartpole` problem with DQN, in general, fewer layers is preferable. As can be seen, the blue curve indicates the learning process for a DQN with only three layers of neurons, which not only learns faster but offers a more optimal final result than DQNs with more hidden layers. Besides the number of hidden layers, several optimizers were tested, as displayed by the middle panel in the second column. `Adam`, a widely adopted optimization function, results in the best execution with a faster convergence of reward and a better final performance. It also requires less tuning of other parameters, such as learning rate, especially contrasted to `SGD` optimizer. `SGD` requires careful tuning of the learning rate, and it is less stable when handling non-stationary objectives, such as Q-values. Different learning rates were also tested for the network within DQN, which is indicated in the upper-right panel. A smaller learning rate, $\lambda = 0.001$, updates parameters slowly, which presents a more stable convergence and finds a more optimal solution while handling noisy gradients. In general, a neural network with fewer layers, a smaller learning rate, a linear activation function at the output layer, and an `Adam` optimizer is the most optimal setup for DQN with a replay buffer and target update.

In addition to the neural network parameters, the Q-learning parameters' tuning results are also provided in Figure 2. The result of various batch sizes for the replay buffer is shown in the upper panel of the second column. When training the network with small batch size, the agent does not have enough information based on past values and could be easily influenced by the most recent experience. The relatively strong correlation between agent experiences causes the agent to learn sub-optimal and even unstable policies. Therefore, the variance of rewards is larger for a small replay batch size. In contrast, with a large batch size the agent will not be able to update the policy frequently enough to achieve a faster learning speed. As a result, a moderate batch size (64) should be considered. The tuning of the the discount factor γ is shown in the upper panel of the third column. A high γ allows the agent to focus more on the long-term rewards, and a low γ encourages the agent to value the immediate reward instead. The agent learns faster for larger γ but, once it reaches the optimal policy, exploring long-term rewards might lead to instability. Hence, the green curve ($\gamma = 1$) drops significantly after reaching the optimal policy early on, while the curves with a lower γ stabilize after reaching the optimal policy despite a slower learning speed.

The next tuning parameter is the target update frequency, ν. When ν is low, the Q-values estimated by the target network might become outdated, which can slow down the learning process and prevent the agent from adapting to changes in the environment. For a simple environment such as `Cartpole`, a relatively more frequent update frequency is favourable, as can be seen in the middle panel of the third column, and the correlation between two estimated Q-values of the Q-learning and the target network is still small when $\nu = 10$. Apart from the aforementioned Q-learning parameters, the ϵ-greedy decay factor, τ, was also explored, as displayed in the last panel of the middle row. If τ is set too high, the agent will continue to explore the state space rather than learn the task. For a simple `Cartpole` environment, excessive exploration could drastically slow down the agent's learning speed ($\tau = 0.999$), and a higher decay rate is preferred ($\tau = 0.98$). In conclusion, to optimize the learning process of a DQN agent with a replay buffer and target network, we prefer to use a relatively small batch size, a moderate discount factor, a minor target update frequency, and a smaller ϵ-greedy discount for Q-learning parameters.

4.2. Exploration Strategies

The bottom panels in Figure 2 give the learning curves for exploration methods alternative to ϵ-greedy, including UCB (left plots), novelty (2 middle plots) and curiosity-based (right plot). For UCB-based exploration, we investigated the impact of the confidence parameter for various values $c = \{0.3, 0.6, 1\}$ (see equation (3)). We see that for $c = 1$, training occurs the fastest, however it converges to a solution which is slightly inferior to lowers values of c. This suggests that less exploration during training ensures that the agent converges to a better policy. Moreover, we discern that for any value c, UCB exploration yields significantly faster learning than the previously discussed ϵ-greedy method.

For novelty-based exploration, we investigated the hyperparameter of the novelty threshold value $\rho_{\min} = \{0.1, 0.5, 0.75, 1\}$ and the number $k = \{15, 50, 100\}$ for the k-nearest neighbours in state space to quantify the novelty of a state. We recognize that for a low threshold value $\rho_{\min} = 0.1$, the agent learns very slowly and performs worse at the end of training. In this case, the condition for novelty is too weak and the agent is more likely to revisit states that are not very dissimilar from those it has already visited, thus explaining the slow learning. Changing the values of k appears to have no significant impact on the learning process of the agent, at least in the investigated range of k. Although novelty-based exploration is able to solve the game, it is significantly slower than the ϵ-greedy approach and UCB exploration. This may suggest that we have not determined the correct combination of hyperparameters to produce improvements using novelty exploration. However, it is more likely that a method relying on intrinsic reward is unnecessary for an environment like `Cartpole`, where the reward function is well defined and not sparse.

For curiosity-based exploration, to test how different contributions of intrinsic rewards influence the training process, we set the initial value of η to 20% to reduce the time the agent spent on the ICM. We also compared the performance when $\eta = 0.2$ to the performance when η takes on other values, such as 100% and 200%, but overall performances between different η values are roughly the same. For a simple environment like `Cartpole`, too much curiosity is not required to complete the task, and conversely, the agent might spend too much time being "curious" instead of solving the problem. This could potentially slow down the learning process and also requires additional computational power. In addition, curiosity-based exploration is more suitable for an environment with sparse rewards, such as the `Cartpole` environment, so the additional training on understanding individual states would help the learning process when the agent is not rewarded frequently. `Cartpole`, however, rewards the agent every step as long as the pole does not fall below 15° from the vertical position, and, therefore, similar to novelty-based exploration, curiosity-based exploration does not contribute much to the learning process for this particular environment. Despite the inadequate application of the ICM for the `Cartpole` environment, tuning the ICM network might improve the performance of this exploration strategy, given the ICM network hosts more layers and is more complex than the Q-learning network.

In general, we discern that UCB-based exploration outper-

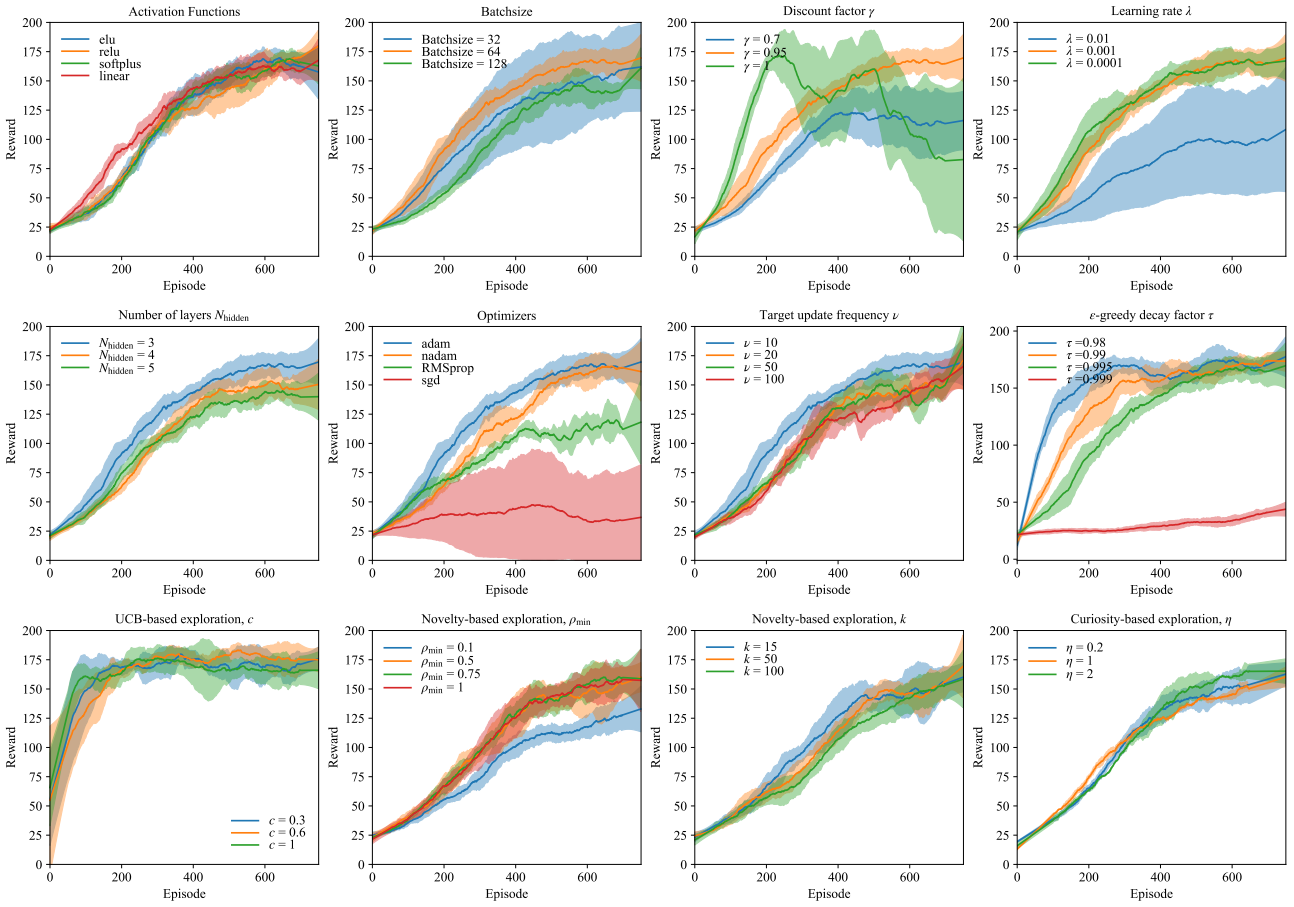

Figure 2. Hyper-parameter tuning & different exploration strategies. Only one parameter at the time is being varied for each learning curve, and the rest are fixed to the values of our baseline model (see Section 3). Each experiment was repeated 5-10 times and the shaded colored region around a curve corresponds to the standard deviation among the repeating training runs.

forms all the other exploration methods, in terms of its ability to quickly learn the `Cartpole` environment. Hence, we conclude that for the parameter space studied in this work, UCB is the preferred exploration method.

4.3. Ablation Study

To better understand the operation of DQNs and how different components impact the results, an ablation study was performed to compare three model complexities. These included a full DQN model with the implementation of both experience replay (ER) and a target network (TN), a model with ER only, and a model with a TN only. Removing parts of the model in this way enables a better evaluation of each model's stability, as well as allowing us to better characterize each component's significance and contribution to the overall performance. As we wish to identify the impact of ER and TN, all other parameters were fixed to ensure that the change in the results were as a direct consequence of the absence of each feature(s).

The left panel of Figure 3 gives the results of the ablation study, where the full DQN model with both ER and a TN is the most stable and performs best. With the inclusion of ER, any correlation between consecutive epochs is diminished and the stability of the learning is improved. Moreover, the inclusion of the TN helps to further stabilize the learning process since any correlation between the predicted and target Q-values is reduced. The model with only ER performs worse, which is likely explained by the lack of a TN since, without its inclusion, the predictions of the Q-values for each state-action pair are influenced by their own updates. The network will struggle to learn in this case due to the instabilities that may arise, making it more difficult to converge to an optimal policy. Finally, the model with only a TN exhibits significantly worse performance. Although the inclusion of a TN helps to add stability to the learning process, the network is expected to overfit without the presence

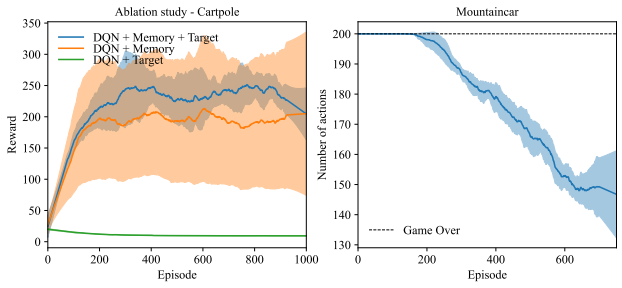

Figure 3. Left: Ablation study for the `Cartpole` environment for DQNs with target network (TN) and memory buffer (blue), memory buffer only (orange), and TN only (green). *Right*: Training curve of the agent for the `MountainCar` environment. We define 200 actions per episode (black dashed line) as loosing the game.

of a ER. The lack of replay buffer means that the network would have overfit to a limited number of correlated experiences, leading to an unstable and inefficient learning process. This is likely to have occurred, since the network looks to have overfit immediately and shows no learning.

4.4. Mountain Car

We applied the baseline model described in Section 3 to the `MountainCar` environment. The right panel in Figure 3 shows the learning curve (blue curve) of the agent. Given the negative definition of reward for an action, we plotted the number of actions taken per episode. The black dashed line indicates 200 actions per episode which we interpret as *Game Over* and the episode is terminated. We discern that for the first ~ 200 episodes the agent is unable to win the `MountainCar` game. However, after this the agent is able to beat the game and progressively improves as it needs less steps to reach the mountaintop. Around episode ~ 650 the agent appears to converge towards a solution. This experiment reflects that our simple DQN model is applicable to a whole range of games. We note that our model relies on a ϵ-greedy exploration approach, however given the sparsity of the `MountainCar` environment using an exploration approach which uses intrinsic reward may yield improved results. Hence, in future work it would be worth implementing a novelty or curiosity-based approach to this environment.

5. Discussion & Conclusions

Within this assignment, the DQN algorithm was successfully implemented for a number of different network architectures and a variety of experiments were performed to address high-dimensional RL problems. From extensive hyperparameter tuning on the `Cartpole` environment, the best network setup for a DQN with a replay buffer and target update was estimated by testing a range of parameters. This optimal setup was found to contain fewer hidden layers, a smaller learning rate, an Adam optimizer, and a linear activation function at the output layer. From tuning the Q-learning parameters, a moderate replay buffer batch size was found to be best due to the balance between reward variance and update frequency achieved. While a higher discount factor helped the agent learn faster, it was noted that this might lead to instability after reaching the optimal policy. A relatively frequent target update frequency is best suited for a simple environment like `Cartpole`, and a smaller ϵ-greedy decay factor is preferred to avoid excessive exploration. Overall, the results provide important guidance for optimizing the learning process of a DQN agent in a RL setting.

In addition to the ϵ-greedy method, three more exploration strategies were investigated and tuned. For any value of the confidence parameter tested, UCB-based exploration yielded significantly faster learning than all other exploration methods considered. This is likely due to the careful exploration-exploitation balance achieved by UCB which is very effective in well-defined reward functions like that of the `Cartpole` environment, while the need for aggressive exploration is more crucial in environments with sparse rewards. This is probably why both novelty- and curiosity-based exploration did not perform as well, since these strategies are generally expected to be better suited for environments with sparse rewards or more complex dynamics. While both methods solved the task and gradually learned over time, they were significantly slower and, generally, hyperparameter tuning appeared to have little impact on the learning process.

The ablation study revealed the full DQN model with both ER and a TN to be the most stable and perform best. The inclusion of ER reduces correlation between consecutive epochs and improves the stability of learning, while the inclusion of TN further stabilizes the learning process by reducing the correlation between predicted and target Q-values. The model with only ER shows the second best performance, while the model with only TN performs worst, struggling to learn anything and overfitting to a limited number of correlated experiences. Hence, a full DQN model with ER and a TN should be used in future work pertaining to high-dimensional RL problems.

Finally, it is worth highlighting the metrics used for testing the performance of our models and identifying areas that needed improvements. In our work, evaluations were made by measuring the reward obtained by the agent over a fixed number of episodes, providing a metric to compare different model architectures. However, during training there will always be a finite probability of exploration, meaning that it is possible for the agent not to fully exploit its learned policy. Hence, an alternative evaluation method that completely turns off exploration at that episode could be used in future work for a potentially more accurate evaluation.

References

Lehman, J. and Stanley, K. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19:189–223, 06 2011. doi: 10.1162/EVCO_a_00025.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, art. arXiv:1312.5602, December 2013. doi: 10.48550/arXiv.1312.5602.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pp. 2778–2787. PMLR, 2017.