DQN for Deep value-based Reinforcement Learning
Reinforcement Learning - Assignment 2

Ioannis Koutalios (s3365530)[1] **Eduard R. Munne (s3687988)**[1] **Maria F. Pinar (s3289850)**[1]

1. Introduction

In this assignment, we will use deep neural networks in Reinforcement Learning (RL). A neural network will be developed as a function approximator where the goal is to learn the optimal action-value function. The input of such a network is the state of the environment, while the output is the action of the agent.

The advantage of using deep neural networks in Reinforcement Learning is that more complex problems can be addressed. In comparison with Tabular Reinforcement Learning, which can only be used in small, low-dimensional state spaces, neural networks can perform in larger state spaces with higher dimensions. They are also more suitable to handle continuous state spaces with no need for discretization which is something that would be required in Tabular Reinforcement Learning.

We will implement one of the most common algorithms for using deep neural networks in Reinforcement Learning which is the deep Q-network (DQN). This is a variant of the Q-learning algorithm which is being used in Tabular Reinforcement Learning. A deep neural network will try to approximate the action-value function by mapping the states to expected rewards for each possible action.

We will be using the "Cartpole" environment from the `gym` module of OpenAI (Brockman et al., 2016; openAI, n.d.). An un-actuated joint connects a pole to a cart, which moves along a frictionless track. The pole can rotate around its pivot point and the pendulum is balanced on the cart by applying forces in the left and right directions. This is an adaptation of an environment described in Barto et al. (1983).

The number of actions that the agent can take is two, either moving to the right or left of its current position. The reward it receives is $+1$ for every step it takes. Each episode can either end when the pole is tilted beyond a certain angle ($\pm 12°$) or if the cart moves too far in the right or left direction (± 2.4). The episode will also terminate if it reaches a maximum number of steps which for the version of Cartpole that will be implemented is 500 according to the documentation (openAI, n.d.). The state space consists of different observations for both the cart and the pole. For the cart, we have its position and velocity, while for the pole we have the angular position and angular velocity. All these values will be encompassed in an array of size four.

In Section 2 we will describe all the different algorithms that we implemented and explain the different concepts that play a role in the performance of our network. Then in Section 3 we will show our results and try to interpret them using general concepts from Reinforcement Learning and Deep Learning. Finally, in Section 4 we will discuss the overview of our experimentation and summarize the things we derived.

2. Methods

In this study, we have employed the PyTorch framework to develop a Deep Q-Network (DQN) model that enables our agent to learn how to balance a pole in a Cartpole environment to keep upright. The Cartpole environment is an open-source environment provided by OpenAI available to Python users through the `gym` library.

The primary objective of our work is to achieve optimal performance in the Cartpole environment using our DQN model, evaluate different addons to the basic DQN algorithm and compare different exploration strategies.

2.1. DQN algorithm

When dealing with high-dimensional problems, traditional tabular learning methods are often insufficient, and more complex solutions are needed. Instead of measuring the expected reward for each state-action pair, we can estimate the expected reward using a neural network that allows us to estimate the Q-values by inputting a state.

Reinforcement Learning entails selecting an action based on previous knowledge from a particular state. When a state has not been visited, all available actions have the same probability of being chosen because Q-values are identical. Nevertheless, this approach becomes unfeasible for high-

[1]Leiden University, P.O. Box 9513, 2300 RA Leiden, The Netherlands.

2nd Assignment for the course of Reinforcement Learning

dimensional problems. Consequently, the convergence of Q-values is expected to occur after visiting various states. By learning from previous states, the model can gradually improve and optimize its performance generalizing to unvisited states.

One such solution for high-dimensional problems is the Deep Q-Network (DQN) algorithm (see Algorithm 1). The DQN employs a neural network that is trained on previous attempts, using the mean squared error (MSE) as the evaluation metric to compare the old update target $y = r + \gamma_{a'}(s', a')$ with the expected reward (Q-value) estimated by the DQN $\hat{y}$ at the actual state s for a total of N steps (Plaat, 2022). Parameter γ denotes the discount factor set at 0.99 for this study. Later, an optimizer is used to find the optimal θ parameters that minimize the error in our DQN model.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{1}$$

The DQN algorithm involves two nested loops. The outer loop is responsible for generating multiple epochs or episodes, during which the model parameters are updated based on the results obtained in each episode.

During each episode, the agent takes a sequence of steps, selecting an action that may involve exploration. Further details on exploration are provided in Section 2.2. The chosen action is executed, and the old updated target y is compared to the expected reward estimated by the DQN model $\hat{y}$. This comparison is used to calculate the loss and converge the model towards optimal parameters that minimize the difference between both values. The episode continues until the pendulum can no longer remain upright, at which point the episode concludes and a new one begins from the initial state.

After a sufficient number of episodes, the agent can accurately predict the optimal action for a given state, and successfully prevent the pendulum from falling.

However, the DQN algorithm poses three significant challenges that must be addressed for successful implementation. The first challenge corresponds to achieving optimal Q-value coverage, as it is not possible to explore the entire state space. Consequently, there may be unexplored state-action pairs that could cause the expected reward to diverge from the actual value.

The second challenge involves dealing with the correlation of subsequent values that are evaluated, which can result in the model focusing towards a specific region of the environment. This bias can unbalance the exploration-exploitation trade-off, causing the model to follow the optimal policy in many cases without exploring further. This, in turn, affects to the first challenge of coverage.

The final challenge is achieving convergence. Since the loss function is the mean squared error (MSE) between two values that are dependent on the model's parameters (the actual Q-value and the old updated target), overshooting the target is a risk.

Successfully addressing these challenges requires implementing effective solutions, and numerous approaches have been proposed in the field of Reinforcement Learning to overcome these issues. In this study, we will evaluate the effectiveness of different approaches in addressing these challenges and their impact on the performance of the DQN algorithm.

Algorithm 1 DQN algorithm (Plaat, 2022)

Input: Exploration type, the number of episodes N_e, the amount of steps S, the discount parameter $\gamma \in [0, 1]$ and the learning rate α.
Return: Rewards ($\mathcal{R}$) for each episode.
Initialization: An empty rewards list ($\mathcal{R}$), an initialized DQN model that we will train with weights W.
$s \leftarrow s_0$
for $e = 1, ..., N_e$ **do**
  $r_e = 0$ {r_e is the reward for each episode, zero at the beginning of each episode}
  for $t = 1, ..., S$ **do**
    $a \sim \pi(a|s)$ {Sample an action}
    $r, s' \sim p(r, s'|s, a)$ {Perform one step}
    $\hat{y} \leftarrow DQN_a(s)$ {Forward pass on the DQN model}
    $y \leftarrow r + \gamma \cdot \max(DQN(s'))$
    $\mathcal{L} \leftarrow MSE(y, \hat{y})$
    $\nabla\mathcal{L} \leftarrow \frac{\partial\mathcal{L}}{\partial W}$
    $W \leftarrow W - \alpha \cdot \nabla\mathcal{L}$ {Backpropagation of the loss}
    $s \leftarrow s'$
    if s is $s_{terminal}$ **then**
      break
    end if
  end for
  $\mathcal{R}[e-1] \leftarrow t$
end for

2.1.1. Hyperparameter optimization

Selecting the optimal combination of hyperparameters is crucial to achieving high performance in Machine Learning models. Before conducting the ablation study, we first searched for the set of hyperparameter values that would maximize the model's performance.

To expedite the search process, we employed the Tree-structured Parzen Estimator (TPE) algorithm using the `optuna` library. TPE is an independent sampling method

that generates a surrogate model to identify the most promising regions for each hyperparameter and then exploits them. Specifically, TPE creates a Gaussian Mixture Model (GMM) for the best performing runs, $l(x)$, and another GMM for the poorer performing runs, $g(x)$, based on the model's performance. By comparing the ratio of these two values, TPE identifies the most promising regions, which are those where the hyperparameter values tend to result in better model performance. This approach allows us to focus our search on the most promising regions, thereby avoiding unnecessary evaluation of irrelevant regions as can occur with a grid search.

We work with a probability density function (pdf), which implies that $\int \frac{l(x)}{g(x)} dx = 1$. This balance between exploration and exploitation is achieved by exploring regions with low probability while exploiting the most promising regions. Initially, the model performs 10 random samples to identify the most promising regions. After that, it selects the hyperparameter values based on the pdf.

When a hyperparameter value results in better performance, the objective function increases, and the ratio $\frac{l(x)}{g(x)}$ becomes more peaked at that region, leading to reduced exploration. This is optimal because it focuses the search on the region that is likely to result in the highest reward.

Ultimately, we obtain the best run, which mitigates the issue of uncorrelated values between different hyperparameters. A value of one hyperparameter may harm or not complement another one, and selecting the best run allows us to optimize all hyperparameters simultaneously.

2.2. Exploration Strategy

In model-free RL exploration is needed to inspect the unknown environment and learn the optimal policy. It is based on adding randomness in the selection procedure of the next action. How to implement this randomness leads to different types of exploration strategies.

The $\epsilon-$greedy policy adds some random selection by the use of the $\epsilon \in (0, 1)$ value, which represents the probability of an action to be randomly selected or using the greedy policy. The mathematical expression for this policy is Equation (2).

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon \frac{|A|-1}{|A|} & if \quad a = argmax_a(Q(s)) \\ \frac{\epsilon}{(|A|)} & otherwise \end{cases} \tag{2}$$

The Boltzmann policy also includes exploration by using the temperature parameter $\tau \in (0, \infty)$. For $\tau \sim \infty$ the policy becomes random and for $\tau \sim 0$ the policy becomes greedy. Mathematically, it can be expressed as the Equation (3).

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{b \in A} e^{Q(s,a)/\tau}} \tag{3}$$

The $\epsilon-$greedy and Boltzmann policies are used in the experimentation. To compare how both policies influence learning, they have been compared using different parameters.

Both strategies use parameters to control the amount of randomness implemented. These parameters can be varied during the train so the amount of exploration is different throughout the experiment. For example, applying more exploration at the beginning and then following the optimal policy -that is, applying more exploitation- can enhance learning.

As an extra experiment, the anneal function has been applied, which varies the parameters from higher to lower, i.e. applying more exploration at the beginning and then more exploitation varying linearly.

For the $\epsilon - \text{greedy}$ policy with linear annealing we have:

$$\epsilon = \epsilon_{final} + (\epsilon_{start} - \epsilon_{final}) \frac{t_f - t}{t_f} \tag{4}$$

where ϵ_{start}, ϵ_{final} are the initial and final values of the exploration parameter,t represents time (number of episodes) and t_f represents the number of episodes after which the annealing terminates and the exploration parameter keeps its minimal value. To calculate this value we use $t_f = (per) \cdot t_{total}$, where t_total is the total number of episodes during the learning process and $0 < per < 1$ is the percentage after which we stop the annealing.

Using an annealing function for the Boltzmann softmax policy is very similar. We only need to change the exploration parameter in the above equations with τ instead of ϵ.

We also implemented a novelty based exploration algorithm as it was described in Tang et al. (2016). More information on the implementation and how this exploration strategy works can be found in Appendix A.2.

2.3. Experience Replay

To address the challenge of correlation when training our model with subsequent actions that are close in the state space and can bias our model, we can use an experience replay buffer. This approach enables the sampling of historical information by utilizing a cache that updates the weights based on non-subsequent information that is uncorrelated. The buffer is an array with a fixed capacity that stores information such as the state, action, reward, next state, and whether the terminal state (pendulum fall) has been reached. At each episode step, we add a new sample to the buffer and

update the DQN weights using random samples of batch size s_b from the buffer, selected uniformly without replace.

This approach allows us to avoid falling into local minima and incorporate supervised learning to our model, as we have features and targets for each sample. Moreover, increasing the buffer's capacity allows us to access older information that is less correlated with the current state, increasing the range of possibilities.

In this study, we compared the DQN's performance with and without the experience replay feature.

2.4. Target Network

In our current approach, we update the model parameters at every step by following the optimal policy for that particular region. However, a different approach that involves infrequent weight updates has been proposed in the literature (Plaat, 2022), which has been shown to yield improved results (Mnih et al., 2015). This method involves cloning the DQN model into a target network, denoted as $\hat{Q}$, after every C steps. This helps to stabilize the algorithm against sudden changes in weights that can arise due to continuous updating, which can result in high oscillations. By delaying the weight update until C steps are done, we achieve greater stability in the model. In this study, we will compare the performance of the DQN algorithm with and without the target network to assess the impact of this additional model $\hat{Q}$ on the algorithm's overall performance.

3. Results

To account for randomness, all plots represent the average of 5 runs with a smoothing of window size of 51 applied. The shaded area indicates the standard deviation at each step across the five runs. All the runs considered were limited to a maximum reward of 500 (consecutive steps) for 1,000 episodes.

3.1. Exploration

To identify the best action selection method, we evaluated six different options, including the basic ϵ-greedy and Boltzmann methods, along with variations that incorporated linear annealing and a novelty-based approach (described in Section A.2). To determine the optimal parameter values for each method, we tested various combinations of ϵ and τ values.

Initially, we experimented with selecting exploration methods by manually trying different hyperparameter values using a trial and error approach. We eventually identified the optimal hyperparameters for each method, which are presented in Table 1 and are applied to all experiments in this subsection.

Hyperparameter	Value
Number layers	2
Hidden units	128
Dropout rate	0.2
Optimizer	Adam
Learning rate	10^{-3}
Batch size	32
Target network	100

Table 1. Manually selected hyperparameters that improve the model's performance and enable evaluation of the best action selection method using fixed values.

In Figure 1, we present the evaluation of the ϵ-greedy method using three distinct values of ϵ (0.02, 0.1, and 0.3), where higher values correspond to an increased exploration of the environment and less reliance on the optimal policy.

The plot illustrates that the best results are achieved with low values of ϵ. Specifically, performance is quite similar for both $\epsilon = 0.02$ and $\epsilon = 0.1$. However, for high values of ϵ (e.g., $\epsilon = 0.3$), performance drops reaching a plateau at a maximum reward of 150 steps without any further improvement in learning after 200 episodes.

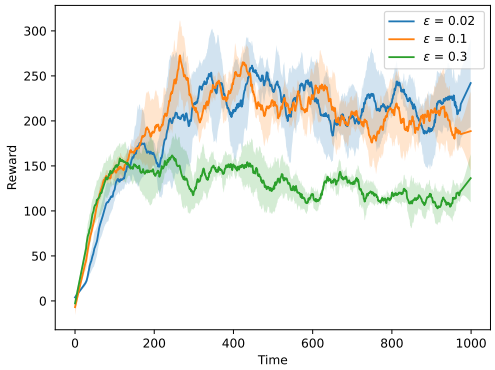

Figure 1. Comparing the average rewards over time for three different values of the ϵ parameter (0.02, 0.1 and 0.3). We can see that lower values, which correspond to high exploitation of the optimal policy, lead to better performance.

We evaluated the simple Boltzmann selection method with three different values of τ: 0.01, 0.1, and 1. Results are shown in Figure 2. The average rewards were fairly consistent across all values, but slightly better for low values of τ, where the model has a higher tendency to exploit the optimal policy without getting stuck in a local optimum.

When applying linear annealing to the ϵ-greedy method, we attempted to explore more at the first stages, gaining more knowledge of the environment and exploiting the optimal

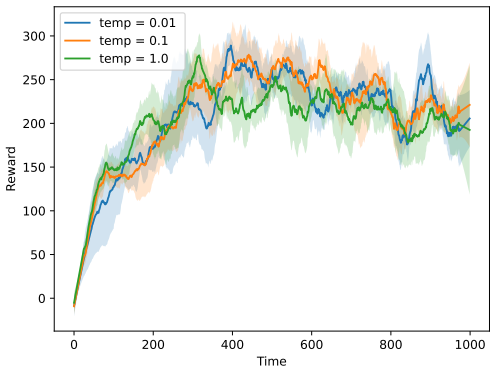

Figure 2. Average reward as a function of time for three different implementations of the Boltzmann softmax policy (τ valued 0.01, 0.1 and 1). Each line represents a different value of the exploration parameter τ. The performance of all models was very similar and only slightly better for lower values.

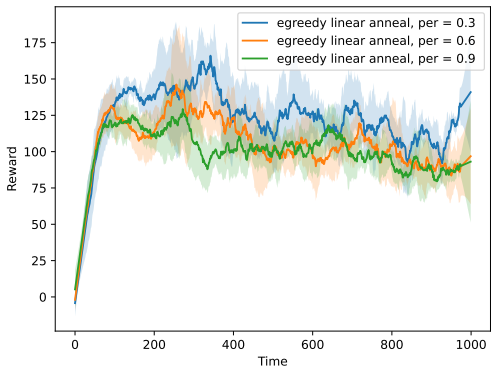

Figure 3. The average rewards over time using the $\epsilon-$ greedy policy with linear annealing. Each line represents a different percentage of the total training time after which the annealing process has finished and the explorations parameter remains constant at the minimal value. Although our agent was able to learn under this selection action policy, the final performance was far from optimal.

policy at the last stages by decreasing ϵ from 0.4 to 0.1 after a certain percentage of the total episodes. We evaluated three different percentages of episodes required to reach the minimum $\epsilon = 0.01$: 30%, 60%, and 90%. The results displayed in Figure 3 show that the model was able to learn. The performance however was less than what we have previously observed in Figures 1 and 2. There could be several reasons for this, such as the wrong selection of ϵ values, being too extreme and causing either too much exploration or exploitation at certain steps, making the model unable to reach the same level of performance as the constant ϵ-greedy method. As a result, we decided to discard this method for the rest of the experiments as the results were not satisfactory enough to compete with the other selection methods.

For the Boltzmann linear annealing method, we chose to vary the exploration parameter from $\tau = 2$ to 0.01. As we can see in Figure 4 the learning process was successful for all the different values of the percentage parameter.

For our last experiment to find the best selection action policy we compared all the different models we implemented. We used the best parameters for each of them training the model for 1,000 episodes. In this experiment, we also included the novelty based exploration strategy for both the $\epsilon-$ greedy and the Boltzmann softmax policy, as it is described in Appendix A.2.

In Figure 5 we can find the results of our experiment. We notice that the performance of all the different implementations is very similar. The agent learns the environment and has a steady performance after a bit more than 200 episodes.

3.2. Hyperparameter optimization

We aimed to identify the optimal hyperparameters that would improve the model's performance maximizing the reward. Using the `optuna` package and a TPE model outlined in Section 2.1.1, we ran a total of 50 trials (with a warmup of 10 trials) to identify the best hyperparameters. Table 2 displays the results of these trials, highlighting the optimal hyperparameters we discovered.

Hyperparameter	Value
Number layers	3
Hidden units	256
Dropout rate	0.2
Optimizer	Adam
Learning rate	10^{-4}
Batch size	32
Target network	75

Table 2. Optimal hyperparameters identified through TPE study after 50 evaluations, with an average reward for the last 20 episodes of 235.85.

We selected the ϵ-greedy method as our default action selection method with a fixed value of $\epsilon = 0.3$. We conducted 500 episodes with a maximum episode length of 500 steps to explore the optimal combination of hyperparameters under the same conditions for all experiments, including the experience buffer and the target network.

More details on the hyperparameter optimization method can be found in Appendix A.1.

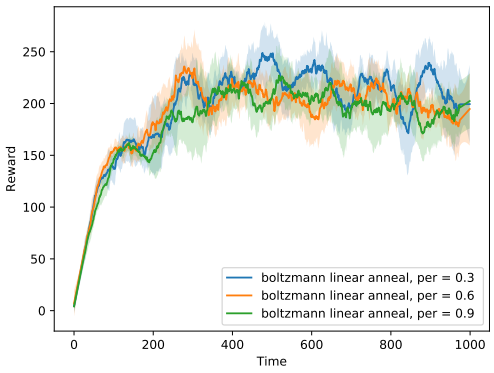

Figure 4. Comparing the average rewards for different percentages when using the Boltzmann softmax policy with linear anneal. The performance of all the different values is very similar, as they were all able to learn efficiently.

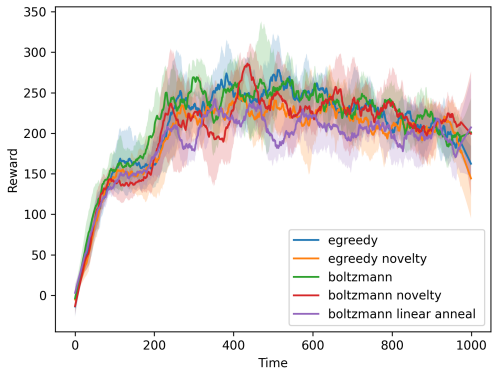

Figure 5. The rewards over time for all the different selection action policies that were implemented. For each model we used the best performing value of the exploration parameter. We see that they all managed to achieve similar performance and learn after a short number of episodes.

3.3. Ablation study

To evaluate the impact of both experience replay and target network on the DQN model's performance, we conducted an ablation study comparing the complete DQN (labelled as DQN in the plots) to simpler models. To denote the removal of a specific feature, we used '-' as a notation, where ER represents the experience replay buffer and TN is the target network. Therefore, DQN-ER refers to the model without experience replay (i.e. simple DQN with target network).

The results of the ablation study, presented in Figure 6, demonstrate the performance of different variants of the DQN algorithm, including DQN-ER-TN, DQN-ER, and DQN-TN. As expected, the complete DQN model, which includes both the experience replay buffer and target network, outperforms simpler models, with the exception of the DQN-TN. These results suggest that while the presence of a target network can improve performance, it has less impact than the experience buffer to achieve a high performance in the DQN algorithm.

The performance of the DQN with target network did not show a significant improvement over the simpler method, which may have been caused by an incorrect implementation of the target network or a suboptimal selection of the update step. However, upon further evaluation, we discovered that the poor performance was primarily due to the biased hyperparameter tuning approach. Our hyperparameter optimization was developed with the complete DQN model in mind, and thus, the selected hyperparameters were optimal only for that specific case and not necessarily for all the models. To address this issue, we compared the performance of the DQN model with the optimal hyperparameters (Table 2) to that of the DQN-ER-TN and DQN-ER models with the manually selected hyperparameters (Table 1). The results of this comparison are shown in Figure 7.

The findings presented in Figure 7 support our hypothesis that the target network does not significantly improve the performance of DQN-ER-TN. Nonetheless, the model appeared to be more stable than the simple DQN, exhibiting fewer peaks due to the delayed update of the target DQN as discussed in Section 2.4 and reaching higher rewards than the ones with the optimal hyperparameters (Table 2).

The combination of ER and TN addresses some of the limitations of the simple DQN method, as explained in Section 2.

3.4. Model Evaluation

As a final evaluation, we wanted to test the performance of our model after the learning process is completed. We chose our best-performing model, the DQN with both the ER and the TN, combined with the Boltzmann softmax policy for the selection action policy ($\tau = 0.1$). Then, we let the model learn and saved its parameters after the training was completed. Finally, we let the agent run without learning and with a completely exploitative selection action policy ($\epsilon - \text{greedy}$ with $\epsilon = 0$) so that our agent is always exploiting the optimal policy.

We performed $1\,000$ repetitions for a maximum of 500 steps and measure the returned rewards. The result was always 500 which is the maximum value for the reward after each episode. This leads to the conclusion that our model was able to learn the optimal policy, which our agent is following during this evaluation.

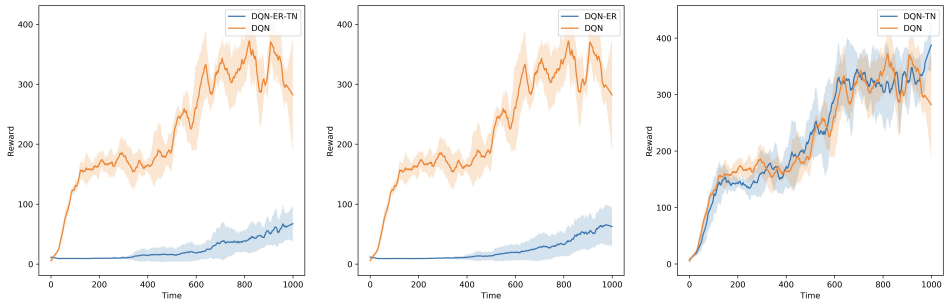

Figure 6. Ablation study comparing the performance of DQN, DQN-TN, DQN-ER, and DQN-ER-TN with optimal hyperparameters (Table 2) after removing the experience replay (ER) and target network (TN). DQN-TN shows similar performance to DQN, indicating that the presence of a TN is not as crucial as the experience buffer. However, the absence of the replay buffer heavily penalizes DQN-ER-TN and DQN-ER, resulting in significantly lower performance.

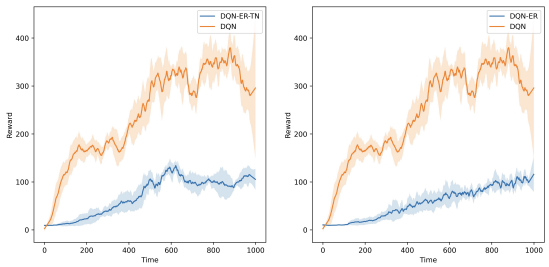

Figure 7. Ablation study with DQN using optimal hyperparameters (Table 2) and DQN-ER-TN and DQN-ER using manually selected hyperparameters (Table 1).

4. Discussion

Four different experiments have been carried out in order to develop and train a Deep Q-Network (DQN) model. The first experiment consists of studying the different exploration strategies as described in section 3.1. The ϵ-greedy strategy presents evident differences in performance for different values of epsilon (see Figure1). The best results correspond to values that allow greater exploitation. The same result can be extrapolated to the rest of the strategies, although the discrepancy between performances using different parameters is less noticeable. One reason behind this is the way the two selection-action policies are exploring. In ϵ-greedy we have a completely random exploration, while the Boltzmann policy uses a weighted probability to explore. This means that after a certain number of episodes the Q-values of the optimal policy will be much higher than the rest and the exploration will be lower. This result can indicate that aggressive exploration prevents exploiting the optimal policy. A similar conclusion can be drawn from the results obtained with the novelty-based policy. It is observed that this policy do not outperform simpler methods by applying more exploration, i.e., increasing the reward for states that have been less visited. In general, it can be concluded that strategies that favour exploitation obtain better results.

Although by applying annealing we would expect an improvement in performance, we observe that the reward obtained is significantly lower for all strategies (see Figure 3 and Figure 4).

The best hyperparameters have been obtained for full DQN. Thus, the results obtained with DQN-ER, DQN-TN and DQN-ER-TN will be influenced by the hyperparameter selection. It is interesting to point out the need to apply regularization in the DQN network. When training, we observed that the network presents overfitting, causing the reward to decline drastically. This was solved by applying a dropout after each layer. The consequence of the hyperparameter tunning can be observed in the ablation experiment (see Figure 6 and Figure 7). The DQN algorithm without TN and ER seems to learn, but as expected, it does not reach the same values as the full DQN algorithm. The noise observed in DQN is corrected when applying TN, although no better performance is obtained. However, by using DQN with ER a better performance is obtained. This is due to the break in the correlation between the points.

A last experiment is done by performing the model after the learning procedure. A total of 500 rewards are obtained in each iteration, leading to the conclusion that our model has learned the optimal policy. During the training procedure, the peak average reward that was observed was around 400. When we evaluate the model we always get the maximum reward of 500 due to no exploration.

In this assignment, we successfully developed a DQN network and applied several techniques to address the challenges faced by the algorithm, such as coverage or correlation, by incorporating a replay buffer and a target network.

The resulting model was able to reach the maximum reward, which involved maintaining a pendulum upright for all iterations. As a result, we can conclude that the agent was able to perform the task it was trained for without errors.

A. Bonus

A.1. Hyperparameter tunning

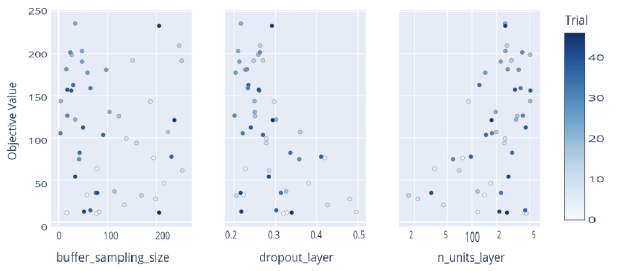

Figure 8. Analysis of the three most important hyperparameters and their impact on the objective value. Each point represents a trial and its corresponding objective value, defined as the average reward over the last 20 episodes. The abscissa displays the value of the corresponding hyperparameter, allowing us to identify the regions that provide more optimal results.

A.2. Novelty based exploration strategy

A novelty based exploration strategy has been implemented by using a count-based algorithm described in Algorithm 2. The original code from Tang et al. (2016) has been adapted to our code. This strategy is based on adding an intrinsic reward for states that have been visited fewer times. This way, the algorithm is encouraged to visit novel states. The expression used for the rewards is Equation (5), which is updated after every episode for each specific state.

$$r_t = r_t^e + \beta r_t^i \tag{5}$$

This policy needs to count how many times each state has been visited. In the cartpole environment, the states are defined in a continuum. To discretize them for counting we use a simhash function 6 that maps the values as hash codes by previously rounding the values to 1 decimal.

$$\phi(s) = sign(Ag(s)) \in \{-1, 1\}^k \tag{6}$$

A.3. Double DQN

As discussed in Section 2.1, the evaluation metric for the selected action was $y = r + \gamma \cdot \max_{a'}(Q(s', a'))$, where Q is the policy DQN network with parameters θ that receives the next state s' as input. However, this approach tends to be overoptimistic since we are using the same model

Algorithm 2 Count-based exploration through static hashing (Tang et al., 2016)

Initialised the counts dictionary
for each e in episodes do **do**
  Collect a set of state-action samples $\{(s_m, a_m)\}_m^M$ with policy π
  Compute hash codes through SimHash $\phi(s_m) = sign(Ag(s)) \in \{-1, 1\}^k$
  Update the counts in the dictionary
  Update the policy rewards using $r_t = r_t^e + \beta r_t^i$
end for

for selecting and evaluating actions, and thus, the same parameters θ.

To mitigate this issue, we can adopt a similar approach as the target network proposed in Section 2.4 by creating a second network called the double DQN. The double DQN, denoted as Q' with parameters θ', will be used to calculate the Q-values for the next state s', and thus, reducing the overoptimistic consideration (van Hasselt et al., 2015). So the selected action for the evaluation metric is obtained with Q' instead of Q. To update the double DQN Q', we can follow the same procedure as the target network and update its parameters ($\theta' = \theta$) after a fixed number of steps C.

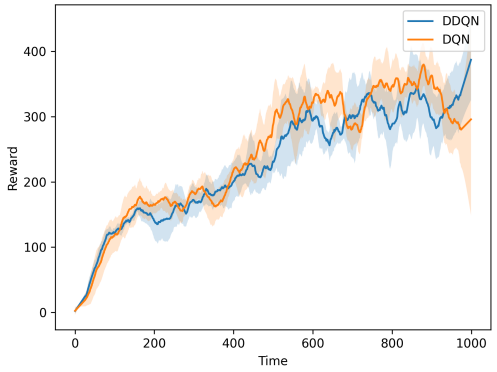

Figure 9. Comparison between the DQN (including ER and the TN) and the DDQN performance after 1,000 episodes with optimal hyperparameters from Table 2.

We trained the Double DQN (DDQN) model for 1,000 episodes using the same hyperparameters as the DQN model with the addition of a replay buffer, as listed in Table 2. However, the resulting performance was similar to the DQN model and did not show any significant improvement. One potential explanation for this is the fixed maximum length of 500, which had already been achieved by the DQN model as demonstrated in Section 3.4. Consequently, the DDQN model was not able to outperform the DQN model that incorporated both the target network and experience buffer.

References

Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015. doi: 10.1038/nature14236.

openAI. Cart pole#, n.d. URL https://www.gymlibrary.dev/environments/classic_control/cart_pole/.

Plaat, A. *Deep Reinforcement Learning*. Springer Nature Singapore, 2022. doi: 10.1007/978-981-19-0638-1.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, X., Duan, Y., Schulman, J., Turck, F. D., and Abbeel, P. #exploration: A study of count-based exploration for deep reinforcement learning. *CoRR*, abs/1611.04717, 2016. URL http://arxiv.org/abs/1611.04717.

van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning, 2015.