Deep Q Learning
Reinforcement Learning, Master CS, Leiden University (2023)

Dinu Catalin-Viorel

1. Introduction

In recent years, deep reinforcement learning has shown remarkable success in solving complex decision-making problems in various domains including robotics, games, and autonomous systems. One of the most prominent algorithms in deep reinforcement learning is the Deep Q-Network (DQN), which combines deep neural networks with the Q-learning algorithm to learn optimal policies. This method appeared in the context of huge state spaces(even continuous) that cannot be saved in a table, so tabular methods like Q-leaning and Sarsa cannot be used to compute the State-Action value function $Q(s, a)$.
This assignment aims to implement the deep Q-learning algorithm in the cart pole environment, a classic control problem that involves balancing a pole on a moving cart. Through this implementation, we will explore the key concepts and techniques involved in deep Q-learning, such as experience replay, target networks and neural network architectures for approximating Q-values. We will also investigate the impact of hyperparameters such as learning rate, and exploration strategy on the performance of the algorithm.

1.1. Environment

1.1.1. Description

The cart pole environment is a simple simulation consisting of a pole balanced on a cart that can move horizontally on a frictionless track. The objective is to control the cart's movement such that the pole stays balanced for as long as possible.

1.1.2. State and Actions

The state space of the environment is represented by a 4d vector, which includes:

- The cart's position (x).
- The cart's velocity ($\dot{x}$).
- The pole's angle (θ).
- The pole's angular velocity ($\dot{\theta}$).

The action space of the environment is discrete and consists of two possible actions: move the cart to the left or right these actions are represented by the integers 0 and 1 respectively.

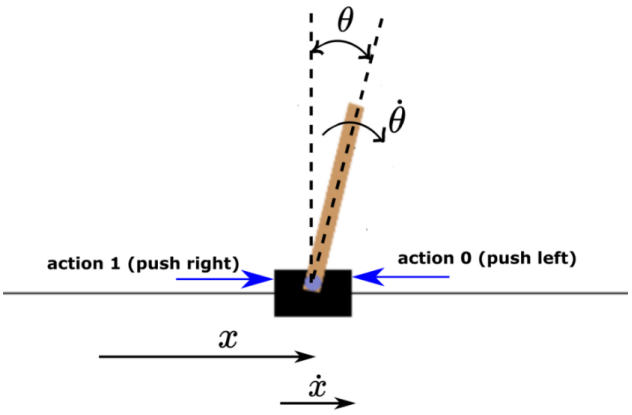

Figure 1. CartPole OpenAI Gym Environment

1.1.3. Rewards

In the Cartpole environment, the reward function is made to give the agent both positive and negative feedback for successfully balancing the pole. The function is defined as follows:

- The agent receives a reward of 1 for maintaining the pole's upright position at each time step.
- If the pole collapses or the cart veers off the path, the episode is over. The agent is rewarded 0 in either scenario.

1.1.4. Objective

The objective of the agent is to maximize its cumulative reward throughout an episode, which is a measure of how long it was able to balance the pole. The threshold for rewards is 500, the episode terminates after the agent can keep the pole upright for 500 consecutive time steps.

2. Background

Usually, the problems solved by Reinforcement Learning are sequential decision problems. Markov decision processes model those sequential decision problems because they have the Markov property (the next state depends only on the current state and the actions available on it). Similar to the formalisation of the Markov Decision Process (MDP) from the book (Plaat, 2022), we will define:

$$(\mathbb{S}, \mathcal{A}, T(s'|s,a), r(s,a,s'), \gamma) \rightarrow \text{MDP} \tag{1}$$

- $\mathbb{S} \rightarrow$ finite set of legal states
- $\mathcal{A} \rightarrow$ finite set of legal actions
- $T(s'|s,a) = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a) \rightarrow$ probability that action a in state s at time t will transition to state s' at time $t+1$
- $r(s,a,s') \rightarrow$ reward received after action a transition from state s to s'
- $\gamma \in [0,1] \rightarrow$ discount factor

An additional important concept of reinforcement learning is policy function π, which describes how an action from state s is chosen.

$$\pi : \mathbb{S} \rightarrow p(\mathcal{A}) \tag{2}$$

Where, $p(\mathcal{A}) \rightarrow$ a probability distribution function over $\mathcal{A}$. $\pi(a|s) \rightarrow$ probability to take action a in state $s (s \in \mathbb{S})$.
An agent that operates in the environment samples traces(τ_t^n) (a sequence of actions, their rewards and the state after action). So, at time t, the agent is in the state s_t, it will repeat the following procedure for n times: It will choose an action a_t which moves it in a new state $s_{t+1} \sim T(s_t, a_t, s_{t+1})$ and it will get o reward of r_t after following the action a_t in state s_t.

$$\tau_t^n = \{s_t, a_t, r_t, s_{t+1}, ..., s_n, a_n, r_n, s_{n+1}\} \tag{3}$$
$$\tau_t = \tau_t^\infty \tag{4}$$

The sum of the cumulative reward of a trace is called a return(the overall reward got by the agent that follows a certain trace)

$$R(\tau_t) = r_t + \sum_{i=1}^{\infty} \gamma^i * r_{t+i} \tag{5}$$

Informally, the goal is to find the optimal policy $\pi^\star$, such that it maximises the return of τ_0(the trace starting in the initial state s_0).
We also introduce two functions:

- State value $V^\pi(s)$ This function returns the expected return value of an agent starting in the state s that follows the policy π;

$$V^\pi : \mathbb{S} \rightarrow \mathbb{R} \tag{6}$$
$$V^\pi(s) = \mathbb{E}_{\tau_t \sim p(\tau_t)} [R(\tau_t)|s_t = s] \tag{7}$$
$$V^\pi(s) = \mathbb{E}_{\tau_t \sim p(\tau_t)} \left[\sum_{i=0}^{\infty} \gamma^i * r_{t+1} | s_t = s \right] \tag{8}$$

- State-Action value $Q^\pi(s,a)$ This function returns the expected return value of an agent starting in the state s choosing action a and following the policy π;

$$Q^\pi : \mathbb{S} \times \mathcal{A} \rightarrow \mathbb{R} \tag{9}$$
$$Q^\pi(s,a) = \mathbb{E}_{\tau_t \sim p(\tau_t)} [R(\tau_t)|s_t = s, a_t = a] \tag{10}$$
$$Q^\pi(s,a) = \mathbb{E}_{\tau_t \sim p(\tau_t)} \left[\sum_{i=0}^{\infty} \gamma^i * r_{t+1} | s_t = s, a_t = a \right] \tag{11}$$

Formally, the goal of reinforcement learning is to find the best policy $\pi^\star$

1. $\pi^\star(a|s) = \text{argmax}_\pi V^\pi(s_0)$
2. $\pi^\star(s) = \text{argmax}_{a \in \mathcal{A}} Q^\pi(s,a)$

Different from tabular q-learning, now we are dealing with a state space $\mathbb{S}$ that is too big, and the idea of representing the State-Action function $Q(s,a)$ as a table is not achievable anymore. To estimate the State-Action values of a network for those big state spaces, neural networks can be used to map a state to the corresponding values.
We consider:

$$Q_\theta : \mathbb{S} \rightarrow \mathbb{R}^{|\mathcal{A}|} \tag{12}$$

$Q_\theta \rightarrow$ the network with weights θ,
$Q_\theta(s,a) \rightarrow$ the estimation of the value of State-Action function for state s and action a using the weights θ

3. Deep Q-Learning Network

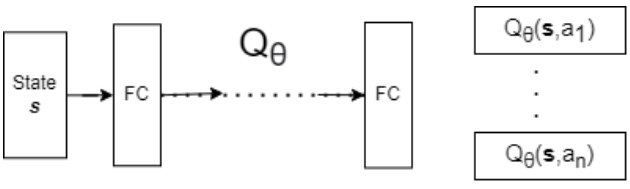

Figure 2. The network structure of DQN

DQN(Deep Q learning)(Algorithm 1) is a reinforcement learning algorithm that combines Q-learning with deep neural networks. It works by using a deep neural network to approximate the Q value. The neural network inputs the current state and outputs the expected Q values for each possible action. The Bellman equation for DQN is:

$$Q_\theta(s,a) = r + \gamma \max_{a'}(Q_\theta(s',a')) \tag{13}$$

From the Bellman equation, we can derive the loss function:

$$\mathcal{L}(\theta) = (Q_\theta(s,a) - (r + \gamma \max_{a'}(Q_\theta(s',a'))))^2 \tag{14}$$

The updating rule is

$$\theta_{i+1} \leftarrow \theta_i + \alpha \nabla_{\theta_i} \mathcal{L}(\theta_i) \tag{15}$$

$\alpha \rightarrow$ learning rate

Algorithm 1 : DQN without Target Network & Experience Replay

Input: env, Q_{θ_i}**,** γ**,** α**, EPISODES**
for episode $\in$ 0...EPISODES **do**
  $s \leftarrow$ env.start()
  $\mathcal{L}(\theta_i) \leftarrow 0$
  while s is not terminal **do**
    output $\leftarrow Q_{\theta_i}(s)$
    $a \leftarrow$ select_action(output)
    $s', r \leftarrow$ env.step(s,a)
    target $\leftarrow r + \gamma \max_{a' \in \mathcal{A}} Q_{\theta_i}(s')$
    $\mathcal{L}(\theta_i) \leftarrow \mathcal{L}(\theta_i) +$ (output$-$ target)^2
  end while
  $\theta_{i+1} \leftarrow \theta_i + \alpha \nabla_{\theta_i} \mathcal{L}(\theta_i)$
end for

3.1. Target Network

One of the problems of the above scheme is that we don't have a real target for our network (the target of the network is computed using the same network). Because of this, every time we adapt the network weights, the target will also change. This will lead to strange behaviour in training by oscillating a lot.
To overcome this behaviour, we introduce the concept of the target network. The target network is a copy of the main network used to calculate the target State-Action values during training. The target network is updated periodically with the weights from the main network. We define:

$$\hat{Q}_{\hat{\theta}} \rightarrow \text{ the target network} \tag{16}$$

The target network $\hat{Q}_{\hat{\theta}}$ has the same structure as the main network Q_θ, but the weights $\hat{\theta}$ remain unchanged for several steps, and it will be changed to the same values of the weights from the main network θ. So, every k-steps $\hat{\theta} \leftarrow \theta$. Now, the Bellman equation becomes:

$$Q_\theta(s,a) = r + \gamma \max_{a'}(\hat{Q}_{\hat{\theta}}(s',a')) \tag{17}$$

This trick reduces the oscillation of the weights over time, stabilising the training process.

3.2. Experience Replay

Another problem derived from the algorithm 1 is that the loss is computed as the sum of losses on each step from an entire episode. Because of this, there is a high correlation between consecutive State-Action pairs, which will reduce the ability of the network to generalise. To handle this problem, we add to the algorithm a replay buffer which stores past observations, and to train the network, every several steps, we will randomly sample a set of observations from the replay buffer and use it to train the network.
So, using experience replay, we reduce the correlation between consecutive states, increasing the ability of the network to learn.

3.3. Double-DQN

Double DQN was introduced in (Van Hasselt et al., 2016) being a method that handles the problem of overestimating the State-Action Q-Value. This problem states that there is no guarantee that the action(a') with the highest value predicted by the target network for the following state(s') is the best action to take. This behaviour is more present in the early training stages when the target network's accuracy is very small (it depends on the number of similar states the network was trained on). Consequently, at the beginning of the training, we don't have enough information about the best action to take. Therefore, taking the maximum ($\hat{Q}_{\hat{\theta}}(s',a')$) value (which is very noisy) as the best action to take can lead to false positives. The learning will be complicated if non-optimal actions are regularly given a higher Q value than the optimal best action. To handle this problem, we will use our model to compute the best action in s', and the value of the target network for that specific action will be used. The Bellman equation becomes:

$$Q_\theta(s,a) = r + \gamma \hat{Q}_{\hat{\theta}}(s', argmax_{a'} Q_\theta(s',a')) \tag{18}$$

3.4. Dueling-DQN

Another improvement to the DQN introduced in (Wang et al., 2016), decomposes the State-Action function ($Q(s,a)$) in 2 functions:

- The State value function $V(s)$
- The advantage of each action in that state $A(s,a)$

$$Q(s,a) = V(s) + A(s,a) \tag{19}$$

This can be achieved by separating the network flows into two streams $V_\theta(s)$ and $A_\theta(s,a)$ (Figure 3).

In the end, we will add an aggregation layer that will

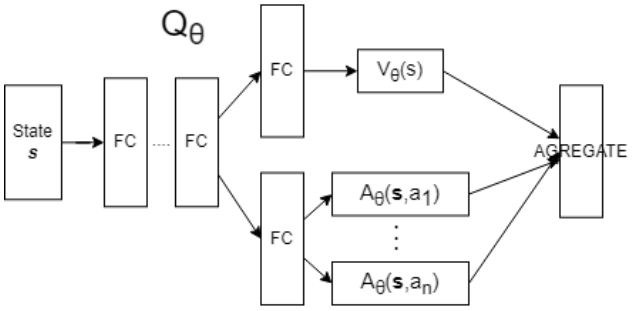

Figure 3. The network structure of Dueling DQN

compute the final versions of $Q_\theta(s,a)$. Because there are infinite ways in which we can separate $Q_\theta(s,a)$, additional restrictions have to be put on the values of $A_\theta(s,a)$, so we forced that the mean value of the advantage to be 0 by:

$$Q_\theta(s,a) = V_\theta(s) + (A_\theta(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a' \in \mathcal{A}} A_\theta(s,a'))$$

The idea behind this network architecture is that in some states, we are not interested in estimating the exact value of the State-Action function $Q(s,a)$ because those states are not critical, and any action is good enough to use. In our case when the pole is close to vertical any action is valid. So we are only interested in estimating as well as possible the State value function ($V(s)$) for any states and for critical states where there is only a valid action to take, the advantage will play a major role in selecting the action.

3.5. Action Selection

- The ϵ-greedy policy:

$$\pi(a|s) = \begin{cases} 1 - \epsilon * \frac{|\mathcal{A}|-1}{|\mathcal{A}|}. \text{ if } a = \text{argmax}_{a' \in \mathcal{A}} Q_\theta(s,a') \\ \frac{\epsilon}{|\mathcal{A}|}, \text{ otherwise} \end{cases} \tag{20}$$

  Where, $\epsilon \in$ [0,1]. This policy gives a high probability of choosing the best possible action based o current tabular values of $\hat{Q}(s,a)$. And if the action is not the best possible one, it is chosen uniformly from the rest. For $\epsilon = 0$, we have a greedy policy (always choosing the best possible action), while for $\epsilon = 1$, we have a discrete uniform distribution over all actions.

- The Boltzmann policy:

$$\pi(a|s) = \frac{e^{Q_\theta(s,a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{Q_\theta(s,a')/\tau}} \tag{21}$$

  where, $\tau \in (0, +\infty)$. This policy computes the probability of choosing an action based on the estimated State-Action value $Q_\theta(s,a)$. For $\tau \to \infty$, we will have a discrete uniform over the actions ($\frac{1}{\tau} \to 0, \pi(a|s) \to \frac{1}{\sum_{a' \in \mathcal{A}} 1} = \frac{1}{|\mathcal{A}|}$).For $\tau \to 0$, we will have a discrete uniform over the actions ($\frac{1}{\tau} \to 0, \pi(a|s) \to \frac{1}{\sum_{a' \in \mathcal{A}} 1} = \frac{1}{|\mathcal{A}|}$). While for $\tau \to \infty$, we will have a greedy policy.

- The Noisy policy:

$$\pi(a|s) = argmax_{a'}(Q_\theta(s,a') + \mathcal{N}(0,\eta)) \tag{22}$$

  $\mathcal{N}(0,\eta) \to$ normal distributed variable; $\eta \to$ scale. This policy chooses the next action using some random noise, so for a scale $\eta \to 0$, the policy is equivalent to a greedy policy

- The Count-Based Exploration and Intrinsic Reward:
  In addition to the policies presented above, we introduce the concept of intrinsic reward to increase exploration. Intrinsic rewards represent an additional reward that the agent gets by getting to a state other than the one given by the environment. This type of reward is computed by the agent according to the information accumulated from the environment.

$$r \leftarrow r_e + r_i \tag{23}$$

  $r \to$ the cumulative reward of an action,
  $r_e \to$ reward given by the environment,
  $r_i \to$ intrinsic reward.
  Similar to (Tang et al., 2017), a way of computing intrinsic reward is represented by counting methods. A less observed state should have an intrinsic reward higher than a state that is frequently observed. So we define:

$$r_i(s) \leftarrow \frac{\beta}{\sqrt{N(s)}}$$

  $N(s) \to$ number of times state s was observed
  $\beta \to$ bonus coefficient.
  The problem with the definition above is that we have a very big (even continuous) state space ($\mathbb{S}$), which will lead to the majority of states having a count of 0. To resolve this problem a hashing method will reduce the state space into several buckets representing similar states. So:

$$\phi : \mathbb{S} \to \mathcal{B} \tag{24}$$

  $\mathcal{B} \leftarrow$ a finite set
  $\phi(s) =$ the hashing value
  We used SimHash as our hashing method,

$$\phi : \mathbb{S} \to \{-1,1\}^k, \phi(s) = sign(\langle A, s \rangle) \tag{25}$$

$$r_i(s) \leftarrow \frac{\beta}{\sqrt{N(\phi(s))}} \tag{26}$$

$N(\phi(s)) \rightarrow$ number of times bucket $\phi(s)$ was observed.
$\langle A, s \rangle \rightarrow$ inner product
$A =$ random matrix of size $k \times |\mathbb{S}|$

4. Experiments

4.1. Hyperparameter tuning

Hyperparameter tuning is a critical step in optimizing the performance of a Deep Q-Network(DQN) algorithm. We have conducted experiments by varying the values of these parameters which include learning rate, number of layers in the target network, frequency of updating the target network and experience replay, and the policy for action selection. We trained the algorithm for variations of these values and measured its performance by evaluating the average reward obtained over 2500 episodes for 3 repetitions.

4.1.1. Learning Rate (α)

One of the most important hyperparameters for a DQN algorithm is the learning rate. It determines how much the weights of the neural network are updated in response to the gradient which influences the algorithm's stability and convergence rate. A suitable learning rate will allow the model to converge to an optimal solution efficiently. In our experiments, we explored three different learning rates: [0.1, 0.01, 0.001]. The results of our experiments are presented in the plot 4, which shows the performance of our DQN agent for each learning rate.

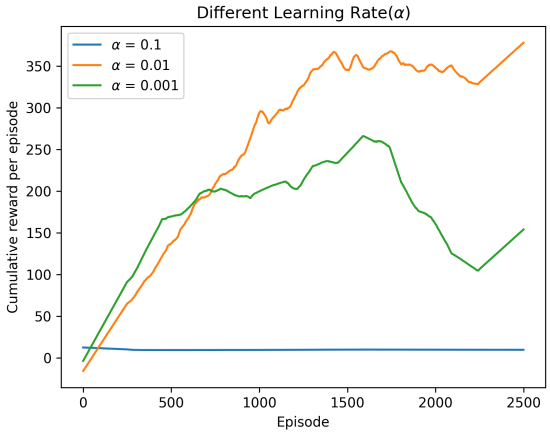

Figure 4. Comparison of rewards we got for 2500 episodes on the DQN using different values of learning rate (α) averaged over three repetitions for this plot

We can see that the graph line for a learning rate of 0.1 is flat, indicating that the agent is not learning anything likely because the learning rate is too high, causing the agent to overshoot the optimal policy and never converge.

The rewards for learning rate 0.01 increase steadily throughout the episodes and fluctuates for an interval of episodes indicating that the agent has converged to a near-optimal policy and then increases again, potentially indicating that the agent found a way to improve its policy. The line for a learning rate of 0.001 increases up to a certain point and then starts to decline which could be because the learning rate is too low, causing the agent to take too long to converge, or the agent is getting stuck in local optima and is unable to find a better policy. Based on the plot 4, it seems that a learning rate of 0.01 performs the best overall. It converges to a near-optimal policy relatively quickly and then continues to improve.

4.1.2. Network Architecture

Network architecture plays a significant role in the performance and learning efficiency of a DQN agent. A suitable network architecture allows the agent to learn meaningful representations of the environment and generalize well to unseen situations. Complex architecture may lead to overfitting on the other hand simple architecture leads to underfitting. so, it is important to tune the network architecture. In our experiments, we explored three different network architectures:

- A single hidden layer with [512] neurons
- Two hidden layers with [512, 256] neurons
- Three hidden layers with [512, 256, 256] neurons

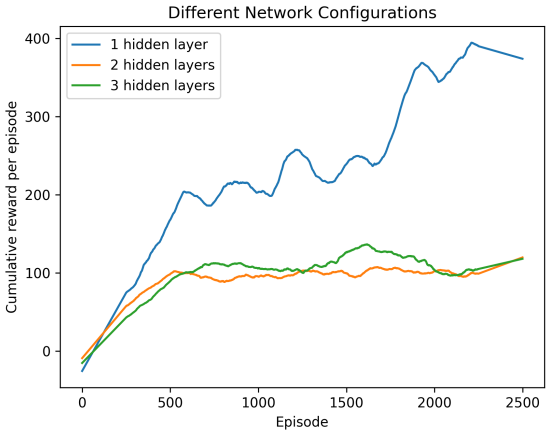

Figure 5. Comparison of rewards we got for 2500 episodes on the DQN using a different number of hidden layers in the target network averaged over three repetitions for this plot

From the graph 5, we can see that the single hidden layer with 512 neurons outperformed the other two architectures.

This suggests that a simple network architecture with 512 neurons is sufficient to learn the underlying structure of the CartPole environment and allowed the agent to learn efficiently and effectively. Also, optimal network architecture can be a problem-specific parameter and we may need to tune further depending on the environment.

4.1.3. Action Selection Policy

Action Selection policy or exploration policy determines how the agent balances between exploration and exploitation while interacting with the environment. A suitable exploration policy allows the agent to explore the state-action space effectively and learn an optimal policy. In our experiments, we explored several exploration policies:

- Annealing ϵ-greedy : As discussed in section 3.5, we used the ϵ-greedy policy(20) with ϵ=0.99 as start value and ϵ=0.01 as end value with annealing up to 50%. The higher start value(0.99) is to make the agent explore more at the initial stages. From the plot 6, we can see that the Annealing ϵ-greedy performed well because of the importance given to the exploration in its initial stages.
- Annealing Softmax: As discussed in section 3.5, we used Boltzmann's policy(21) with a temperature τ=0.99 as start value and τ=0.01 as end value with annealing up to 50%. The higher start value(0.99) is to make the agent explore more at the initial stages. From the plot 6, we can see that the annealing softmax performed the worst compared to the other policies, even though we started with a high τ value to induce exploration. Some problems may be better suited for other policies like ϵ-greedy or Count-based policy than Softmax.
- Noisy Policy: As discussed in section 3.5, we used the noisy policy(22) which introduces stochasticity by adding noise to the action values, encouraging the agent to explore different actions even when it has already learned a potentially optimal policy. The noisy policy allows for continuous exploration, as the agent's behaviour is affected by the noise throughout its interaction with the environment. From the plot 6, we can see that adding noise to the action values and encouraging the agent to explore more made the agent perform well than some other policies.
- Count Based: As discussed in section 3.5, we used the count-based exploration policy to encourage the agent to explore less visited states and actions. In this exploration policy, the agent prioritizes exploring states and actions that have been visited less frequently. From the plot6, we can see that the count-based exploration policy performed better than the other policies, this is because of the exploration bonus we give to each action the agent takes.

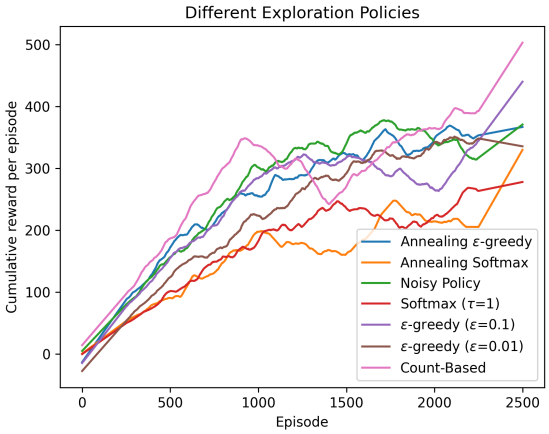

Figure 6. Comparison of rewards we got for 2500 episodes on the DQN using different types of action selection policies averaged over three repetitions for this plot

As seen from the plot 6, we can see that both the Annealing ϵ-greedy and the Count-Based exploration policies performed better than the other policies. Annealing ϵ-greedy allowed the agent to start with a high exploration rate, which decreases over time, encouraging the agent to shift from exploration to exploitation as it learns more about the environment. On the other hand, Count-based policy encourages exploration of less frequently visited states and actions, which can help the agent discover novel solutions and learn an optimal policy. These features made both these policies get better rewards than the other policies. We also did the experimentation with different values of ϵ like 0.1 and 0.01 without annealing and softmax with τ = 1 without annealing they didn't perform well.

4.1.4. Frequency of Updating the Target Network and the Experience Replay

The frequency of updating the target network and experience replay is an important hyperparameter in the DQN algorithm that can significantly affect its performance. When these components are updated too frequently, the computational cost and memory usage increase, while updating them infrequently can lead to instability and slower convergence.

To experiment with the number of steps to update the target network and experience replay we used [100, 1000, 5000] and [3, 5, 10] respectively as step values. The results of our experiments are presented in the plots below, which show the performance of our DQN agent for each of these three values. From the plot 7, we can see that the line repre-

senting 100 steps shows updating the target network more frequently, resulting in faster learning and achieving the maximum reward in a shorter number of episodes. However, the decline in performance after that suggests that the network overfits the training data. The line representing 1000 steps shows that updating the target network less frequently achieves steady improvement in reward throughout the episodes. The line representing 5000 steps shows that updating the target network rarely results in slower learning and lower overall performance. The network takes a long time to start learning and gets a lower reward than the other two parameters.

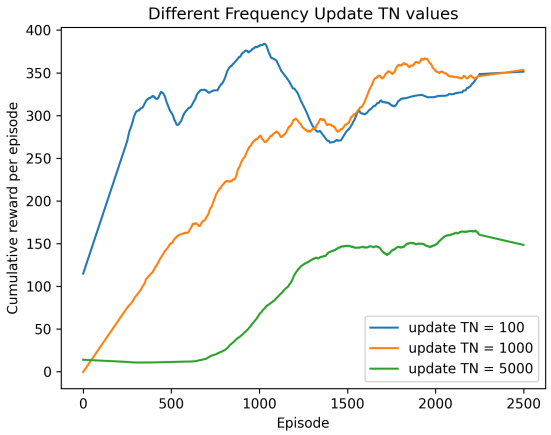

Figure 7. Comparison of rewards we got for 2500 episodes on the DQN by varying the number of steps taken to update the Target Network averaged over three repetitions for this plot

In conclusion, it seems that a frequency update value of 100 might be preferred to achieve a high reward value as quickly as possible. However, if the goal is to achieve a more stable and consistent reward value, then a frequency update of 1000 might be preferred. From the plot 8, the line representing 3 step learning follows the same trend as the 100 steps target network update where the agent reaches max reward fastest and then steadily declines which may be due to overfitting. On the other hand, the lines representing 5 and 10 steps both increase their rewards without any significant decline throughout episodes, indicating that they are learning more stably. However, the line representing 10 steps slightly declines at the end, suggesting that the larger number of steps might have made the network more prone to noise and instability.

Overall, the results from the plot 8 suggest that the optimal number of steps for updating the experience replay lies between 5 and 10. While 3 steps may lead to faster learning, it appears to be too aggressive and prone to overfitting, and 10 steps might be too conservative and prone to noise. A

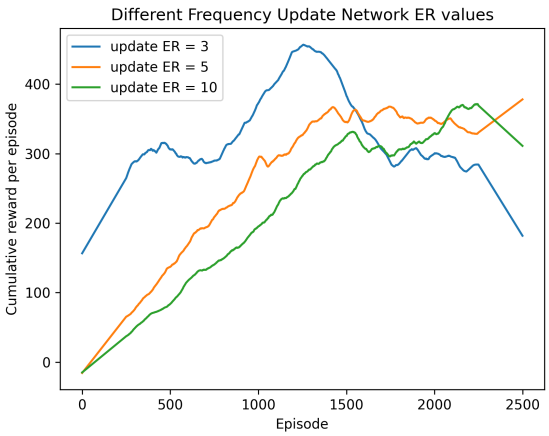

Figure 8. Comparison of rewards we got for 2500 episodes on the DQN by varying the number of steps taken to update the Experience Replay averaged over three repetitions for this plot

value of 5 steps seems to offer a good balance between stability and learning speed.

4.2. Ablation Study on DQN

To evaluate the contribution of different components in the algorithm, we conducted an ablation study on our DQN model, focusing on the Target Network (TN) and Experience Buffer (ER) components by removing one component at a time and comparing its performance with the baseline model. The configurations we used for the ablation study are listed below,

- **DQN**: The baseline version with both TN & ER
- **DQN-TN**: DQN model without Target Network.
- **DQN-ER**: DQN model without Experience Buffer.
- **DQN-ER-TN**: DQN model without both TN & ER.

The results of our experiments are presented below in plot 9, which shows the performance of our DQN model with other configurations mentioned above, From the plot 9, we can see that the baseline DQN model performed the best compared to the other configurations. This is expected because a DQN model with both the TN and ER is known to perform well in many environments, as these components contribute to the stability and efficiency of the model's learning process. DQN without TN performed poorly because removing the target network removes the stability of the model's learning process. As discussed earlier, without the target network, the same network is used for both action-value estimations and updating the target values, which can

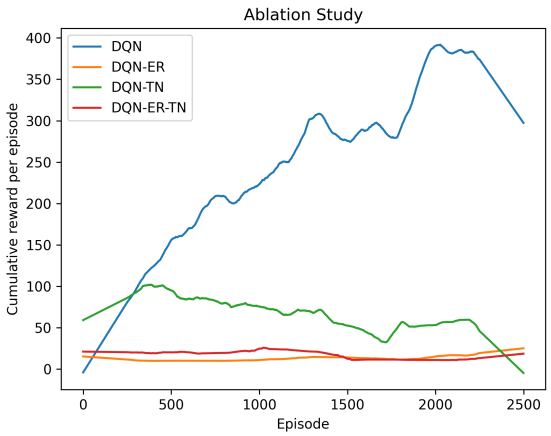

Figure 9. Ablation Study: Comparison of rewards we got for 2500 episodes on the models DQN Vs. DQN-ER Vs. DQN-TN Vs. DQN-ER-TN averaged over five repetitions for this plot

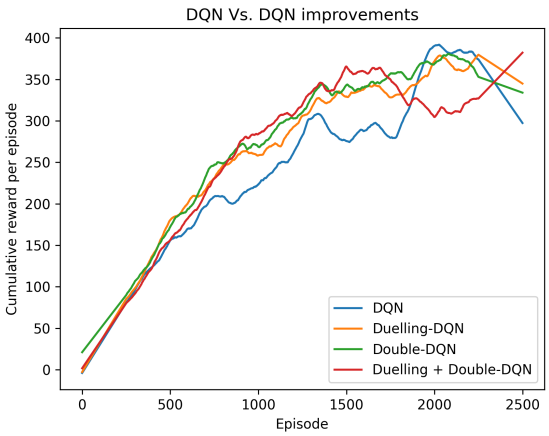

Figure 10. Comparison of rewards we got for 2500 episodes on our DQN with Duelling-DQN, Double-DQN, and combination of both Duelling and Double DQN averaged over 5 repetitions

lead to instabilities and deviation in the learning process. DQN without ER performed worst because removing the experience buffer makes the model train on single experiences they encounter, which leads to inefficient learning and poor performance. A DQN model without both the TN and ER's poor performance is expected. Because the model will lose both stability and the efficiency provided by the target network and the experience buffer. These results highlight the importance of the target network and experience replay components in the DQN algorithm

4.3. DQN vs Other Variants of DQN

We also did experimentation to show the performance of our DQN model with improved DQN algorithms on the same CartPole environment and measured its performance by evaluating the average reward obtained over 2500 episodes for 5 repetitions. We used the below variants of the DQN,

- Duelling DQN
- Double DQN
- Duelling DQN + Double DQN

From the plot 10, we can see that all the variants of DQN performed well in the CartPole environment. The 3 improved versions of the DQN learnt faster than the DQN baseline version and continued to improve. The target network and experience replay buffer helped the DQN model to get better performance. The Double DQN handled the overestimation bias on the state-action values leading to better performance. The duelling network architecture in the Duelling DQN model provided a better representation of the state-action values, leading to better performance. The combination of both Duelling DQN and Double DQN learnt faster in the initial stages. However there is a slight dip at the end, then it recovered and continued to learn. These results show the potential of the improved DQN algorithms and their ability to learn quickly and effectively.

5. Discussion and future work

5.1. Conclusion

In conclusion, the Deep Q-Learning algorithm has been successfully applied in a cart-pole environment, the implementation results demonstrate that the agent was able to learn an optimal policy that balances the cart-pole system, leading to significantly higher rewards. Our experimentation on the hyperparameters, ablation study and the comparison of different DQN variants shows the importance of hyperparameters tuning and the components like target network and experience buffer and the potential of combining different DQN variants for better performance. Our study demonstrates the possibility of the DQN algorithm to solve various challenging problems in robotics and gaming. Even though our experimentation is based on a particular environment, the understanding acquired can be applied to other environments. Additionally, exploring other advanced variants like distributed DQN, and prioritized experience replay may lead to even better performance and wider application in solving real-world problems making it an exciting area for future research.

References

Plaat, A. Deep reinforcement learning. *CoRR*, abs/2201.02135, 2022. URL https://arxiv.org/abs/2201.02135.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Xi Chen, O., Duan, Y., Schulman, J., DeTurck, F., and Abbeel, P. # exploration: A study of count-based exploration for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pp. 1995–2003. PMLR, 2016.