Assignment 1: Q-Learning - Tabular and Deep

Ekansh Khanulia [*1]

Abstract

Reinforcement Learning (RL) has for long time relied on tabular methods to estimate value functions. However, in environment with continuous large spaces these methods become infeasible because of memory constraints. To resolve this, DQNs (Deep Q-Networks) which utilizes neural networks as function approximators, enables efficient learning in high-dimensional spaces. In this assignment we will explore the realm of DQN by implementation of a DQN-based agent in the CartPole-v1 environment, evaluating its stability and performance .Moreover We conduct an ablation study on hyper-parameters like learning rate,exploration strategy, network size, to underline their impact. Furthermore, we enhance the baseline naive-DQN with TN (Target Network) and a (Experience Replay) ER comparing their results across different configurations.

1. Introduction

Reinforcement Learning (RL) helps agents to learn optimum policies through interactions with an environment, aiming to get a maximum cumulative rewards. In traditional tabular Q-learning, as the name suggest an explicit table is uses to store Q-values for each state-action pair. However, with continuous large state spaces , this approach is tedious to implement. So Function approximation using neural networks, provides us with a scalable solution, making a way for Deep Q-Networks (DQNs).

This assignment focuses on implementing Q-learning with function approximation in the CartPole-v1 environment. The main objective is to evaluate and compare whether a deep learning-based agent can successfully balance the pole and achieve optimal performance. We start by coding a naive DQN agent,which is followed by an ablation study to analyze the effects of hyper-parameters in learning the environment. Moreover to improve our result we make use of two strategies , ER (Experience Replay) and a TN(Target Network) —to stabilize learning and mitigate divergence.At the final step the results are compared to differentiate the learning performance across four configurations: (1) Naive DQN,(2) DQN with Experience Replay (3) DQN with Target Network, and (4) DQN with both enhancements. y running every model 4 times the best learning curve for that model is computed.

[*]Equal contribution [1]Faculty of Science, Leiden University, Leiden , Netherlands. Correspondence to: Ekansh Khanulia <jatinkhanulia@gmail.com>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).

2. Theory

2.1. Loss function in Q-Learning

Loss function in Q-learning is based on temporal difference ,a learning which refines an estimated action-value function Q(s,a) using experience. Q-learning uses Bellman equation to estimate the value of both future and immediate rewards, derived from the state-action pair. This process, known as the update rule, is a practical implementation of the Bellman equation.

$$\text{Q(s, a)} \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

When Q-learning is combined with a neural network, a loss function is introduced to measure the difference between the predicted Q-value and the actual target determined by the Bellman equation. The most commonly used loss function is Mean Squared Error (MSE). To address challenges like the moving target problem, additional techniques such as target networks and experience replay are incorporated.

$$\text{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{\text{target}}) - Q(s,a;\theta)\right)^2\right]$$

2.2. Why we cannot use update rules as in tabular RL

There are may challenges regarding using update rules in deep RL which arises from function approximation such as using neural network to represent value function. Methods like Q learning or SARSA uses table to capture each value for each action in an independent fashion, however storing these values become tedious in case of continuous large state spaces,phenomenon known as State space explosion.In Tabular Rl we only update a specific Q-value of single state action pair, which doesn't effect other values in the table , but in case of neural network where instead

of table we have to deal with weights ,updating weights for one state action pair also effect changes prediction for others,which prevent convergence instability.In tabular method when a unseen state is encountered there is nothing much they can do for that.Deep Learning model on other hand uses a parametrized function that help to generalize from seen state to unseen state.In Deep RL as the target value keep on changing as learning progresses, we cannot update values directly from the table as we do in case of Tabular RL. This is called moving target problem ,causing convergence issues and making updates unstable.In Tabular method, making an update is straightforward as each state-action-reward next state is treated individually,but in Deep RL , experience are collected in a sequential manner, which leads to highly correlated updates.On the other hand Tabular RL gaurentees convergence to optimal Q-value ,but using function approximation in deep RL causes unstable updates ,ultimately resulting in divergence

2.3. Pseudo code

1. Initializing the Experience Replay Buffer $\mathcal{D}$an empty space that will be used to store past transcation.
2. Initializing Q-Network $Q(s, a; \theta)$ with random weights θ.
3. Initializing Target Network $Q'(s, a; \theta')$ and set $\theta' \leftarrow \theta$.
4. For each episode:
   (a) Reset the environment and obtain the initial state s_0.
5. For each time step t in the episode:
   (a) Select action a_t using ϵ-greedy approach from $Q(s_t, \cdot; \theta)$.
   (b) Execute action a_t in the environment, observe reward r_t and next state s_{t+1}.
   (c) Store experience (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$.
6. Sample a mini-batch of past experience (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$.
7. Compute the target value y_j:
   (a) If s_{j+1} is terminal, set $y_j = r_j$.
   (b) else, use target network:
   $$y_j = r_j + \gamma \max_{a'} Q'(s_{j+1}, a'; \theta')$$
8. To minimize loss perform gradient descent :
   $$L = (y_j - Q(s_j, a_j; \theta))^2$$
   and update the Q-network weights θ.
9. After every C step, update the target network: $\theta' \leftarrow \theta$.
10. Repeat until training completes.

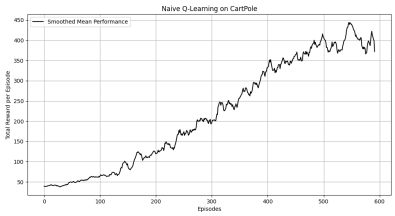

Figure 1. shows the smoothed average reward per episode increasing over time as the Naive Q-Learning agent learns to balance the CartPole

3. Experiments

3.1. Environment Setup

Gymnasium - Provides pre-built Reinforcement Learning environments, e.g., cartpole-v1, which is used in this assignment, enabling interaction via reset(), step(), and render(). In this experiment, we use gymnasium.vector to parallelize the execution for faster training of the models. Torch (PyTorch) is used for building and training deep Q networks, with torch.optim for optimization and torch.nn for neural networks. Matplotlib.pyplot is used for plotting learning curves, and gym.make() creates an RL environment, interacts using step() and resets with reset(), and returns the next state, reward, and done status.

3.2. Implementation of Naive Q-Learning

This technique involves training a neural network (q-network) to approximate the optimal Q-value function, which estimates the expected future reward for taking a specific action in a given state. Naive Q-learning is a baseline setup using a Neural Network for cartpole-v1.Neural network used in this task is a fully connected feedforward network consisting of layers of 256, 256, and 128, each followed by ReLU activation for non-linearity.The agent interacts with the CartPole-v1 environment using an epsilon-greedy policy: it either selects a random steps to explore or chooses the action with the highest Q-value predicted by the network to use its current knowledge. The update q-network estimates the target value using the Bellman equation, and uses Mean Squared Error (MSE) as the loss function to update parameters via the Adam optimizer and backpropagation. The agent trains in a vectorized environment with the network using Gradient descent. The agent's performance is evaluated by tracking the total reward per episode, which is then smoothed and plotted to visualize the learning progress across episodes and runs. The exploration rate epsilon is decayed over time to encourage more exploitation as the agent learns..

.

3.3. Ablation Study on Hyper-parameters

The study conducted aimed to analyze the impact that some hyper-parameters create on the performance of the Q-agent. The goal of the study is to compute what values of those hyper-parameters can give us optimal learning and stability.

The four hyper-parameters used in this study are the learning rate, update-to-data ratio, exploration factor (epsilon decay), and network size. We first set default values for the other 3 parameters while varying one hyperparameter at small, medium, and high values. The training setup ensured that each configuration of a hyperparameter was run for four repetitions, followed by applying an exponential moving average to smooth the results.

Three different values were tested for each hyper-parameter, and the results were plotted in a graph. During this process, the remaining hyper-parameters were set to their baseline values taken from naive DQN.

Once the best learning rate ,which was the first hyperparameter have ablation study , was identified, we proceeded to determine the optimal update-to-data ratio, using the best learning rate while keeping the other hyper-parameters at their baseline values. Each time we found the best value for a hyperparameter, we updated its value accordingly before moving on to optimize the next one.

3.4. Experience Replay(ER)

In this model, we used an Experience Replay (ER) setup to enhance sample efficiency by storing past experiences in a buffer size of 50,000. It helps the agent decide future actions and reduces the correlation between consecutive updates. Each experience tuple is appended to a deque, removing older experiences automatically when buffer reaches its maximum capacity.The experiences gathered from the environment (state, action, reward, next state, done) are stored in an experience replay buffer (ReplayBuffer). During the training update, a batch of experiences is randomly sampled from this buffer. Mini-batches are sampled from the buffer and converted into NumPy arrays for computational efficiency. Without ER, training on sequential data can lead to catastrophic forgetting and unstable updates to the value function.

3.5. Target Network(TN)

If we update Q(s,a) using the same network that predicts Q(s,a), it results in bootstrapping from itself, leading instable results. Since the network being continuously updating its values, the target keeps shifting.To address this issue, we use a target network in this model, which is a copy of the Q-network. While the Q-network is updated constantly, the target network is updated only every 1,000 steps. This helps

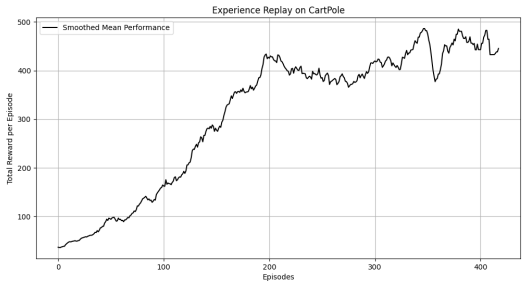

Figure 2. shows smoothed average reward per episode increasing over time as the agent using Experience Replay learns to balance the CartPole, reaching higher rewards than Naive Q-Learning

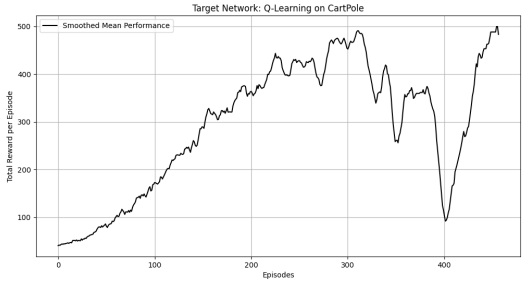

Figure 3. shows agent using a Target Network generally increasing smoothed average reward per episode, reaching high performance but showing significant instability with sharp drops

ensure that the target Q-values remain stable in the process. $\mathrm{Q}_{\text{target}}(s,a) = r + \gamma \max_{a'} Q_{\text{target}}(s',a')$

3.6. ER and TN

Although using a target network helps prevent the moving target problem and ensures that Q-values do not update too frequently, it can still lead to overfitting to recent experiences when used on consecutive samples, as each experience is used only once. On the other hand, Experience Replay (ER) allows the agent to leverage past experiences for decision-making but remains susceptible to the moving target problem.

By combining ER and the target network (TN), their weaknesses complement each other, maximizing both stability and efficiency. In the ER+TN model, ER is implemented using a replay buffer class that stores past transitions in a deque with a fixed size of 50,000. Random sampling of mini-batches helps prevent overfitting to recent experiences. The target network is initialized as a copy of the Q-network at the start of training and is implemented as a separate instance of the Q-network, which is updated periodically to

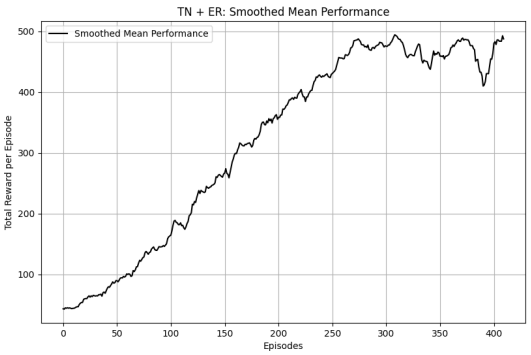

Figure 4. show agent which combines a Target Network and Experience Replay leads to stable and high-performing learning , achieving consistently high average rewards with minimal instability

match the main Q-network.

Instead of adjusting Q-values based on a constantly shifting target, the target network provides a delayed and more stable estimate of future rewards, mitigating the moving target problem. The target network is updated every 1,000 steps to maintain stability during training. While ER enables the agent to learn from a diverse set of past experiences, TN ensures that learning updates do not introduce instability due to rapidly changing Q-values.

4. Discussion

This section discusses the results, analyzing the performance of different approaches (Naive, ER, TN, and DQN). It also highlights the strengths and weaknesses observed during training.

4.1. Result: Naive Q-learning

The key observation in the plot is that the model initially performs poorly, achieving lower rewards. This is expected because the Q-network is untrained, and exploration dominates the early stages. However, between episodes 150 and 400, there is a noticeable improvement, with rewards showing a slight upward trend.

Toward the end, a performance drop is observed, likely due to an imbalance in exploration and exploitation, leading to overfitting to suboptimal Q-value updates. Despite this, the model achieves a decent performance, surpassing a reward of 300, indicating that the agent can balance the pole for a significant duration. The agent demonstrates an overall improvement trend with steadily increasing rewards. However, it exhibits slow convergence, requiring a large number of episodes to reach its full potential. Additionally, performance fluctuations, particularly near the end, suggest that the learned policy lacks robustness and may degrade over time. Without a replay buffer, relying solely on the -greedy strategy can lead to poor generalization.

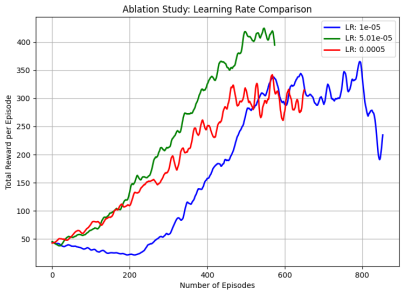

Figure 5. shows the green line (LR: 0.0000501) appears to represent the best learning rate among the three tested

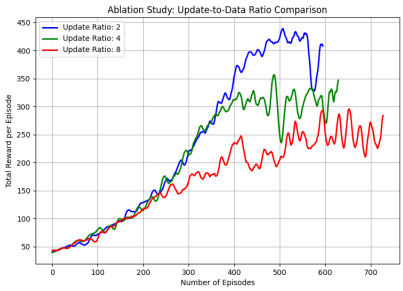

Figure 6. Update to data ratio 2 performs the best

4.2. Result: Ablation study

Learning rate (figure 5): The graph compares different learning rates 0.00001, 0.0000501, 0.0005. The best out of them all comes to be 0.0000501 (green line). We observed from the graph that the blue line has too small learning rate, causing slow learning, while too high a learning rate (red) is the cause of instability. So 0.0000501 allows for a fast learning rate and stable convergence.

Update to data ratio(figure 6): The graph compares update to data ratios of 2, 4, and 8. The best one comes out to be 2 (blue line). The graph shows that the higher ratio of 8 causes instability due to excess updates, while 2 balances performance and stability.

Network size(figure 7): The graph compares three network sizes [128, 64], [256, 256, 128], and [512, 256, 256, 128]. The best comes out to be [256, 256, 128]. The graph illustrates that an overly large network (red line) over-fits and causes fluctuations in rewards, while too small a network (blue) struggles to learn.

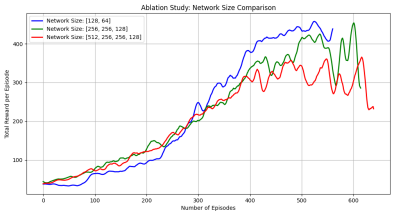

Figure 7. The green line shows ([256, 256, 128]) is performing the best

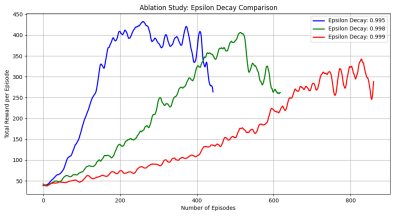

Figure 8. blue line (Epsilon Decay: 0.995) appears to be the best performing

Epsilon decay(figure 8): We compare three values for epsilon decay (0.995, 0.998, and 0.999). And the best comes out to be 0.995 (blue). The higher decay of 0.999 (red line) leads to slower learning as the agent explores too long, but 0.995 decay learns faster, achieving higher reward earlier, and stabilizing performance.

4.3. Result: Experience replay(ER)

The learning curve reaches higher rewards and smoother than naive Q learning. The performance stabilizes above 400 reward per episode with occasional dip in the graph. Because of the usage of past experience, it allows the agent to sample efficiently. And the agent manages to get 450 reward per episode. But we can see the graph represents some instability and does not fully prevent catastrophic forgetting or sudden q value changes.

4.4. Result: Target Network

It follows a similar trend like ER, but manages to reach peak performance (500 reward per episode), but it is very much prone to sharp drops. It helps stabilize training by preventing sudden q value updates. But sharp and sudden dips in the performance indicate the model struggles to adapt.

4.5. Result: ER+Tn

By far the best performing model with a steady learning curve and maintaining 500 reward per episode, and also showing minimum drops. ER prevents correlated updates while TN stabilizes the training process.

5. Conclusion

5.1. Summary

The primary goal of the experiment was to analyze different reinforcement techniques for training Q learning agent on the cart pole environment. We compared naive Q learning baseline, experience replay (ER), target network (TN), and a combination of ER+TN. We also conducted ablation study on the naive baseline and got the best hyper-parameters and used them in the other three models, ensuring an efficient balance between exploration and exploitation.

In the results, we find that Naive q learning learned slowly and plateaued early, and it struggled for stability. ER improved learning efficiency by using past experience, leading to efficient performance, but still could not achieve peak performance. Target network, although achieving the highest reward compared to the previous two, was too unstable to maintain it. But lastly, ER+TN provided the best performance, ensuring stability and a higher final reward by addressing the shortcomings of both TN and ER. In the ablation study, the best values came out to be: Learning Rate: 0.0000501, Update-to-data Ratio: 2, network size of [256, 256, 128], and epsilon decay of 0.995.

5.2. Future work and improvement

There are several ways to achieve far better results. Instead of sampling past experience in a uniform fashion, we can prioritize important actions/transitions which could help improve results.Experiment used a fixed decay rate; instead of that, we could try using adaptive epsilon decay which will focus on long-term exploration rather than short. We could also use grid search or Bayesian optimization to automate hyperparameter tuning; future refining the model for even better results.

.