Reinforcement Learning Homework (2025)

Fizza Usman(s4402375) [*1]

Abstract

This paper dives into how DQNs with ER and TN are implemented and analyzed in a Cartpole Environment. Impact of multiple components and hyperparameter configurations are investigated and evaluated throughout the paper.The goal is to find a graph that performs better or in competiton with the baseline data. The results indicate that while TNs contribute to more stable learnings achieving optimal results depends on the specific environment at hand and the exploration-exploitation trade-off. The analysis highlights the need for more comprehensive experimentation, such as combining multiple DQN improvements as done in Rainbow studies.

1. Function Approximation - Theory

1.1. Motivation and Tabular RL

In traditional tabular methods in RL we store each state-action pair and its corresponding Q value.

An update rule is needed that helps an agent to improve its understanding of the optimal policy and behavior over time.

The update rule in Tabular Q-learning uses the Bellman Equation. The Bellman Equation is:

$Q_{\text{opt}}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_{\text{opt}}(s',a')\right]$

Here Q_{opt} denotes the optimal Q-value for taking action a in state s. It is optimal in the sense that it aims to get the highest cumulative rewards. r is the immediate reward that follows when we take an action a in state s. γ is the discount factor that tells how much of future rewards are taken into account as opposed to immediate rewards. $\mathbb{E}$ is the expectation operator that is responsible to average over all possible next states in stochastic environments. The optimality comes from the maximization of Q over all possible actions and taking an average over all expected states.

The Q-update rule is given by:

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

However, this becomes quickly infeasible for large action or state spaces. To deal with these situations, we deal with function approximation.

We are essentially trying to learn a formula that can estimate the Q-value for any state-action pair.The function also has some parameters which tweak how the function works or behaves.

1.2. Why can't we use the same update rule in DQN

In DQN we are dealing with large state spaces which can be continuous in nature.

For example, in environments where state spaces are represented by images it is not possible to store Q-values explicitly for each state.

1.3. Loss Function in DQN

DQN makes use of neural networks to approximate the Q-value function.We cannot use the tabular update directly as they do not incorporate generalization over state space. Instead we play around with and estimate the parameters of the network.

The idea is to minimize the difference between the predicted Q values from the neural network and the true Q values.

This is the mean squared error between the predictions and the true values given by:

$L(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q_{\theta_{\text{T}}}(s',a') - Q_{\theta_{\text{P}}}(s,a)\right)^2\right]$

2. Function Approximation - Experimentation

The CartPole excel document has the following column names or parameters : 'Episode_Return', 'Episode_Return_Smooth', 'env_step'. The episode return parameter tells the total reward an agent received during each episode. The episode return smooth parameter represents the average return over last few episodes. The env step parameter tells us how many steps are taken by the agent up to that point in training.

In Cartpole environments the state usually consists of four dimensions including the cart position, cart velocity, pole angle and pole velocity.

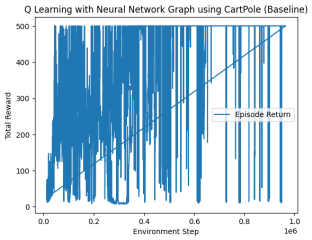
I had to look at gymanisum documentation to see how we can specify the state space for a particular environment. The attribute observation_space defines the upper bound, lower bound, number of state dimensions and the storage data type. Furthermore, I also did the same in order to see how action space can be defined. The attribtue action_space from the gymanisum environment was utilized here.

2.1. Baseline Performance

The baseline data is given in a file and the number of environment steps is around 10^6. As this was done over 5 repetitions, I interpreted this as 5 repetitions * 100 episodes * 2000 target steps.

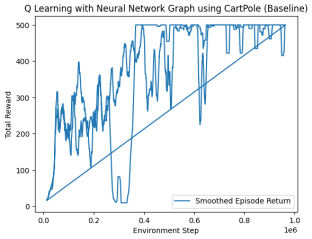
I tried to graph the given performance using averaging, smoothing and convolving.

2.1.1. Performance Graph for Baseline Data using Totaling



This graph does not tell a reader much as it is too noisy and lacks a clear trend. A person would expect the total reward to increase as environment steps increase or time progresses.

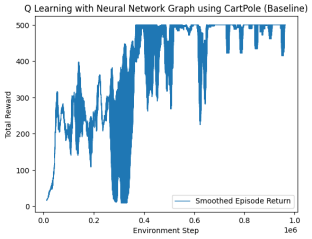
2.1.2. Performance Graph for Baseline Data using Smoothing



As you can see above initially the reward is very low and the agent is still learning.The reward increases with time. The fluctuations help depict the curiosity and the exploration nature of the agent.The final plateau means that the agent has reached optimal policy.

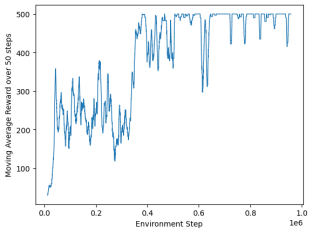
It seems like there are 2 overlapping graphs. This is because of the data becomes unshuffled that happens in due to pandas library. I fixed this by sorting the data by the env_step and then plotting the graph.

2.1.3. Performance Graph for Baseline Data using Sorting and Smoothing



This seems more reflective of what happens in an environment. From 0 to 200k steps, the agent's performance is very volatile and unstable.The rewards stabilise around 200k to 500k steps. The final phase reaches the maximum reward multiple times but still experiences some dips due to the curious nature of the agent.

2.1.4. Performance Graph for Baseline Data using Sorting, Smoothing and Convolving



Convolving involves smoothing data by averaging neighbouring points. Since the previous graph was jagged in nature, this graph provides a better understanding of the trend of agent performance over time.

2.2. Implementing an Agent with naive function approximation - without experience replay (ER) and target network (TN)

2.2.1. CartPole Environment

The gym.make command in my code creates a CartPole-v1 environment from gym. Here, an environment is created where an agent must balance a straight pole on a moving or unstable cart.

Since this was my first time working with RL environments, I had to refer to the OpenAI Gym section of the Deep Reinforcement Learning Book by Aske Plaat for guidance.

I used the cartpole version 1 instead of the 0 since the environment ends an episode after 200 time steps in version

0. Even though the episode length is not fixed in version 1, my goal was to try with a combination of different number of episodes and maximum steps that can be taken in an episode.

This also allowed me to make a priority system where I updated Q-values based on episodes with a bad number of steps taken or where episodes completed without target ever being reached.

2.2.2. Function to Build Neural Network

I made a function called 'build_neural_model_Q' which took the state of the environment as input and outputted a Q-value for each possible action that the agent can take.

The neural network was built using Keras, leveraging the information learned in the 'Introduction to Deep Learning' course.

I created a sequential model in which the layers are stacked and all layers except for input and output layers were relu.

The linear activation function seemed relevant here in the output since this is a regression problem where a real- valued output is needed.

I compiled the model for training with the specific configurations for optimization.

I tested with specific activation functions, optimizers and learning rates in different blocks. This was only in the ipynb file.

Later, in the ipynb file I have modified this model so that other activation functions, optimizers, and learning rates are used. This was done by specifying each of these hyperparameters as dictionaries.

2.2.3. Function to Build Epsilon Greedy Action Strategy

The inbuilt random.uniform was used to generate a random number between 0 and 1.We defined a random explore_epsilon variable which was pre-defined with a number of our choice. If the program generated number was greater than the explore_epsilon variable we used a exploration strategy where a random action is chosen in the training.

However, if the program generated number was lesser than the explore_epsilon variable we implemented a exploitation strategy where the action which produces the best future rewards was chosen in training.

2.2.4. Function to Implement Bellman Update to Update Q-values in training stage

My bellman update used a priority mechanism to give special attention to certain experiences where agent performs badly to use for learning.

In particular, I used the following cases:
1. Reaching the target takes too many steps.
2. The target never reaches but the steps complete.
3. A suboptimal solution is reached, but the program thinks it is the target.

Here, there is an adaptive learning rate that depends on the current priority case that the model encounters.

If the episode is not terminated or the agent hasn't acheived its goal, the bellman equation given in the motivation section of this paper is used to update the target Q value (right hand side of the equation).

2.2.5. Function to Implement Overall Training

This function represents the entire training loop.It is used to explore the environment, take actions, update the policy and keep a tab of the performances over multiple episodes.

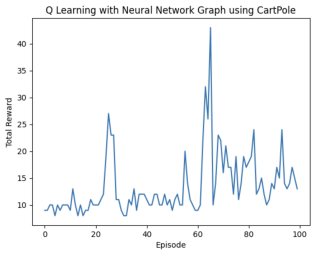
There is an interaction loop that keeps running till the episode is not terminated. Here the agent selects action based on epsilon greedy policy. The agent takes that action and the environment responds by going to the next state. The environment also provides the immediate reward for this action. Along with this reward the environment signals whether the episode is over or has been cut short.

After receiving this feedback the agent revises its knowledge by adjusting q values. This keeps on happening for a specified number of episodes.

2.3. Initial Model Performance in Comparision to Baseline

Here, I implemented a simple model where the number of episodes is 100 and the maximum steps per episode are 200. Furthermore, there was no repetition done and the q values were just observed as this was my first run.

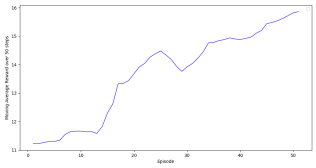
2.3.1. Performance Graph for Naive Function without ER and TN (without smoothing or averaging over runs)



When this was compared to the convoluted graph for the baseline data, the total reward decreased when the episodes

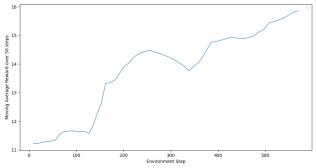
progressed. This was alarming for me and I remembered what happens in the pandas library.

2.3.2. Performance Graph for Naive Function without ER and TN (by smoothing and convolving)



Note that this graph is plotted per episode instead of per step as baseline was done.

In order to show how the graph per step would look like please see below:



These graphs suggest that the agent is improving its behavior in a predictable manner. The lack of dips means that there is not enough randomness or exploration in agent's policy. In the baseline performance there were occasional dips to show the agent's curious nature.The agent is prematurely converging to a suboptimal policy either because of fewer episodes, fewer steps per episode or the exploration strategy.

2.4. Implementing an Agent without target network

The original code was modified to include the experience replay. Here, a buffer is implemented that stores past experiences into memory till a specific time point and then samples from buffer to update the q-values of the model.

I made functions to store and sample episodes in a buffer.

Code snippet

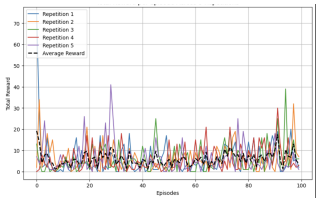
```
def add_episode_to_buffer(state, action,
    reward, next_state, terminated):
    episode = (state, action, reward,
        next_state, terminated)
    episodes_buffer.append(episode)

def sampling_batch():
    return random.sample(episodes_buffer,
        batch_size)

def storage_in_buffer():
    return len(episodes_buffer) >
        batch_size
```

These functions were then utilized in the overall training of the neural network where samples from batches were used to update the q-values using the bellman update training function.

2.4.1. Performance Graph for Naive Function without TN(by averaging and totaling over 5 runs)



Since this is done over multiple repetitions, it is graphed with different colors.When this is compared to the convoluted graph from the baseline data, we can see that none of them perform better and the average reward is generally low. The priority conditions do not accurately capture the dynamics of the environment. Furthermore, the learning rates for the prioritiy cases. In fact high learning rates between 0.1 to 0.8 can cause instability in the learning process.

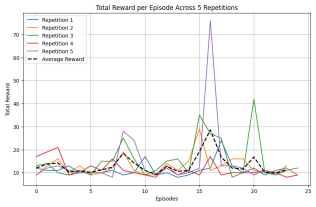
The explore_epsilon parameter is kept at 1.0 and the random number is compared to this, which controls the exploration-exploitation trade-off, may also be contributing to the poor performance.There may be cases where the random number generated by the generator is always greater than the parameter in which case the agent never gets a chance to exploit its learned policy.

2.5. Implementing an Agent with target network - without experience replay

The original code was modified to include target network. Here, a subsidiary network(target network) was created in order to provide the main network with consistent target values. After every Nth episode, the target network is updated to match the weights of the main network.

2.5.1. Performance Graph for Naive Function with TN(by averaging and totaling over 5 runs) - violation

For this graph, I wanted to see what would happen if I violated the principles of Q-learning and did not call the update_target_network function. There is a high variability, inconsistency and low average reward.



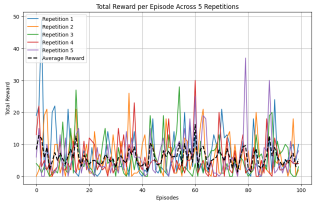
Here, it can be observed that the average performance is still low but peaks at the 16th episode. In repetition 5 as well the performance peaks at episode 16. Additionally episode 3 shows a notable peak at episode 20. Maybe early stopping could have been implemented, or a moving average could have been taken.

Please note however, the performance is suprisingly way better than performing ER. Here the results are distinguishable from each other, and are not consistently moving up and down. The fluctuations show meaningful trends even though high learning rates are taken (a continuation of the issue in the previous section code).Even without a dedicated target network, the neural network itself is performing a form of implicit averaging or regularization.

Please note that the TN normal file crashed and was unable to run. However, the subsidiary or target network was updated after every 10th episode as promised at the beginning of this section.

2.6. Implementing an Agent with target network and experience replay

2.6.1. Performance Graph for Naive Function with TN and ER (by averaging and totaling over 5 runs) - with violation



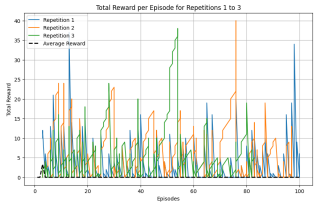
Here the impact of missing call of update_TN_network function is clearly seen. The agent's learning process becomes erratic as unstable targets contribute to the high variability in reward. The lack of stable targets prevents the agent from consistently improving its policy, resulting in limited learning and no clear convergence towards higher rewards.

2.6.2. Performance Graph for Naive Function with TN and ER (by averaging and totaling over 3 runs) - without violation

The graph below shows the performance when both TN (with updation happening every 10th episode) and ER are incorporated in code.

There is inconsistent learning. While some repetitions show spikes in reward, they might be plateauing or even experiencing drops in performance after those peaks. The variations, when averaged, can cancel each other out, leading to a flatter average.

Furthermore, 100 episodes might not be enough for the agent to fully learn and consistently achieve high rewards. The average is only a simple mean. If the initial episodes have very low rewards, they will impact the average, even if later episodes show improvement. The weighting mechanism should be fixed that gives more importance to recent episodes.



When you compare this to the baseline, this graph gives a microscope view of the learning process, showing the details of each episode.

Overall, the baseline is a better graph because rewards increase consistently with episode. However it is better to the graph of ER as the fluctuations are distinguishable and have occasional highs.

3. Pseudo Code for DQN with Experience Replay (ER)

1. Initialize Replay Buffer.
2. Set Batch Size for training.
3. Initialize Q-Network and Target Network.
4. Compile Q-Network with optimizer and learning rate.
5. Start Episode Loop (1 to `num_episodes`).
6. For each episode: Choose action, take action, observe environment, and add experience to buffer.
7. If Replay Buffer has enough data, sample a batch for training.
8. Decay epsilon after each episode to balance exploration and exploitation.
9. Update Target Network with Q-Network parameters every `target_update_freq` episodes.
10. Repeat until desired number of episodes or terminal

state is reached.

Algorithm 1 DQN with Experience Replay (ER)

Initialize Replay Buffer with `max_size`
Set `batch_size`
Initialize `Q_network` and `Target_network`with `state_space_size` and `action_space_size`
Compile Q-network with optimizer and learning rate
for episode = 1 **to** `num_episodes` **do**
  state = Reset environment()
  cumulative_reward = 0, terminated = False, steps_taken = 0
  while not Terminated **do**
    action = `epsilon_greedy_action(state)` ▷ Select an action using epsilon-greedy policy
    next_state, immediate_reward, terminated, info = `cartpole_env.step(action)`
    `add_episode_to_buffer(state, action, immediate_reward, next_state, terminated)`

    if there is enough data in buffer **then**
      batch = `sampling_batch()` ▷ Sample a batch from the buffer

      for each `sample` in the batch **do**
        `Unzip the batch into individual components`
        `q_value_before` = `get_q_value(states, actions)`
        `Perform Bellman update`
        `q_value_after` = `get_q_value(states, actions)`
      end for
    end if
    `epsilon = max(epsilon_min, epsilon * epsilon_decay)` ▷ Decay epsilon after each episode
  end while
  Update `Target_network` with `Q_network` parameters every `target_update_freq` episodes
end for

4. Ablation

Due to the constraints of time and computing power, I tested with 3 configurations of different hyperparameters on the ER model (with number of steps as 2, number of episodes as 50 and the number of target steps as 500). This included changing the optimizer, the loss functions, the buffer size, learning rates and batch sizes. The number of repetitions and target steps were kept constant in order to enable comparision.In further studies I aim to also see how changing the number of layers in the deep network, the activation function and action selection policy change the performance.

The different configurations tested were:

- **Config 1**: Optimizer: Adam (0.001), Replay Buffer: Small (5000), Batch Size: 16, Loss Function: Mean Squared Error
- **Config 2**: Optimizer: RMSprop (0.002), Replay Buffer: Medium (10000), Batch Size: 32, Loss Function: Huber
- **Config 3**: Optimizer: SGD (0.005), Replay Buffer: Large (20000), Batch Size: 64, Loss Function: Mean Absolute Error

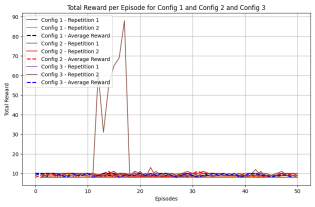
As you can see below, the majority of rewards for all configurations remain low at around 10.

There is one configuration that spikes in rewards between episodes 10 and 20.

The dashed lines are stable and do not show major growth.

On average, none of the configurations achieved significant learning improvements over time.

The spike in Config 3 - Repetition 2 indicates a brief moment where the agent found an optimal strategy, but it did not generalize.



This bad performance can also be explained by the low number of episodes and the high learning rates used in the priority mechanism, which may have led to unstable updates, preventing the agent from converging to an optimal policy.

The total computations done in this section are around 3*2*50*500 =150,000.

5. Analysis

It is challenging to say which model performs the best in the absolute sense.

In my opinion, the TN model performs the best due to distinguishable results and occasional but not volatile high points.The target network contributes to a more stable learning process compared to the other implementations, even if it doesn't consistently reach the highest rewards.

The document clearly shows that TN,ER and TN and ER models have potential but due to inefficient hyperparameters and the 'exploration-exploitation' trade-off the optimal policy isnt met. The fluctuations show that the agent fails to consistenly improve performance.

There is a need to test with more hyperparameters and different types of networks other than deep neural networks like tree based models. Furthermore, is a need to systematically combine methods as done in Rainbow studies.