CS6482 Deep Reinforcement Learning

Assignment 2: Sem2 AY23/24 - DQN Classic Control

Open AI Gym Classic Control Problem: Mountain Car-V0 (DISCRETE ACTION SPACE)

SUBMITTED BY:

1. SAJIN MOHAMED PALLIKKATHODI ERATHALI (23037601)
2. AKSHATA BHEEMASAMUDRA MALLIKARJUNAPPA (23164204)

Table of Contents

1. Objectives

To implement a Reinforcement Learning (RL) agent using a Deep Q Network (DQN) applied to the Classic Control problem in OpenAI Gym. From the gym's classic control environments provided, we have chosen the mountain car problem (`MountainCar-v0`).

1.1 Why is Reinforcement Learning the machine learning paradigm of choice for this task?

The classic control problems are based on physics and real-world problems and mountain car problems is one of these classic control problems. Reinforcement Learning is a method of learning paradigm in Machine Learning which helps in creating solutions in real-time scenarios. The problem, being in a continuous state, requires the agent to learn and adapt continuously. This method helps the agent in sequential decision-making enabling it to learn from interaction with the environment and reach the goal by getting good rewards. Some well-known examples of RL models are Alpha Go, automated robots, recommendation systems etc.

2. The Gym Environment

The two main components of RL are –

- Agent is the learner and decision maker which is the car in our problem.
- Environment is a simulation that the agent interacts with and acts, which is the Mountain Car.

Learning can take place in a real environment however it is not feasible in terms of time and money. The gymnasium (previously known as gym) is an open-source python library that provides a simulated environment for an agent to interact with, and learn the different states available, which is done by giving standard APIs to communicate between learning algorithms and environments. The mountain car environment setup is discussed below.

2.1. The Setup

The environment consists of a 1-dimensional track located between two mountains where the car is placed and the objective of the problem is to get the car which is placed randomly at the base of a sinusoidal valley to climb up a mountain by accelerating right or left, as quickly as possible. The target is to reach the flag placed at the top of the hill on the right side with as few steps as possible. Here we have 2 states in the observation space which are position of the car along the x-axis and velocity of the car and 3 actions in the action space denoted by,

0 - accelerate to left
1 - don't accelerate
2 - accelerate to right

- The position range is clipped to the range of [-1.2, 0.6] and the velocity range is clipped to the range of [-0.07, 0.07].
- Each action taken is associated with a reward of -1. Hence the objective would be to reach the flag in as few steps as possible.

- Each episode is considered as done when the car takes either 200 steps without reaching the flag or if the car reaches the flag within 200 steps (which is the goal).

3. Hyper-Parameters

3.1. Epsilon(ε)

Epsilon is used to determine the action taken by the agent (car). This parameter helps us control the amount of exploration and exploitation an agent can perform in the environment from the values learned in the previous runs. Since we have adopted an ε-greedy policy, we started with a high value of epsilon as we need to explore the environment and try out various states and actions. Once the agent learns more about the environment, we need it to exploit the values learned to make better decisions on each step. To achieve this, we used Epsilon Decay with a rate of 0.99 times its previous value. We kept a threshold at 0.001 as the lowest value that epsilon can assume. Figure 1 shows the epsilon decay policy adopted.

We tried to start the epsilon at 1 and gradually decrease it. The problem with this approach was that the agent explored more which led to it starting to learn at a much later episode (closer to 280 episodes), when compared to an example of ε starting at 0.1 where it started to learn at 89 episodes and applying the same decay rate on it.

This tells us when applying ε-greedy policy for mountain cars we should not let the agent explore more if we want a faster learning graph.

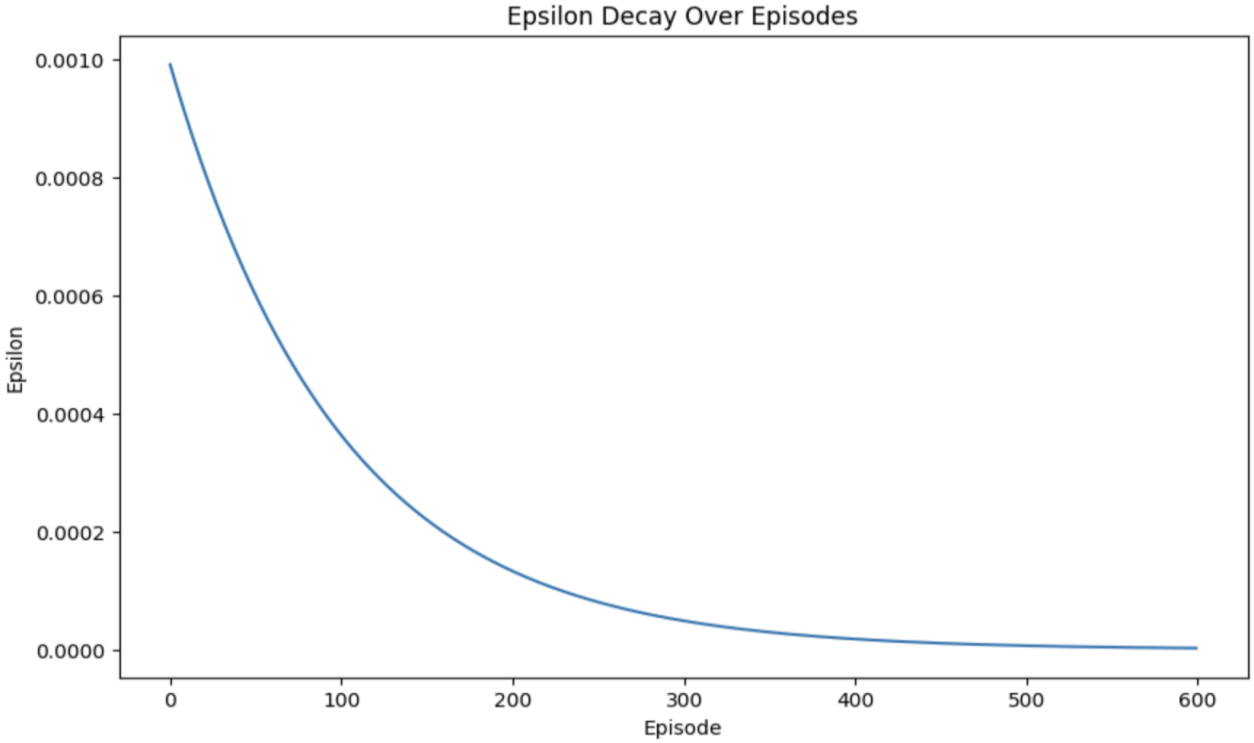

Figure 1. The Epsilon Decay approach in the notebook

3.2. Replay Memory Size

Replay memory is used when obtaining the sample experience for training. This is created as a double-ended queue in our code, which means as new records enter the queue the older records are removed. Initially we started with a small replay memory of size 1000 where we provided the minimum size of the replay memory as 50, i.e. 50 records of (state, action, reward, next_state, done) should be entered into the replay memory (1 record for each action taken by the agent) before the training of the DQN can start. This meant once 50 steps were run the replay buffer had enough records that it can start training the networks. This did not yield a good result as the maximum steps allowed by the gym environment is 200 hence it started to learn from the second episode which was our intent. However, this also meant that the agent did not do much exploration of its states and actions, thereby taking a long time to produce any results.

Based on this observation we improved the minimum size of the replay memory to 1000 and increased the size of the replay memory deque to 100,000 for it to store a large number of results from which we randomly sample 'batch_size' number of experiences for training the network.

3.3. Mini-batch size

Since the experiments were run for 600 episodes, we started with the mini-batch size as 32 and were able to obtain a good learning. Logically if we increase the number of samples obtained from the replay buffer, we should be able to get a better result as the number of samples would increase which would lead to better learning as there would be more state action pairs taken for the application of Bellman equation. However, because we were running it only for 600 episodes and increasing the number of samples by N leads to N more calls to the predict method of the model, which leads to the increasing the time taken for training and updating the mini-batch size to 64 did not yield a much better result, we continued the experiments with mini-batch size as 32.

3.4. Discount Factor

"The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ_{k-1} times what it would be worth if it were received immediately".(Sutton and Barto, 2018). The empirical data suggests that the discount factor can be kept between .95 and .99. There was no observable change when altering this between .95 and .99, which suggests that the agent's ability to balance immediate reward against long-term goals remain consistent in the above-mentioned range.

3.5. Network Structure

The final network structure we decided after many trials involves two hidden layers with 400 neurons in the first hidden layer and 300 neurons in the second hidden layer, while the input layer and output layer have 2 and 3 neurons respectively. We started the testing with 3 hidden

layers having 64, 128 and 256 neurons in each layer respectively. However, the computation time increased and it did not enhance the learning of the agent. The network exhibited learning behavior only at around 250 episodes. Hence, we wanted to increase the number of neurons and reduce the number of hidden layers. This led us to the above choice which gave us an acceptable performance with respect to time taken to train and learning rate.

The learning rate chosen is 0.005 and the Optimizer is Adam because it provided a better result when compared to NAdam and other learning rates.

3.6. Number of Episodes and Steps

The maximum number of Episodes chosen is 600. It suffices to present the results up to 600 episodes as this number demonstrates observable learning behavior, as evidenced by the Rewards vs. Episodes plots for all the scenarios tested.
The maximum steps are defined by the gym environment as 200, which means the car would take a maximum of 200 steps in each episode.

4. DQN

4.1 Training Step

4.1.1. Data capture and Sampling

The training step involves sampling data from the replay buffer, also known as Experience Replay. The Experience Replay is formed by storing results of performing **ε-greedy action** on the environment and the next state and reward information is obtained, other than the done and info associated with the action. The data from the replay buffer is sampled at random with a batch size of 32 and is used as training data for the train network (model).

NOTE: The same data capturing and Sampling methods are applied throughout the experiments.

The training is done with the help of Bellman optimality equation whereby we obtain a way to estimate the optimal state value of any states Q (s, a) = sum of all discounted future rewards the agent can expect on average after it reaches a state s, assuming it acts optimally.

The equation for DQN is,

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max Q(s', a') - Q(s,a)]$$

```python
# predict state action values
cur_action_values = model.predict(X_cur_states, verbose=0)

# action values for the next_states taken from our agent (Q network)
next_action_values = model.predict(X_next_states, verbose=0)

for index, sample in enumerate(minibatch):

    cur_state, action, reward, next_state, done = sample
    # estimate the optimal state action values
    if not done:
        # Bellman's optimality equation
        '''A way to estimate the optimal state value of any state s(Q(st, at)) = sum of all discounted future rewards
            the agent can expect on average after it reaches a state s, assuming it acts optimally'''
        # Q(st, at) = reward + DISCOUNT * max(Q(s(t+1), a(t+1)))
        cur_action_values[index][action] = reward + DISCOUNT * np.amax(next_action_values[index])
    else:
        # Q(st, at) = reward
        cur_action_values[index][action] = reward
```

Figure 2. Vanilla DQN: calculation of Q values

Here we use a single neural network(model) to estimate Q-values. The same network is used to select the action that maximizes the Q-values(np.amax(next_action_values)) and to calculate target Q-values. In Figure 2, we can see that the model.predict() method is used to estimate the Q-values for the current and next states. The Q-values for the current state are then updated based on the rewards and the maximum Q-values for the next state, following the Bellman equation. This updated Q-values are then used to train the model. This is a basic implementation of a Vanilla DQN illustrated in Figure 2.

4.2. Stratagem - Approaches to enhance the learning of the agent

4.2.1. Discretization

The aim was to enhance the learning of the agent by discretizing the states as illustrated in Figure 3. Even though the action space has discrete states [0,1,2] we tried to discretize the state space [position, velocity]. Since the problem is set in a continuous state space, that is, the position and velocity changes for every step the agent takes and is a continuous value from -1.2 to 0.6 for position and -1.7 to 1.7 for velocity. Agent can take on an infinite number of possible states, discretizing these states (as illustrated in Figure 3 and Figure 4) by putting them into a grid of bins can help reduce the number of states enhancing the learning process.

```python
num_bins = 500
position_bins = np.linspace(env.observation_space.low[0], env.observation_space.high[0], num_bins)
velocity_bins = np.linspace(env.observation_space.low[1], env.observation_space.high[1], num_bins)
```

Figure 3. Creating position and velocity bins

```python
def discretize_state(state):
    position, velocity = state
    position_discrete = np.digitize(position, position_bins) - 1
    velocity_discrete = np.digitize(velocity, velocity_bins) - 1
    return (position_discrete, velocity_discrete)
```

Figure 4. Defining function to discretizing the states

For each episode, the Q table is updated with discretized values on which the agent learns and takes the next action.

4.2.1.1. Observations and Results

The agent managed to receive a huge reward at once, but this took a minimum of 150-200 episodes to learn. With discretization, there was not much learning but most of the time it learned, it had managed to learn the actions that gave it the huge maximum reward. This pattern repeated for different number of bins, but 250 bins proved to be the most optimal value of bin while discretizing. We can also see from the below observations the model depicts a stochastic behavior making it difficult to analyse the best result of it learning.

10 bins - The agent received a best maximum reward of 134 at 120th episode when the epsilon rate was at 0.03148.
100 bins - The agent received a best maximum reward of 118 at 54th episode when the epsilon rate was at 0.061111.
250 bins - The agent received a best maximum reward of 110 at 224th episode when the epsilon rate was at 0.01106.
500 bins - The agent received a best maximum reward of 116 at 184th episode when the epsilon rate was at 0.0165.

The three plots in Figure 5, Figure 6, and Figure 7 show the average of rewards received over 600 episodes when the discrete state bins are 100, 250 and 500 respectively. There is not much learning by the agent when the state space is put into 100 bins. However, when the states are put into 250 bins, the agent learns better but is delayed by 50 episodes compared to agent learning in 100 bins. The agent when using 500 bins only starts learning after 400 episodes. However, as seen by the moving average graph, the learning is not steady and makes it difficult for the agent to converge towards optimal policy.

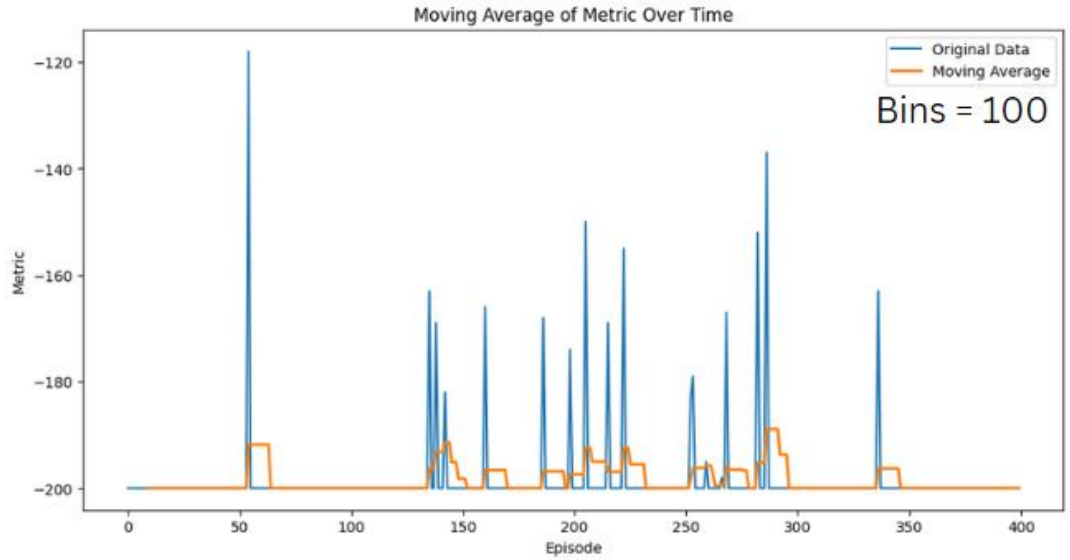

Figure 5. Moving average of rewards over episodes for 100 bins of discrete state

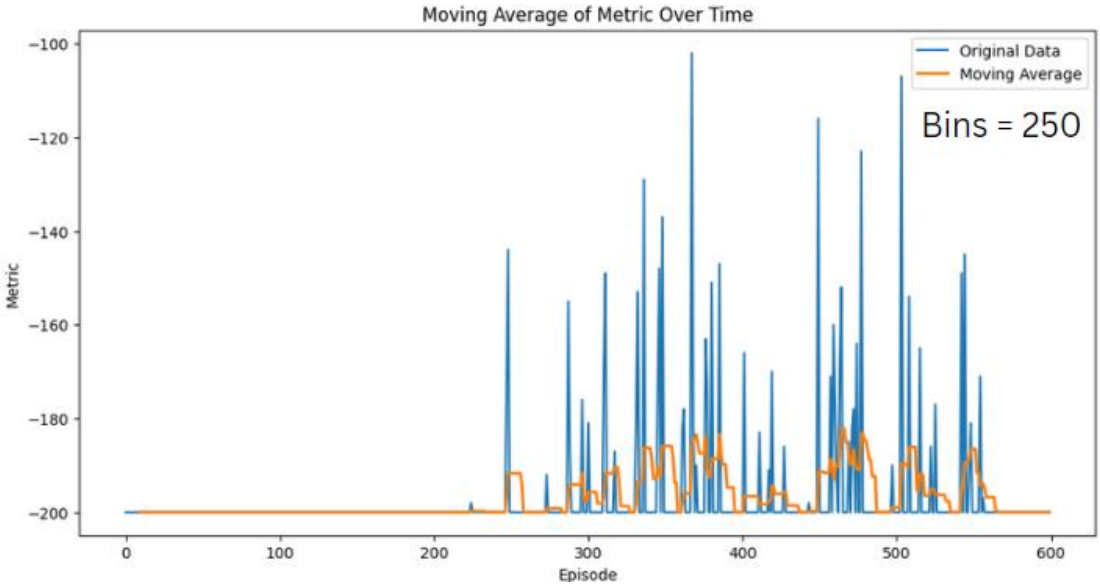

Figure 6. Moving average of rewards over episodes for 250 bins of discrete states

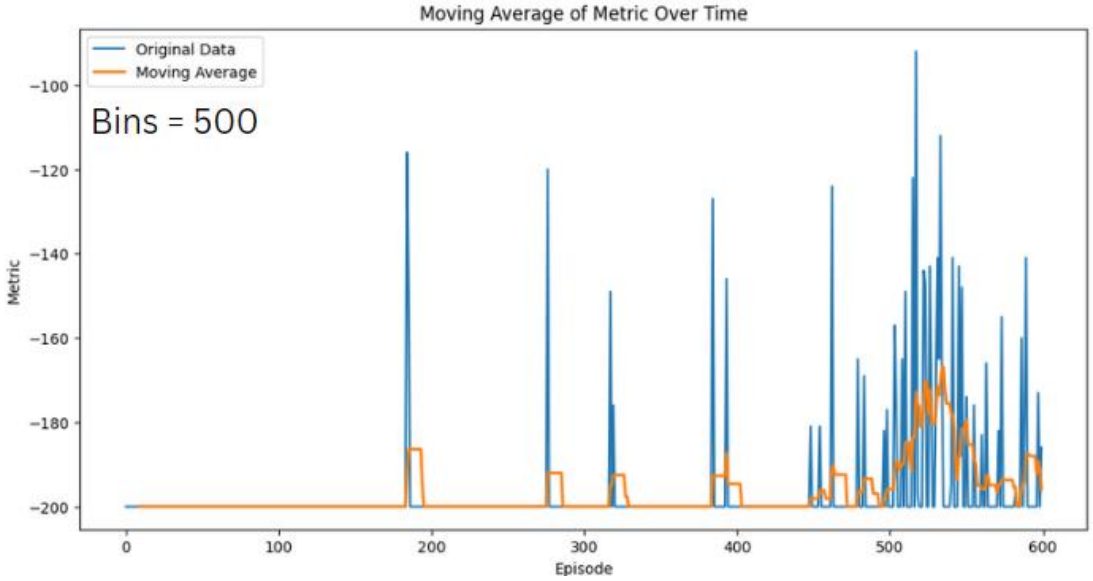

Figure 7. Moving average of rewards over episodes for 500 bins of discrete states

4.2.2 Deceiving the agent.

The aim of this approach was to make the agent learn the optimal policy by deceiving the agent to acquire a higher reward of 250+ episodic rewards when the agent reaches the flagpole under 200 steps (episode_length). This is illustrated.

```
def play_one_step(env, state, epsilon, episode_length):
    # defining reward acquired in each episode
    global episode_reward
    # Calling epsilon greedy function to select action
    action = epsilon_greedy_policy(state, epsilon)
    # Executing the action and returns 4 values
    next_state, reward, done, info = env.step(action)
    # accumulating rewards in each step of the epsiode(max=-200)
    episode_reward += reward
    # truncating if the car reaches the flag on the mountain top within 200 steps
    if done and episode_length < 200:
        # If episode is ended then we have won the game. Defining Stratagem to make the agent learn faster
        reward = 250 + episode_reward
        # save the model weights if we get a epsiode reward greater than maximum reward defined
        if(episode_reward > max_reward):
            print("Saving the model with reward", episode_reward)
            model.save_weights("agent"+str(episode_reward)+"_agent_.weights.h5")
    else:
        # In oher cases reward will be proportional to the distance that car has travelled
        # from it's previous location + velocity of the car
        reward = 5*abs(next_state[0] - state[0]) + 3*abs(state[1])

    # append this result to the replay buffer(experince replay)
    replay_buffer.append((state, action, reward, next_state, done))

    return next_state, action, episode_reward, max_reward, done, info
```

Figure 8. Code snippet strategizing the agents process for learning the policy.

If the agent doesn't reach the flagpole within 200 steps, then we assign the reward by enhancing the position and velocity by 5 and 3 times respectively. Hereby, ensuring the position is given more priority than the velocity. This is depicted in Figure 8.

4.3. Loss Function for DQN

Since at the start of the training we start with random weights the predicted Q-values would be different from the target Q-value(r + maxa′Q(s′, a′)). Therefore, at each training step we aim to minimize the squared difference between the predicted Q-value and the target Q-value obtained from the target network.

$$L(\theta) = [Q(s, a) - r - \gamma \max_{a'} Q(s', a')]^2$$

An illustration:

Say we do a random sampling of states from the experience replay (replay buffer), and obtain the result as (s, left, s', -1 => Moving from s to s' when action left is taken and receives a reward of -1), With state s, suppose we do a forward pass through our Q network and for action 'left' it gives us a Q-value of -100. Then,

$$Q(s,\text{left}) = -100.$$

Now we pass s′ to the network, which returns a Q-value of -110 for left and -102 for right. As per the Bellman equation, we take the max of these which is -102 for the right, for s′, thus.

$$\max_{a'} Q(s',a') = -102.$$

But we know that reward r was -1. So, our Q network prediction was wrong initially, because as per the Loss function illustrated above, we get a loss value of 0.04 as illustrated below.

$$L(\theta) = [-100 - 1 + 0.99*102]^2 = 0.04.$$

So, we back-propagate the error and correct the weights θ slightly using the neural networks. Now if we are to calculate the same transition it would be better. This activity is continued for each episode by taking some random values from the experience replay and we update the network weights.

NOTE: The Loss function used is Mean Squared Error throughout the experiments as MSE is empirically proven to be useful when working with Q-learning or Deep Q-Networks (DQN) to minimize the difference between predicted and target Q-values.

4.4. Results for DQN

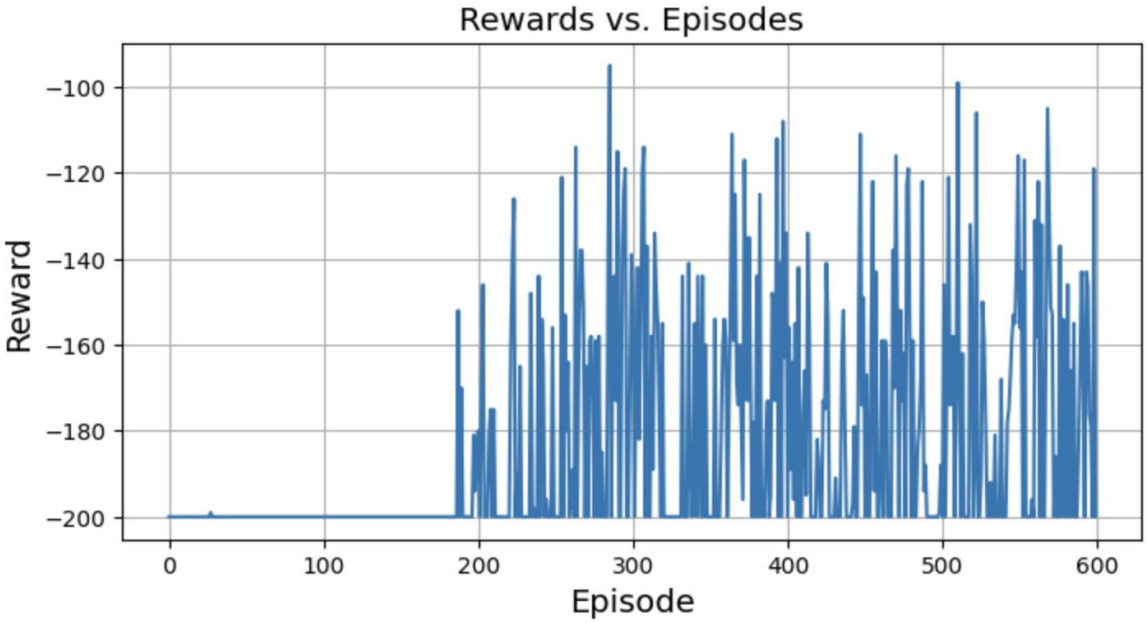

Figure 9. Episode vs Reward for Vanilla DQN

The plot of Rewards vs. Episodes shown in Figure 9 illustrate that the network with 400, 300, 3 structure is learning as it exhibits the car reaching the flagpole in less than 200 steps, which is shown as the reward obtained is changing its value from -200 to a value greater than that at subsequent steps (for instance at 280th step the car reached the flagpole at 84 steps).

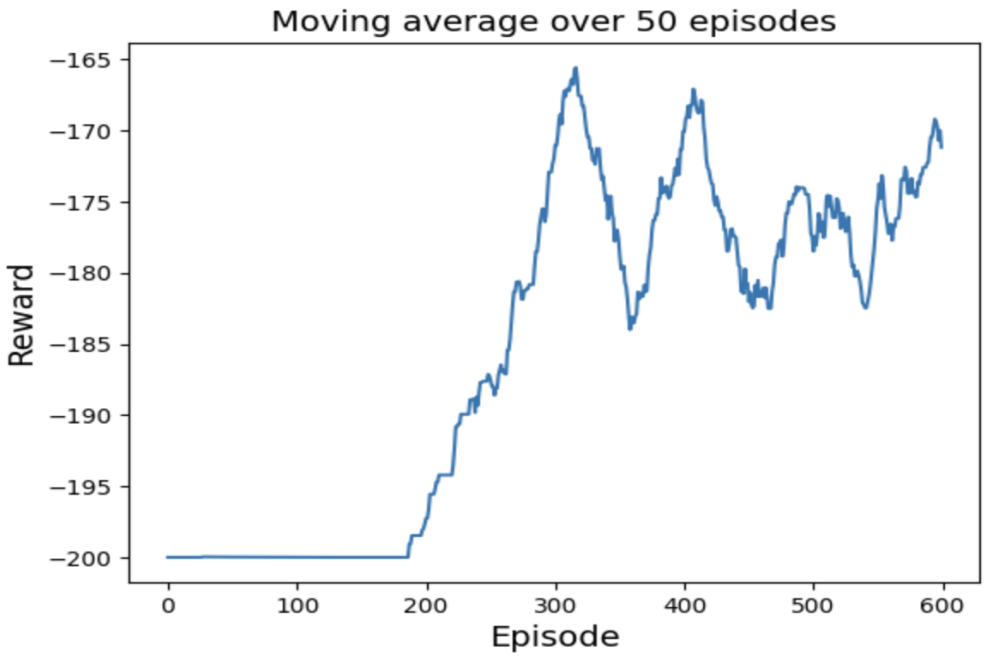

Figure 10. Moving average over the last 50 episodes for Vanilla DQN

The moving average over the last 50 episodes displayed in Figure 10 shows this more clearly where, after 190 episodes the graph shows a consistent rise until the 300th episode. The figure also illustrates the catastrophic forgetting(described later in the report) after the 300th episode where we have a steady fall in the graph preceding the rise again at the 370th episode.

This design is having a flaw in that we do two forward passes through the network which leads to an unstable learning. To solve this problem, we introduced Fixed Deep-Q-Network.

5. Fixed Deep-Q-Network

5.1. How Fixed DQN Works

While it is possible to build a Deep Q Network (DQN) without any target network i.e. using only one main neural network where we perform two passes through the Q network, the first one to output the predicted Q value and second one to output the target Q value.

Here the weights of the Q Network are updated on each step whereby we improve the predictions of the Q value. However, since the weights of the two networks are the same each time, we run through different steps due to the training done on the same Q network, this changes the direction of the targeted Q-value. This would be like chasing the target where it gets updated on each step. This is illustrated in the figures from Manning(2024) Figure 11 and Figure 12.

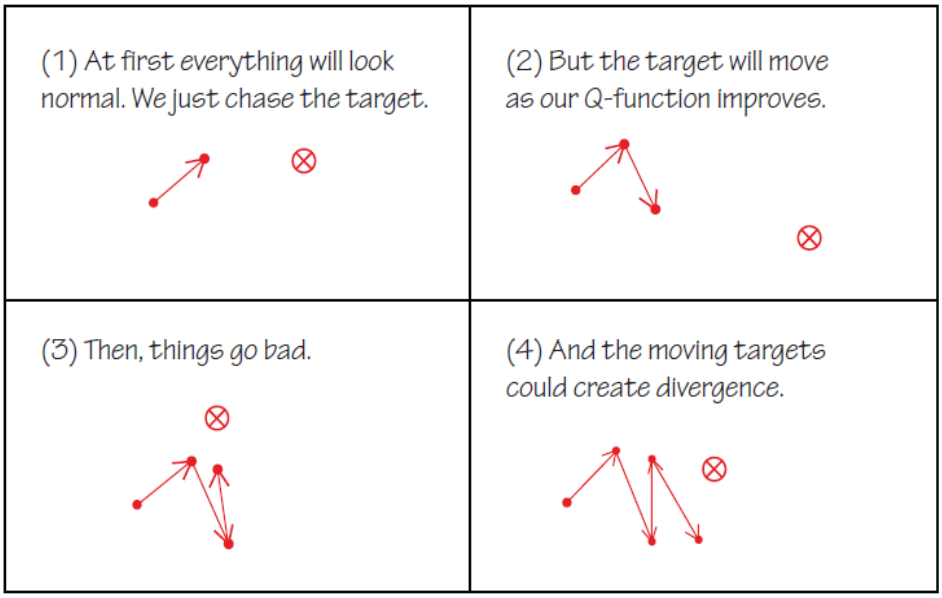

Figure 11. Q-value approximation without target network

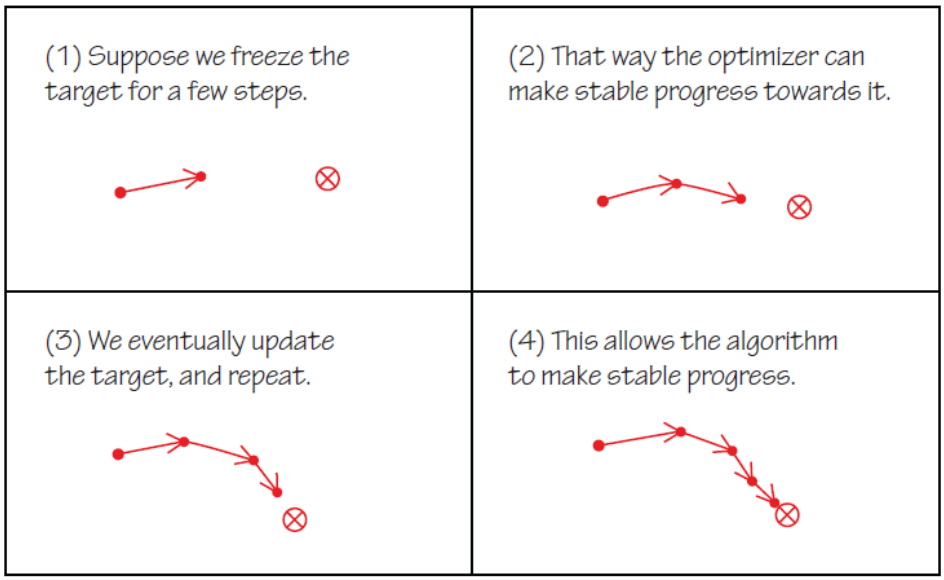

Figure 12. Q-value approximation with target network

Hence, by using a second network that does not update the weight at each step in the episode we obtain a stable result for a short period of time. In our code we update the target network with the weights of the training network for every 20 episodes i.e. every 4000 steps.
This ensures the target network results in more stable training.

5.2. Fixed DQN Implementation

In our implementation we update the weights of the target network every 20 episodes (roughly 4000 steps since we are running the experiments for 600 episodes)

The training of both the train and target networks starts with random weights. The samples obtained from the replay buffer are used as input for both the training network and the target network and we make the train_model to predict the current action values i.e., all the actions that can be taken from the current state. This is the initial predicted Q values on top of which the target network (target_model) is used to predict the next action values i.e., all the actions that can be taken from the next state and selects the maximum value and we use the Bellman equation to assign the Q value for all actions.

$$Q(s,a) = \text{reward} + \gamma * \max_{a'} Q(s',a'),$$

Where s' is the next state after s, γ is the discount factor.

This code for this is illustrated in Figure 13

```
# Initialize an array to store the Q values for the current state-action pairs
cur_action_values = model.predict(X_cur_states, verbose=0)

# Use the target model to predict the Q values for the next states
next_action_values = target_model1.predict(X_next_states, verbose=0)

'''Update the Q values for the current state-action pairs based on the rewards
   and the maximum predicted Q values for the next states'''
for index, sample in enumerate(minibatch):
    cur_state, action, reward, next_state, done = sample
    if not done:
        # if episode is not done then we have to calculate the Q value using Bellman's equation
        # Q(st, at) = rt + DISCOUNT * max(Q(s(t+1), a(t+1)))
        cur_action_values[index][action] = reward + DISCOUNT * np.amax(next_action_values[index])
    else:
        # if episode is done then Q value will be the reward
        # Q(st, at) = rt
        cur_action_values[index][action] = reward

# train the agent with new Q values for the states and the actions
model.fit(X_cur_states, cur_action_values, verbose=0)
```

Figure 13. Code for Fixed DQN

The target Q value is the output of the target network($\max_{a'} Q(s',a')$) + reward from sample. **NOTE**: the output of the target network is multiplied by the Discount Factor for reducing the impact of the later actions on the current actions and making the results focus on the immediate actions more than the later actions.

The equation for Fixed DQN is,

$$Q_{train}(s,a) = Q_{train}(s,a) + \alpha[R(s,a) + \gamma \max Q_{target}(s', a') - Q_{train}(s,a)]$$

With the Bellman equation we are approximating a good Q-function using which we update the weights of the neural network (train and target network). The Training network is updated at every step; however, we update the Target network only every 20 episodes to maintain stability in the training process. Finally we call the fit() method on the training network where we have defined

the model using the loss function as Mean Square Error and Optimizer as Adam with a learning rate of (0.005)

5.3. Results

The rewards obtained from the Fixed DQN is illustrated in Figure 14. against each episode. As we can see the results show a less erratic graph when compared to the Vanilla DQN. This is because of the introduction of the target network into the code, where we have fixed the target for 20 episodes, each episode having 200 steps as a maximum. The figures below show the network exhibiting a learning from 100th episode and continuously exhibiting the learning behavior with **catastrophic forgetting** exhibited in between.

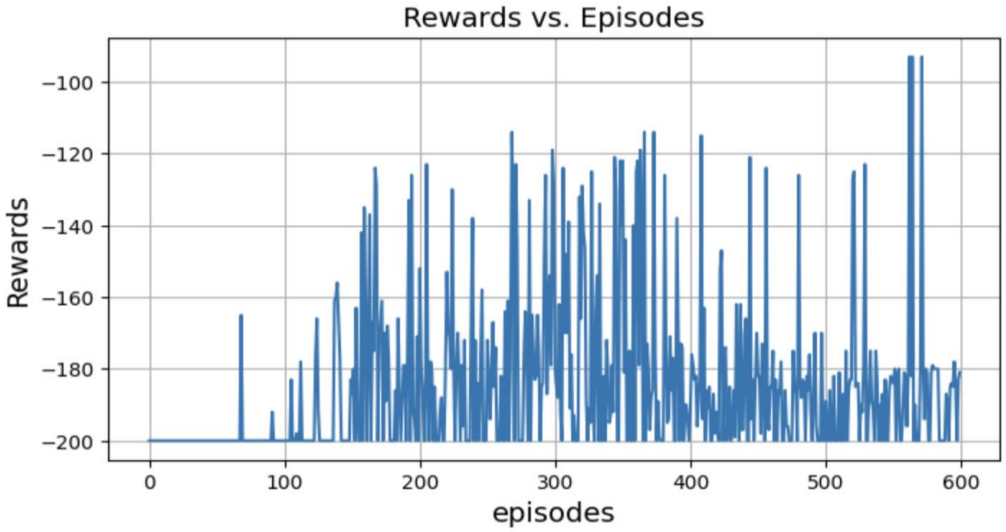

Figure 14. Rewards vs Episodes

The moving average provides valuable insights into the training dynamics and performance of the reinforcement learning algorithm. The figure 15. shows the plot of the moving average over the last 50 episodes which more clearly illustrates the learning and the catastrophic forgetting behavior.

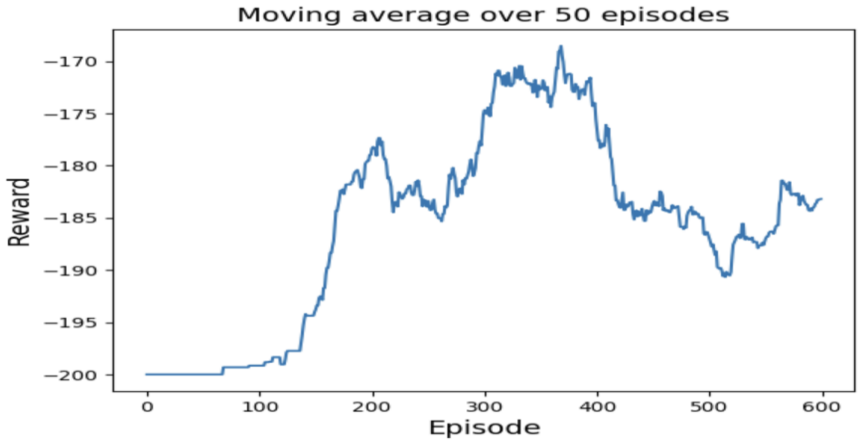

Figure 15. Moving Average rewards over the last 50 episodes

Figure 16. shows the action chosen at each state (position, velocity). This plot tells us that, on the provided set of action space (0-push left, 1-do nothing, 2-push right) the car chose to move left and right more often than to stay and do nothing. This also tells us that as per the Bellman equation, the agent calculated that choosing a left (depicted using blue points) or right (depicted using red points) action provides more reward, when compared to staying idle (depicted using green points).

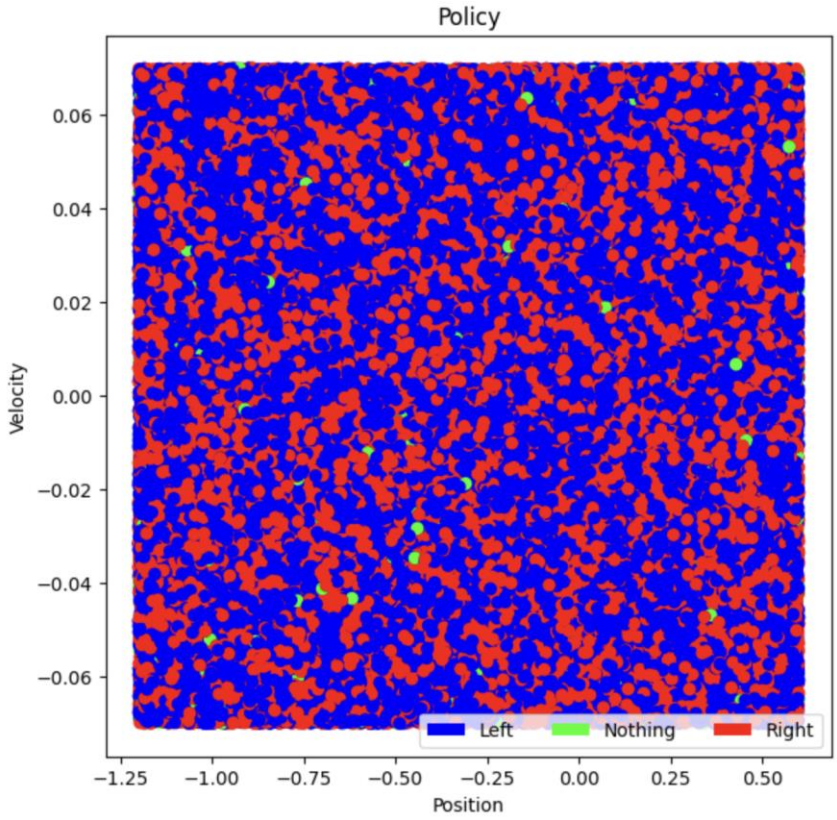

Figure 16. Position vs velocity depicting the action chosen by the car.

6. Double DQN

6.1 Why we need Double DQN

There are two problems that are faced by vanilla DQN and Fixed DQN which are,

A. Maximization bias,
B. Catastrophic forgetting

Catastrophic forgetting is a situation within the environment when the agent does not have access to previously recorded data. Random sampling, that is, the process of randomly picking the state action values from the experience buffer (replay buffer) is one of the causes for this, it can be addressed by prioritized experience replay and Double DQN does not contribute much for the solution.

If we recall from the previous two implementations, we are using $\max_{a'} Q(s',a')$ to get the maximum state-action value from the Q table.
Taking the maximum values of the state-action values leads to the agent overestimating the action's value which will lead to choosing that action as the best action and hence the overestimated value will be used as the target. This might lead to the agent learning the Q values incorrectly and is referred to as the Maximization bias.

6.2 Double DQN Working.

This problem is addressed by using a Double DQN, as the name suggests, it makes use of two Q networks, that is it uses two different neural network models to select an action and perform an evaluation on that action independently during training. This method uses two separate Q functions to estimate the values and each of these Q functions are updated using one another. While updating Q1, the selection of the best action is done by Q1, however the estimation of its value is carried out by Q2 to tackle overestimating the state-action value to an extent.
The updated equation for Double DQN is,

$$Q_{train}(s,a) = Q_{train}(s,a) + \alpha[R(s,a) + \gamma\, Q_{target}(s', \max Q_{train}(s',a')) - Q_{train}(s,a)]$$

6.3 Double DQN Implementation

As seen in Figure 17, the training_step function starts by sampling a minibatch_size batches of experiences from the replay buffer. X_cur_states and X_next_states are arrays containing the current state and next state values from each sample experience which will be fed into the neural networks. Current Q values and next state Q values are predicted using the main neural network. Target neural network is used to predict the next state Q values and best_action_next selects the action predicted from the main neural network while the updated Bellman's equation uses the Q value predicted from target network to estimate this action chosen by the main neural network. Model.fit is called to update the weights where X-cur_states is the input and cur_action_values is the output. The goal here is to minimize the error between the targeted Q values and the predicted Q values to learn the policy optimally.

"Suppose we divided the plays in two sets and used them to learn two independent estimates, call them Q1(a) and Q2(a), each an estimate of the true value q(a), for all a 2 A. We could then use one estimate, say Q1, to determine the maximizing action A⇤ = argmaxa Q1(a), and the other, Q2, to provide the estimate of its value, Q2(A⇤) = Q2(argmaxa Q1(a)). This estimate will then be unbiased in the sense that E[Q2(A⇤)] = q(A⇤)." (Sutton and Barto, 2018)

```python
def training_step(MINIBATCH_SIZE):
    # Samples batches of size MiniBatch_size(32) from  the replay buffer
    minibatch = random.sample(replay_buffer, MINIBATCH_SIZE)
    # Array containing current state values from each sample in the minibatch as the replay buffer looks like (state, action, reward, next_state, done)
    X_cur_states = np.array([sample[0] for sample in minibatch])
    # Array containing next state values from each sample in the minibatch as the replay buffer looks like (state, action, reward, next_state, done)
    X_next_states = np.array([sample[3] for sample in minibatch])

    # Predicting the action using main neural network
    '''The inputs for the Neural Network selecting the action using main network.
    This uses the main neural network model to predict the current Q-values for all actions given the current states'''
    cur_action_values = model.predict(X_cur_states, verbose=0)

    ''' This step predicts the Q-values for the next states using the same main network and
    helps to decide the next action to take'''
    next_action_values_main = model.predict(X_next_states, verbose=0)
    # Evaluating the action using target network
    ''' This uses a separate target network to predict the Q-values for the next states.
    This target network is an older version of the main network and is used to stabilize the training'''
    next_action_values_target = target_model1.predict(X_next_states, verbose=0)

    # This loop iterates over each sample from sample experience
    for index, (cur_state, action, reward, next_state, done) in enumerate(minibatch):
        # If the episode is not completed,
        if not done:
            # Return action with highest Q value in next state as per the main neural network
            best_action_next = np.argmax(next_action_values_main[index])
            ''' The bellman equation is updated to to return Q value for the action taken using reward,
            discount and best next action from the target network and not the main network'''
            cur_action_values[index][action] = reward + DISCOUNT * next_action_values_target[index][best_action_next]
        else:
            cur_action_values[index][action] = reward
    # updates model weights using Q values to minimize the error between target value and predicted value.
```

Figure 17. Code for Double DQN

6.4. Results

The results of Double DQN are as follows:

Figure 18 depicts the rewards the agent has learnt over 600 episodes. The agent initially explores for a very few episodes and received a few rewards within 30 episodes however it explored even further until 250 episodes and vigorous learning begins from 300th episode where the agent exploits the environment to get the maximum reward. However, we can see the rewards being set back to -200 between 300th and 400th episodes which suggests catastrophic forgetting. After 400th episode

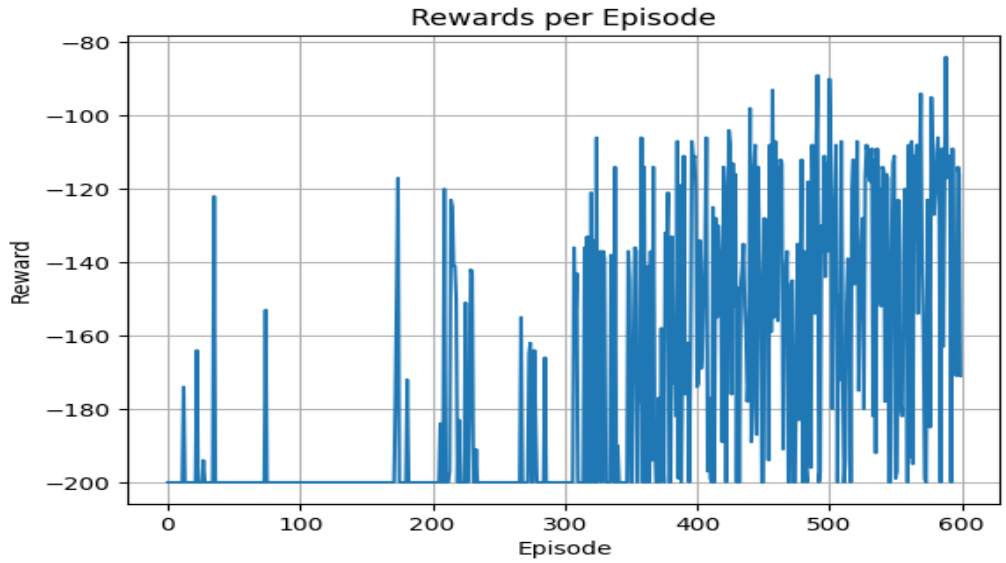

Figure 18. Rewards received per episode for Double DQN

Figure 19 depicts the Mean of rewards (in orange line) and each reward (in blue line) received over the length of episodes. The mean of rewards graph shows the performance of the agent over a length of episodes of 600 without the noise of original data (which is the rewards that gets varied at each point). The agent explores the environment in the first half of the episodes length but as it starts learning, we can see steady improvement in its learning trying to maximize the rewards achieving the maximum reward at -84.

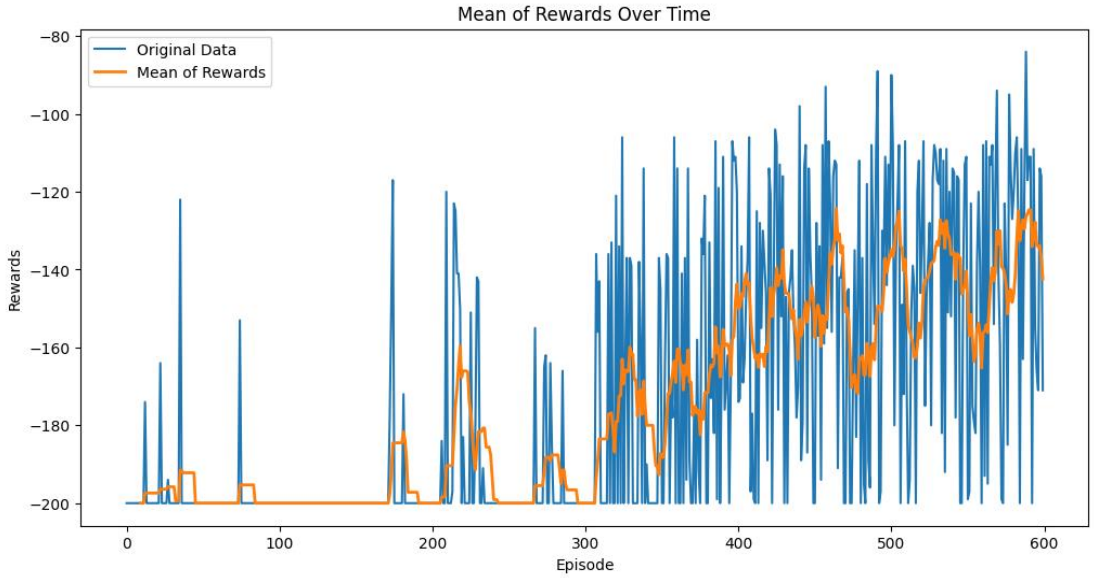

Figure 19. Average rewards received over time for Double DQN

Figure 20 shows the average of rewards moving over a window size of 50 episodes. It is evident that the car has managed to steadily learn to get to the goal which is shown by the graph lines moving up constantly and this is the stability of learning the Double DQN offers. However, this was only achievable from the 300th episode and until then the rewards received remained stagnant.

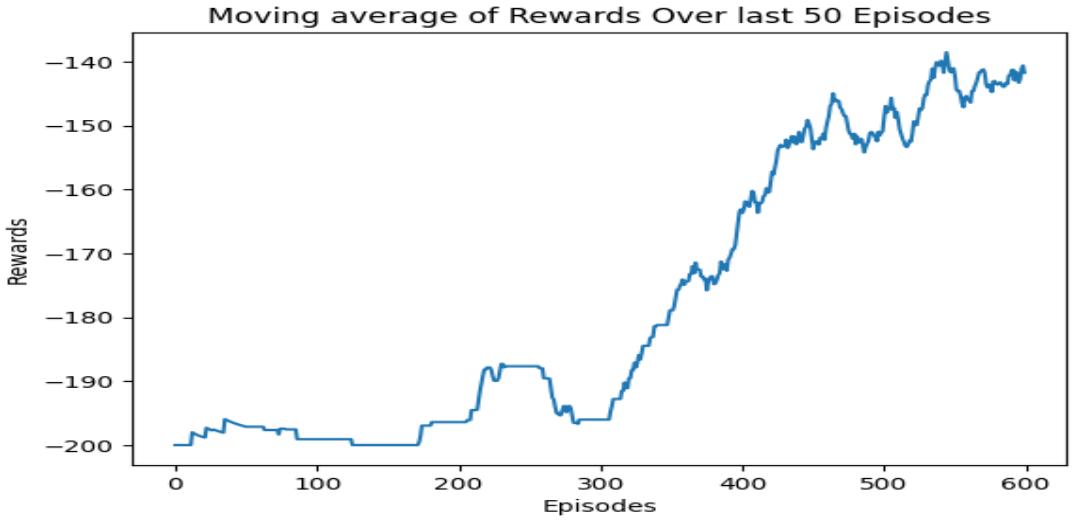

Figure 20. Moving Average of Rewards

Figure 21. depicts the number of steps the agent has taken in each episode before it was terminated or truncated.

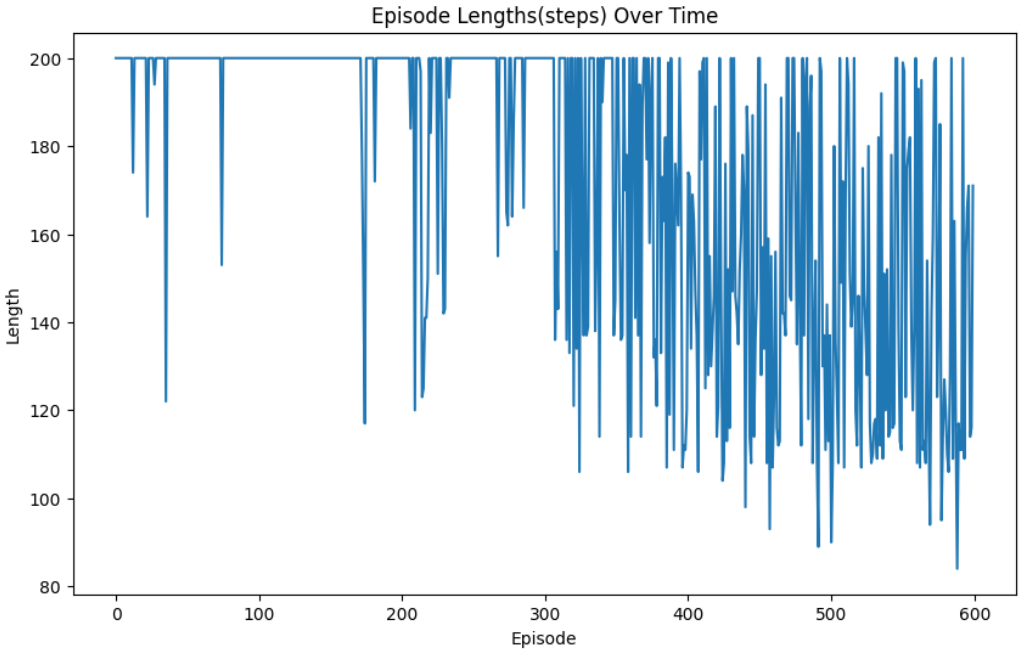

Figure 21. Number of steps taken in each episode

7. Double DQN with Double Q learning

7.1 How Double DQN with Double Learning works

In this implementation, on top of using two different neural networks to separately select an action and evaluate an action, the two networks can be swapped based on "flip coin" approach to *randomly select the network for prediction and evaluation of the Q values* as stated by Sutton and Barto(2018)

"Divide the samples into two halves. For the first half, we can use the above equation to train, and for the second half swap the "1" with the "2" in the equation. In practice, data keeps coming into the agent, and thus a coin is flipped to decide if the above equation should be used or the swapped one. How does this help? Since the samples are stochastic, it is less likely that both the halves of the samples are overestimating the same action. The above method separates two crucial parts of the algorithm, choosing the best action and using the estimate of that action, and hence works well." Ameet Deshpande(2018)

The two networks are used to decouple the training process by updating action selection network less frequently and is used to estimate the Q value of the selected action which helps in reducing the overestimation of action values and improves the stability of the learning process, aiming to efficiently converge on an optimal policy.

7.2. Double DQN with Double Q Learning Implementation

```
cur_action_values = model.predict(X_cur_states, verbose=0)

'''The inputs for the Neural Network selecting the action using target network.
This uses the target neural network model to predict the current Q-values for all actions given the current states'''
cur_action_values_target = target_model1.predict(X_cur_states, verbose=0)
''' This step predicts the Q-values for the next states using the same main network and
helps to decide the next action to take'''
# Q1(a)
next_action_values_main = model.predict(X_next_states, verbose=0)
# Evaluating the action using target network
''' This uses a separate target network to predict the Q-values for the next states.
This target network is an older version of the main network and is used to stabilize the training'''
# Q2(a)
next_action_values_target = target_model1.predict(X_next_states, verbose=0)
# This loop iterates over each sample from sample experience
for index, (cur_state, action, reward, next_state, done) in enumerate(minibatch):
    # If the episode is not completed,

    if not done:
        ''' Using "coin flip" approach to randomly select the networks to be used
        for predicting an action and evaluate the selected action during training'''
        if np.random.rand() < 0.5:
            # Return action with highest Q value in next state as per the main neural network
            # Q2(A*) = Q2(argmax(Q1(a)))
            best_action_next = np.argmax(next_action_values_main[index])
            ''' The action selection (the 'argmax' in the max Q-value) for the next state is done using the main network,
                while the Q-value of this selected action is estimated using the target network.
                This helps to decouple the action selection from the target Q-value estimation,
                reducing the overestimation bias of standard DQN.'''
            # E|Q2(A*)| = q(A*)
            cur_action_values[index][action] = reward + DISCOUNT * next_action_values_target[index][best_action_next]
        else:
            # Q1(A*) = Q1(argmax(Q2(a)))
            best_action_next = np.argmax(next_action_values_target[index])
            ''' The action selection (the 'argmax' in the max Q-value) for the next state is done using the target network,
                while the Q-value of this selected action is estimated using the main network.
                This helps to decouple the action selection from the target Q-value estimation,
                reducing the overestimation bias of standard DQN.'''
            # E|Q1(A*)| = q(A*)
            cur_action_values_target[index][action] = reward + DISCOUNT * next_action_values_main[index][best_action_next]
```

Figure 22. The code for Double DQN with Double Q learning Coin Flip Approach

In Figure 22, If the episode is not done, it uses a "flip coin" method (if a randomly generated number is less than 0.5) to decide whether to use the main network or the target network to evaluate the next action. Depending on this choice, it uses the Bellman equation to update the Q-values accordingly and we update the weights of both the main network and target network. If the episode is done, it only assigns the current reward to the Q-value of the taken action.

```
# updates model weights using Q values to minimize the error between target value and predicted value.
model.fit(X_cur_states, cur_action_values, verbose=0)
target_model1.fit(X_cur_states, cur_action_values_target, verbose=0)
```

Figure 23. The train network and target network fit methods are called here.

We also removed the updating of weights of the target model with the weight of the training model and called the fit() on the target network as well as the training network(model)as illustrated in Figure 23.

Even though this might compromise the stability achieved by the second neural network to an extent (not completely), due to the randomness factor in deciding which of the networks (train or target network) is used for selecting the best action and which of the networks is used for evaluating the action selected, we maintain the stability and thereby reduce the maximization

bias. The idea for removal of updating the weights of the target network stems from the work done by Ameet Deshpande(2018).

NOTE: Because we update both the networks when trying the coin flip approach, the logic of updating the target network with the weights of the train network does not make sense, hence we call the fit() method on the train network and thereby remove the updating of target network weights every 20 episodes.

Here the update rule stays the same as the Double DQN however we add a randomness to the update like:

If randomness < 0.5

$$Q_{train}(s,a) = Q_{train}(s,a) + \alpha[R(s,a) + \gamma\, Q_{target}(s', \max Q_{train}(s',a')) - Q_{train}(s,a)]$$

Else

$$Q_{target}(s,a) = Q_{target}(s,a) + \alpha[R(s,a) + \gamma\, Q_{train}(s', \max Q_{target}(s',a')) - Q_{target}(s,a)]$$

7.3. Results

The key observation here is that we are able to obtain a good result even after removal of the target network updating with the weights of the training network, by calling the fit() on the target network along with the randomness introduced to pick the network for selection of action and evaluation of action to determining the Q-values.

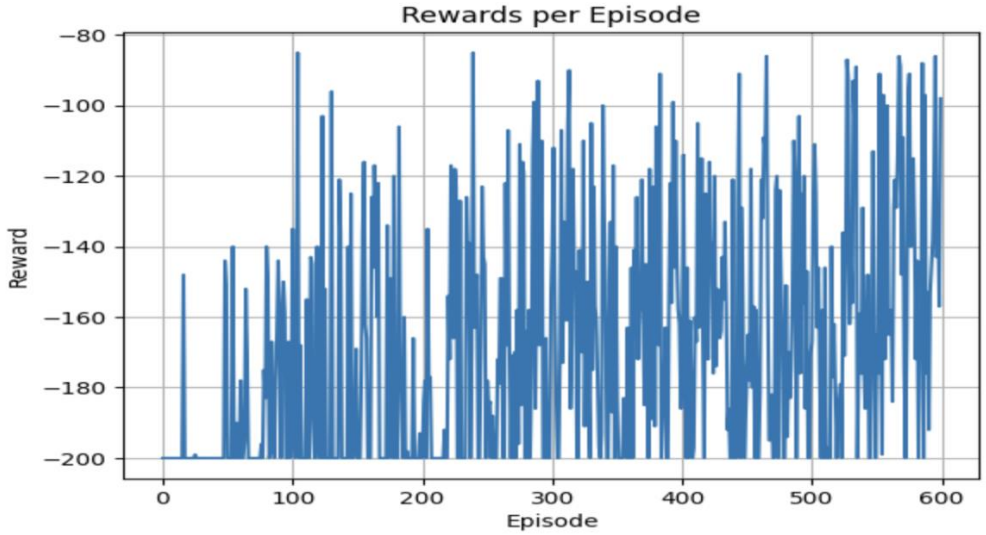

Figure 24. Reward vs Episode for Double DQN with Double Q learning Coin Flip Approach

Figure 24 shows the network starting to learn at 50 episodes and achieving a reward of 84 at 110th episode which means the car took only 84 steps to reach the flag on top of the mountain. However catastrophic forgetting makes it forget this result due to the randomness in the sampling. This is illustrated better in the Mean Rewards over time in Figure 25 which shows at around 200 episodes the mean reward goes back up to -200 i.e.. the car did not reach the flagpole even with

200 steps. However, it continued to learn as the orange line in the graph continues to rise even after the aforementioned dip.

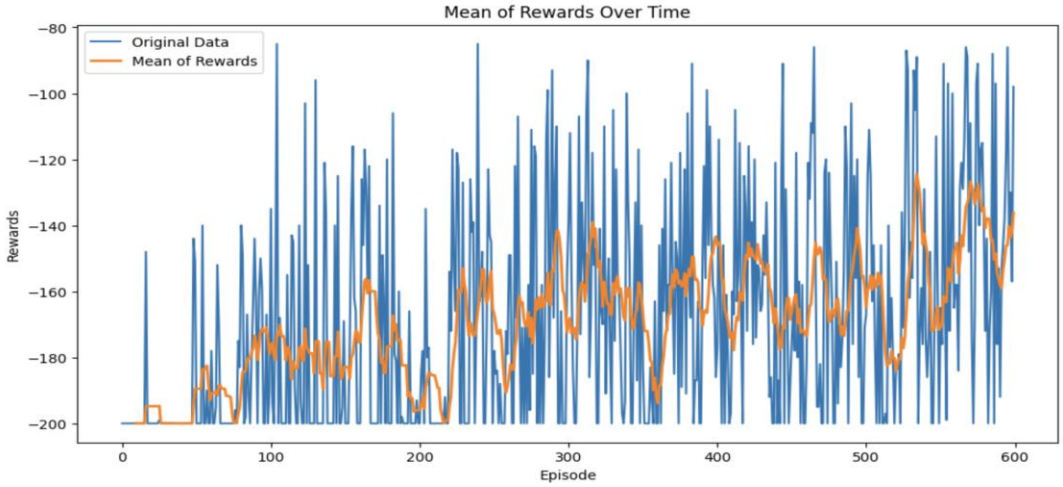

Figure 25. Mean Reward over Time for Double DQN with Double Q learning Coin Flip Approach

Figure 26 also highlights the above-mentioned scenario with the catastrophic forgetting. However, even with the removal of the updating of weights of the target network every 20 episodes we get a good learning as per the graphs in the figure. The agent achieves steady learning from early on in the episodes (from 100th episode) as compared to Double DQN alone (from 300th episode).

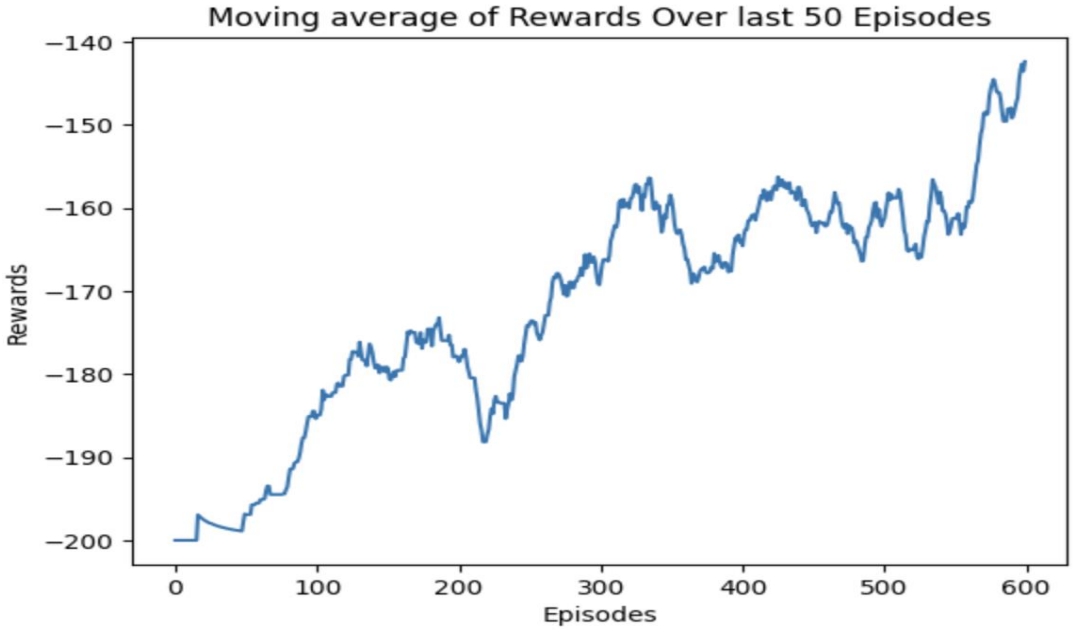

Figure 26. Moving average of Rewards over the last 50 Episodes for Double DQN with Double Q learning Coin Flip Approach

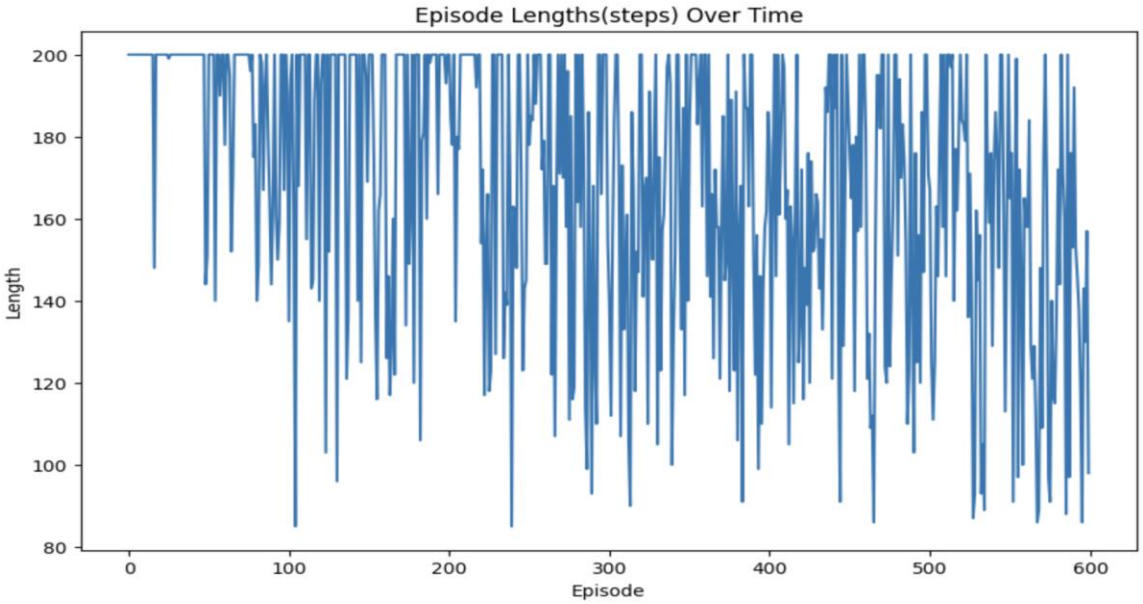

Figure 27. Episode Length over Time for Double DQN with Coin Flip Approach

The figure 27 provides an overview of the number of steps taken by the agent throughout the Episodes. We can see that the number of steps taken decreases as the network learns to predict the correct Q-value.

This implementation tried to achieve a tradeoff between maintaining the stability of the learning with the model while reducing the maximization bias.

8. Comparison of Different Q-learning Approaches

Bellman's equation helps the agent find the optimal policy to receive the maximum reward. Figure 28, below, depicts Bellman's optimality equation showing a sequence of improvements from DQN Learning to Double DQN with Double Q learning. Alpha is the learning rate and Gamma is the discount factor.

- Equation 1 is the basic Bellman equation for a DQN where the new Q value for a given state-action is updated using the old Q values and reward received for transitioning into next state plus the discounted Q value for the next state-action values.
- Equation 2 is a variation of DQN which makes use of a separate target network (Q_{target}) to calculate the temporal difference (error loss) which helps in stabilizing the learning of the agent.
- Equation 3a and 3b represent the Double Q learning where two different Q tables (Q_1 and Q_2) are used to address the maximization bias problem. As seen, selection of an action is done by one table and evaluation of that action is carried out by another Q table values to ensure the agent does not over-estimate that action alone.

- Equation 4 depicts the Bellman equation used in Double DQN implementation which uses the Double Q learning by using one network (Q_{train}) to select an action and (Q_{target}) to evaluate that action. It also adapts the Fixed Q learning approach to stabilize the learning.
- Equation 5a and 5b makes use of Double DQN learning however it uses Double Q learning on top of it to by randomly using Qtarget to evaluate the action for a certain set of episodes and Qtrain for evaluating the action for the other set of episodes.

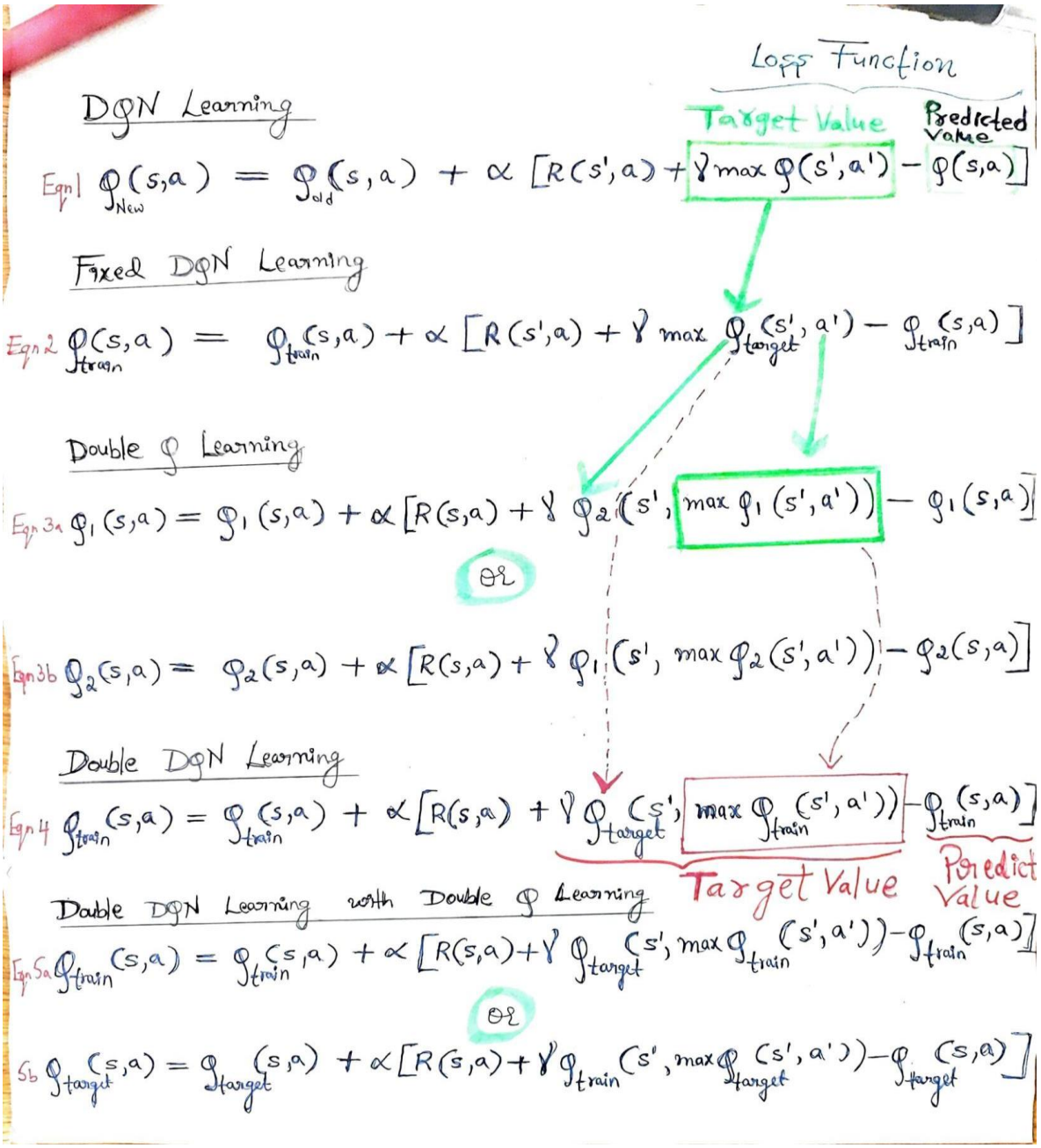

Figure 28. Workflow from DQN learning to Double DQN with Double Q learning.

9. Animation Results

The following shows the results obtained for all the 4 implementations done namely Vanilla DQN, Fixed Q-Networks DQN, Double DQN and Double DQN with Double Q Learning Coin Flip approach.
NOTE: To view the movement of the car, open the document in word.docx format.

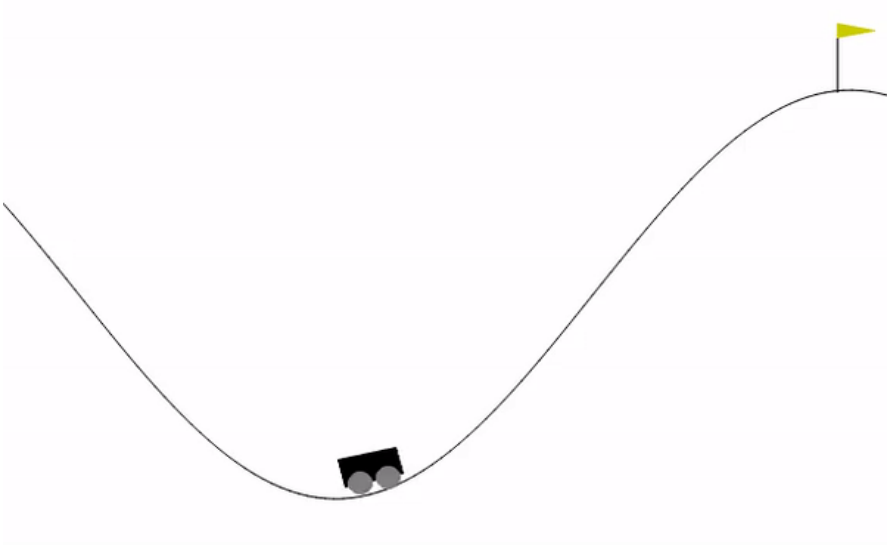

Figure 29. Agent Starting to learn (Link to video in Github)

Figure 29 illustrates the case where we have the car starting to learn to climb up the mountain, however, completes the 200 steps within that episode before it could reach the flagpole.

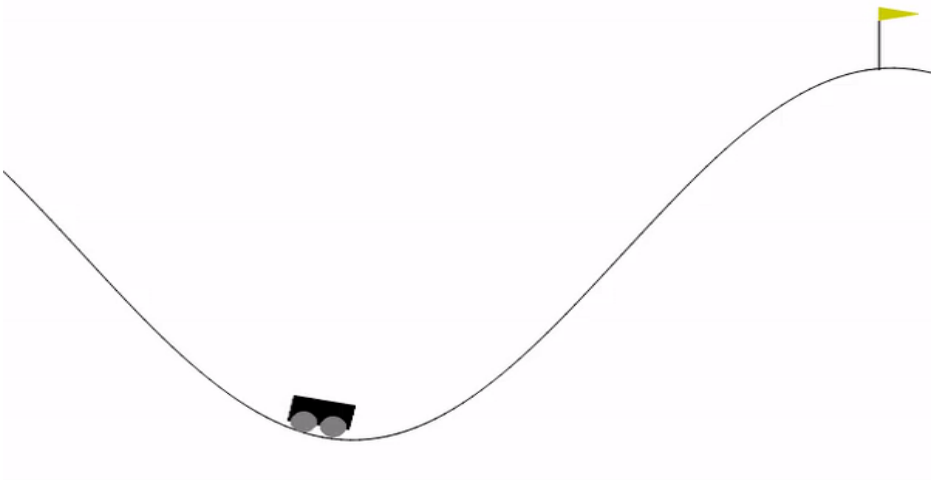

Figure 30. Agent exhibiting learning with Neural Networks(Link to video in Github)

Figure 30 shows the movement of the agent (mountain car) in terms of velocity and position. We can see climbing further up due to the learning done by the deep Q-networks; however, it still fails to reach the flagpole. However, it is evident that the agent is having a higher reward when compared to Figure 29 as it shows the car moving closer to the desired target.

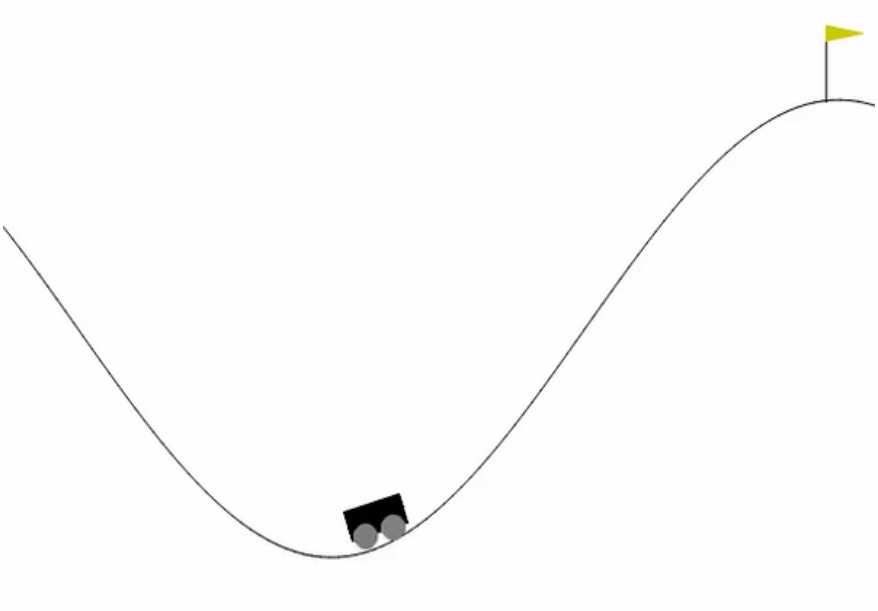

Figure 31. Agent reaching the goal within the maximum steps(Link to video in GitHub)

Figure 31 illustrates an example of the agent (mountain car) reaching its desired goal (the flagpole) within the 200 steps (maximum steps the agent can take within each episode). This means the neural networks we had used are learning to better decide the actions for the agent to get to the desired target which is, reaching the flagpole.

10. Future Work

In future, we hope to employ metrics to measure our neural network models in Deep Reinforcement Learning (DRL) model to learn and enhance the behavior of our agent. A challenge here is that we won't have access to the predicted values for the entire run time, however it can be investigated, to understand how the neural network model is behaving and to efficiently tune the hyperparameters to align the learning of the agent in the environment.

11. Bibliography

1. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). The MIT Press. P. 55 Available at : http://incompleteideas.net/book/RLbook2020trimmed.pdf [Accessed 14 March, 2024]
2. Towards Data Science. (2021). Reinforcement Learning Explained Visually: Part 5 — Deep Q Networks, Step by Step. [online] Available at:

https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b. [Accessed 19 March, 2024]
3. Manning. (2024). concept target network in category reinforcement learning. [online] Available at: https://livebook.manning.com/concept/reinforcement-learning/target-network#:~:text=By%20using%20target%20networks%2C%20we,a%20new%20one%20is%20set.[Accessed: 17 March, 2024]
4. Kaggle. (2020). Core SP20 RL Notebook. [Online]. Available at: https://www.kaggle.com/code/ucfaibot/core-sp20-rl/notebook [Accessed: 19 March, 2024].
5. Kaggle. (2020). Mountain Car is a Classic Reinforcement Learning Problem Notebook. [Online]. Available at: https://www.kaggle.com/code/rezafazel63/mountain-car-is-a-classic-reinforcement-learning/notebook [Accessed: 20 March, 2024].
6. RIS AI. (2020). Mountain Car. [Online]. Available at: https://www.ris-ai.com/mountain-car [Accessed: 19 March, 2024].
7. YMD H. (2021). Run and Render OpenAI Gym on Google Colab (Gym-Notebook-Wrapper). [Online]. Available at: https://ymd_h.gitlab.io/ymd_blog/posts/gym_on_google_colab_with_gnwrapper/ [Accessed: 19 March, 2024].
8. Github:nitish-kalan. (2023). MountainCar-v0-Deep-Q-Learning-DQN-Keras: train_model.py. [Online]. Available at: https://github.com/nitish-kalan/MountainCar-v0-Deep-Q-Learning-DQN-Keras/blob/master/train_model.py [Accessed: 25 March, 2024].
9. Github:pylSER. (2021). Deep-Reinforcement-learning-Mountain-Car: MountainCarV2.py. [Online]. Available at: https://github.com/pylSER/Deep-Reinforcement-learning-Mountain-Car/blob/master/MountainCarV2.py [Accessed: 27 March, 2024].
10. Steinbach, A. (2021). Actor-Critic Using Deep RL: Continuous Mountain Car in TensorFlow. [Online]. Available at: https://medium.com/@asteinbach/actor-critic-using-deep-rl-continuous-mountain-car-in-tensorflow-4c1fb2110f7c [Accessed: 2 April, 2024].
11. Ameet Deshpande (2018), Medium, Deep Double Q Learning: Why you should use it [Online], Available at: https://medium.com/@ameetsd97/deep-double-q-learning-why-you-should-use-it-bedf660d5295 [Accessed: April 1 2024].
12. Diego Unzueta (2022).Towards Data Science. Reinforcement Learning Applied to the Mountain Car Problem. [Online]. Available at: https://towardsdatascience.com/reinforcement-learning-applied-to-the-mountain-car-problem-1c4fb16729ba [Accessed: 5 April, 2024].
13. Adam Stelmaszczyk (2024). Deep Dive into Reinforcement Learning. [Online]. Available at: https://www.toptal.com/machine-learning/deep-dive-into-reinforcement-learning [Accessed: April 1, 2024].
14. rlpy. (2022). MountainCar.py. [Online]. Available at: https://github.com/rlpy/rlpy/blob/master/rlpy/Domains/MountainCar.py [Accessed: April 1, 2024].
15. Nikola M Zivkovic (2021), Rubikscode, Double Q Learning & Double DQN with Python and TensorFlow, [Online], Available at: https://rubikscode.net/2021/07/20/introduction-to-double-q-learning/ [Accessed: April 5, 2024].

16. Leo Simmons, Double DQN Implementation to solve OpenAI Gym's CartPole v-0, (2019), Medium, [Online], Available at: https://medium.com/@leosimmons/double-dqn-implementation-to-solve-openai-gyms-cartpole-v-0-df554cd0614d [Accessed: April 2, 2024]
17. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). The MIT Press. p. 135 Available at : http://incompleteideas.net/book/RLbook2020trimmed.pdf [Accessed 14 April, 2024]