CISC 856 – Reinforcement Learning

Final Project Report

MountainCar (Classic control)

Group 6: Eman Elrefai ID: 20104066
Hanan Omara ID: 20398559
Amal Fawzy ID: 20399126

1 – Introduction

The Mountain Car system is a simple physical system that consists of a car that must climb a steep hill to reach a goal position. The car has limited power and cannot climb the hill directly, so it must first accelerate towards one side of the hill to build up enough momentum to climb the other side.

The goal of the Mountain Car problem is to develop a control policy that allows the car to reach the goal position as quickly and efficiently as possible while obeying physical constraints such as the limited power of the vehicle and the laws of motion. This problem is interesting because it requires the agent to learn an effective and efficient control policy in a complex and dynamic environment, without any prior knowledge of the system or the task.

In this project, we aim to solve the Mountain Car problem using Q Learning and DQN. We hope to gain insights into the different trials and compare between them.

2 – Problem Formulation

The Mountain Car problem can be formulated as a Markov Decision Process (MDP), which consists of a set of states, actions, transition probabilities, and rewards. In this problem, the state space consists of the position and velocity of the car, which are continuous variables. The action space consists of two discrete actions: accelerate to the left or accelerate to the right. The transition probabilities are determined by the laws of motion and the physical constraints of the system. The reward scheme is designed to encourage the car to reach the goal position as quickly as possible, while penalizing the car for using excessive power and taking too long to reach the goal.

State Space

The agent's state space in the MountainCar task consists of two continuous variables:

1) Position: The position represents the current position of the car along the horizontal axis.

2) Velocity: The velocity represents the current velocity of the car.

The position and velocity values can take any real number, allowing for a continuous representation of the state.

Action Space

The agent's action space in the MountainCar task consists of three available actions that the agent can choose from. These actions are discrete and represent the possible actions the agent can take at each time step. The three actions are:

1) Throttle Forward: This action allows the car to apply a forward throttle, which means accelerating in the positive direction along the horizontal axis.

2) Throttle Backward: This action allows the car to apply a backward throttle, which means accelerating in the negative direction along the horizontal axis.

3) Zero Throttle: This action represents no throttle, where the car maintains its current velocity without any acceleration.

Reward Scheme

The reward scheme for the Mountain Car problem is designed to encourage the car to reach the goal position as quickly as possible, while penalizing the car for using excessive power and taking too long to reach the goal. Specifically, we can define the reward function as follows:

If the car reaches the goal position (position >= 0.5), the reward is 100. If the car does not reach the goal position, the reward is -1 for each time step. If the car uses excessive power (|action| > 0), the reward is -0.1 for each time step. If the car takes too long to reach the goal position (more than 200 steps), the reward is -1.

3 – Solution Overview

In this project, we aim to solve the Mountain Car problem using Q-learning and DQN.

We implemented both **Q-learning** and **DQN approaches** using Python and the Gym library, which provides a Python interface to the Mountain Car problem. We used the NumPy library for numerical computations and the TensorFlow library for deep learning. To evaluate the performance of these approaches, we trained them on the Mountain Car problem with different initial conditions and track configurations, and compared their performance in terms of the average number of time steps to reach the goal position and the average total reward per episode. We also analyzed the learned Q-functions and policies and compared their behavior in different parts of the state space.

We used the standard OpenAI Gym environment for the Mountain Car problem to simulate the system and evaluate the performance of our control approaches. The Gym environment provides a convenient and flexible platform for testing and comparing different control algorithms on the same problem domain.

Q-learning

Q-learning is a model-free reinforcement learning algorithm that learns an optimal Q-function, which estimates the expected cumulative reward for each state-action pair. The Q-function can be updated iteratively using the Bellman equation, which expresses the expected value of the Q-function in terms of the expected value of the next state-action pair. The Q-learning algorithm typically uses an epsilon-greedy policy, which chooses a random action with probability epsilon and the action with the highest Q-value with probability 1-epsilon. To apply Q-learning to the Mountain Car problem, we define a discrete state space and action space based on the position and velocity of the car. We then initialize the Q-function randomly and update it iteratively using the Bellman equation with a learning rate and a discount factor. We use an epsilon-greedy policy to choose actions during training, and a greedy policy to choose actions during testing.

Implementation Approach:

- This implementation of the Q-learning algorithm trains an agent to find an optimal policy for a given environment with a discrete action space. The agent learns to balance exploration and exploitation and uses the Bellman equation to update its Q-values over time.
- **The MountainCar-v1 environment using the Q-learning algorithm**
  - The MountainCar-v1 environment is registered, and the number of actions and observation space bins are set.
  - A QLearning object is initialized with the MountainCar-v1 environment and the specified parameters.
  - The agent is trained using the train() method of the QLearning object. The method trains the agent for the specified number of episodes and returns the average timesteps per episode.
  - The agent's Q-values are updated using the Bellman equation, and the exploration rate and learning rate are decayed over time. The test_timesteps list is used to store the test timesteps obtained during testing.

Then we start our training using the training function we built.In the following part we will present our trials:

Trial 1:

- Learning Rate (ALPHA_BASE): 0.05
- Discount Factor (GAMMA): 0.99
- Exploration Rate (EXPLORATION_RATE_BASE): 0.6

Trial 2:

- Learning Rate (ALPHA_BASE): 0.3
- Discount Factor (GAMMA): 0.99
- Exploration Rate (EXPLORATION_RATE_BASE): 0.8

Trial 3:

- Learning Rate (ALPHA_BASE): 0.5
- Discount Factor (GAMMA): 0.99
- Exploration Rate (EXPLORATION_RATE_BASE): 0.2

In the result part we will present our trials performance and the best trial we got .

DQN approach

The DQN approach is a variant of Q-learning that uses a neural network to estimate the Q-function instead of a look-up table. The neural network takes the state as input and outputs the Q-value for each possible action. The DQN algorithm uses experience replay to store and sample transitions from a replay buffer, which improves the stability and convergence of the algorithm. The DQN algorithm also uses a target network to stabilize the Q-value estimates during training, by periodically updating the target network with the weights of the online network. To apply DQN to the Mountain Car problem, we use a convolutional neural network to process the state input, followed by two fully connected layers to estimate the Q-values for each action. We use experience replay and a target network to train the network, and a greedy policy to choose actions during testing.

In this part we apply two different methodology

1. **First Methodology using DQN Learning(the original approach)**

   In this methodology we apply two trial:

Trial 1:

- The neural network model in Trial 1 has two hidden layers with 48 and 64 units respectively, both using LeakyReLU activation.
- The learning rate is set to 0.0001, and the optimizer used is Adam.
- The discount factor is 0.95, and the epsilon value starts at 1 and decays exponentially over time.
- The maximum size of the agent's memory is set to 2000.
- The agent uses experience replay by randomly sampling a minibatch from memory for training.
- The target model is updated periodically to match the current model's weights.
- The agent uses the epsilon-greedy policy for exploration, with a random action taken with probability epsilon.
- The agent trains the model using the mean squared error loss.

Trial 2:

- The neural network model in Trial 2 has three hidden layers with 64, 128, and 64 units respectively, all using LeakyReLU activation.
- The learning rate is set to 0.0005, and the optimizer used is Adam.
- The discount factor is 0.99, and the epsilon value starts at 1 and decays exponentially over time.
- The maximum size of the agent's memory is set to 2000.
- The agent uses experience replay by randomly sampling a minibatch from memory for training.
- The target model is updated periodically to match the current model's weights.
- The agent uses the epsilon-greedy policy for exploration, with a random action taken with probability epsilon.
- The agent trains the model using the mean squared error loss.

2- Second Methodology of using Deep Q-Learning

Shallow learning algorithms can also be used in reinforcement learning, which is a type of machine learning that involves training an agent to make decisions based on rewards and punishments received from the environment.

In reinforcement learning, the agent interacts with the environment and receives feedback in the form of rewards or penalties based on its actions. The goal of the agent is to learn a policy, which is a mapping from states to actions, that maximizes the cumulative reward over time.

Shallow learning algorithms can be used to learn the policy of the agent. Here are some common terms used in reinforcement learning:

1. Q-learning: A popular reinforcement learning algorithm that uses a table to store the expected reward for each state-action pair.
2. Deep Q-network (DQN): A variant of Q-learning that uses a deep neural network to approximate the Q-values.
3. Policy gradient: A reinforcement learning algorithm that directly optimizes the policy using gradient descent.
4. Actor-critic: A reinforcement learning algorithm that uses two networks: an actor network that learns the policy and a critic network that learns to estimate the value function.
5. Exploration-exploitation tradeoff: The balance between taking actions that are known to be good (exploitation) and exploring new actions to discover potentially better ones (exploration).

Shallow learning algorithms can be used to learn the policy of the agent in reinforcement learning by approximating the Q-values or the policy directly. However, these algorithms may struggle with high-dimensional state spaces or complex environments. In these cases, deep reinforcement learning algorithms, which use deep neural networks to represent the policy or the value function, may be more suitable.

In this methodology we apply two trial:

Trial 1:

- The agent in Trial 1 is trained using a policy implemented in the Policy class.
- The agent is initialized with a MountainCarAgent object, with specified parameters for discount factor, learning rate, epsilon, and epsilon decay.
- The loss function used is mean squared error (MSE).
- The optimizer used is Adam optimizer with a learning rate of 0.001.
- The train function is called to train the agent for 1000 episodes.

Trial 2:

- The agent in Trial 2 is trained using a different policy implemented in the Policy2 class.
- The agent is initialized with a MountainCarAgent object, with the same specified parameters as in Trial 1.
- The loss function used also means squared error (MSE).
- The optimizer used is Adam optimizer with a learning rate of 0.001.
- The train function is called to train the agent for 1000 episodes.

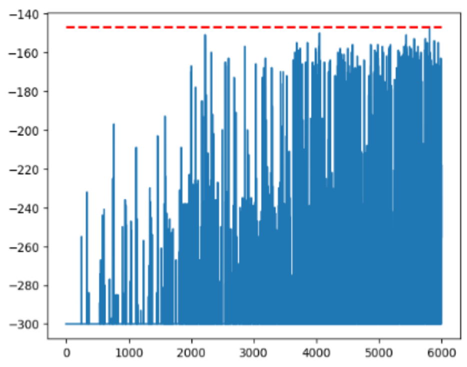
4 – Results

First, the result of Q-Learning

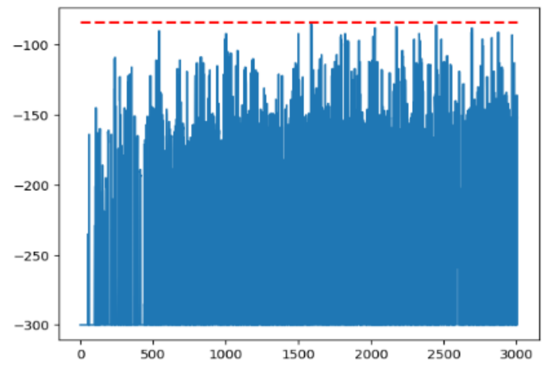
- ❖ **Trial 1**

ALPHA = 0.05 && EXPLORATION_RATE = 0.6

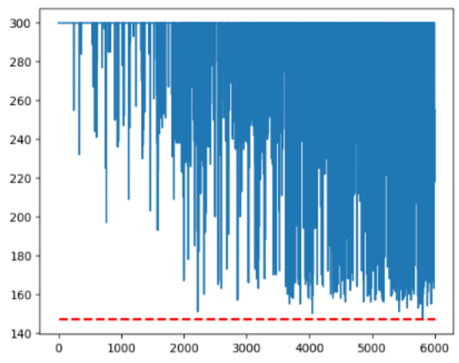
The performance of training and testing (plots)



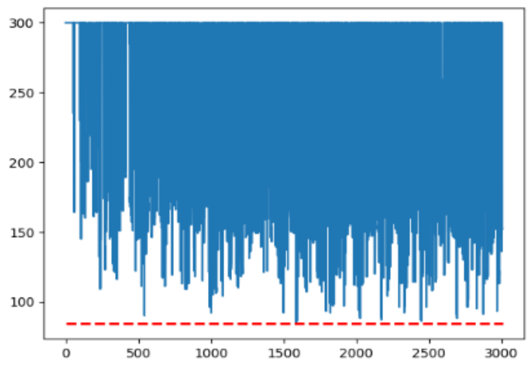
Rewards over episodes during training



Rewards over episodes during test



Timesteps over episodes during training



Timesteps over episodes during training

Training **Testing**

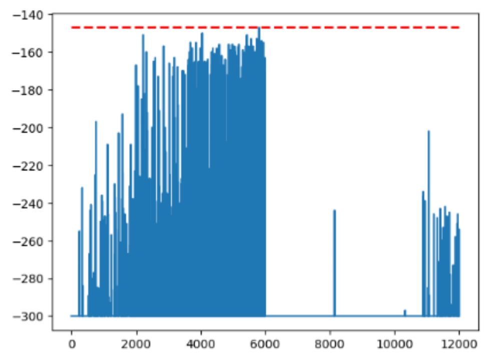
Trial 1 observation:

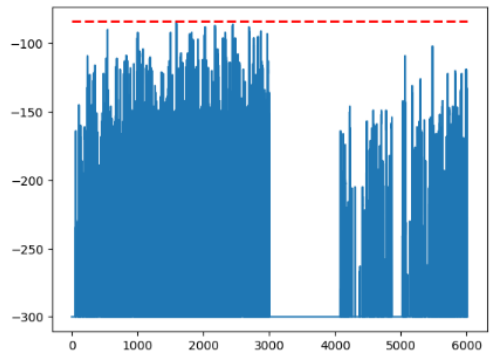
- The maximum reward achieved during training is -147.
- The average reward during training is -285.67 with a standard deviation of 32.74.
- The maximum reward achieved during testing is -84.
- The average reward during testing is -231.14 with a standard deviation of 73.23.

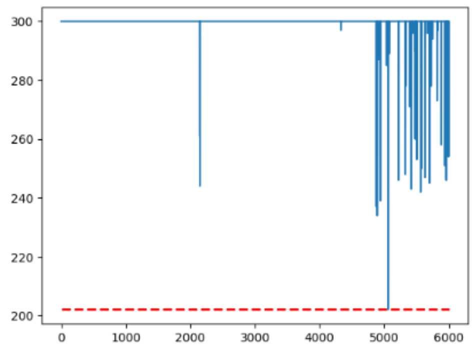
- **Trial 2**

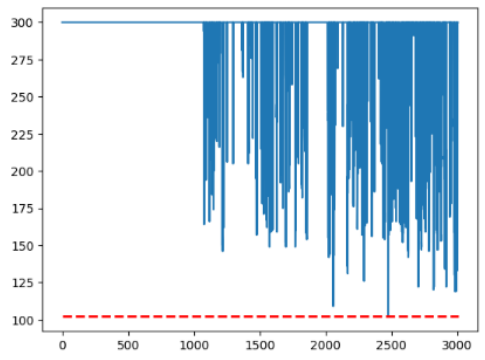
ALPHA = 0.3 && EXPLORATION_RATE = 0.8

The performance of training and testing (plots)


Rewards over episodes during training


Rewards over episodes during test


Timesteps over episodes during training


Timesteps over episodes during training

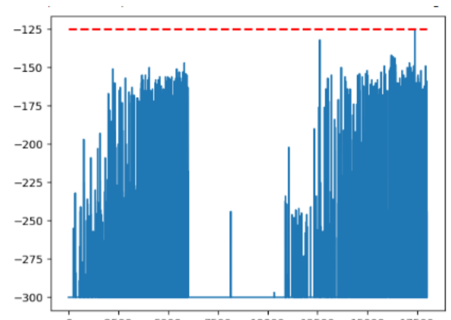
Trial 2 observation:

- ❖ The maximum reward achieved during training is -147.
- ❖ The average reward during training is -292.72 with a standard deviation of 24.32.
- ❖ The maximum reward achieved during testing is -84.
- ❖ The average reward during testing is -257.86 with a standard deviation of 64.78.

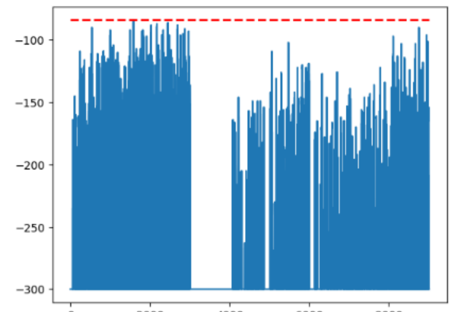
- ❖ **Trial 3**

ALPHA = 0.5 && EXPLORATION_RATE = 0.2

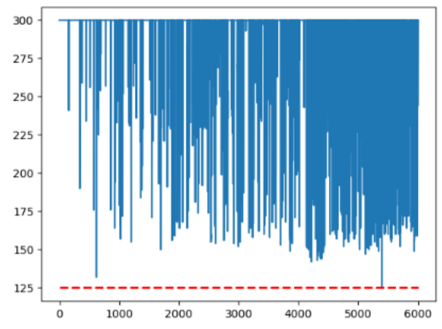
The performance of training and testing (plots)



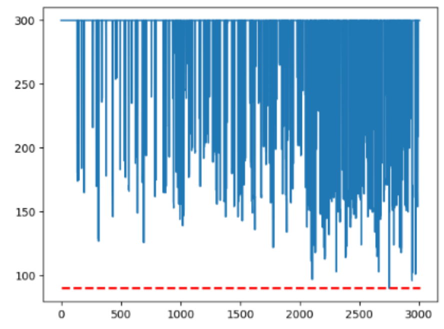
Rewards over episodes during training



Rewards over episodes during test



Timesteps over episodes during training



Timesteps over episodes during training

Trial 3 observation:

- ❖ The maximum reward achieved during training is -125.
- ❖ The average reward during training is -290.77 with a standard deviation of 28.10.

- ❖ The maximum reward achieved during testing is -84.
- ❖ The average reward during testing is -262.98 with a standard deviation of 61.26**.**

Determination of the Best Trial:

Considering both the average rewards and standard deviations during testing, we can determine the best trial as follows:

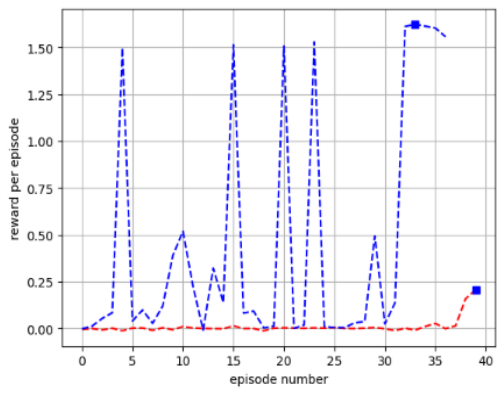
- Trial 1 has the highest average reward during testing (-231.14) among the three trials, with a relatively higher standard deviation (73.23).
- Trial 2 has a slightly lower average reward during testing (-257.86) compared to Trial 1, but a lower standard deviation (64.78).
- Trial 3 has the lowest average reward during testing (-262.98), but a moderate standard deviation (61.26).
- Based on these observations, Trial 2 demonstrates a relatively better performance with a lower standard deviation during testing. Therefore, considering both the average reward and stability of results, Trial 2 can be considered as the best-performing trial among the three.

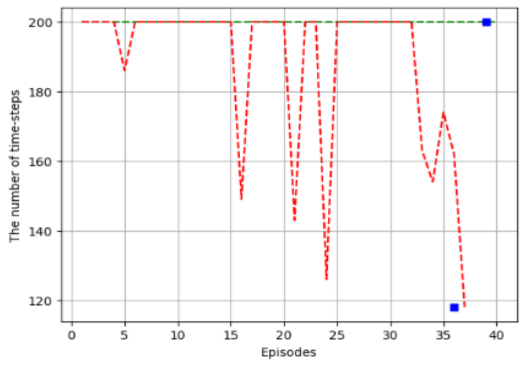
Second, the result of DQN

1. First Methodology using DQN Learning(the original approach)

In this methodology we apply two trial:

Compare performance of the Two trials:





Observations with respect to the results obtained

When comparing the average rewards of the last 100 episodes (mean) for Trial 1 and Trial 2, we observe the following:

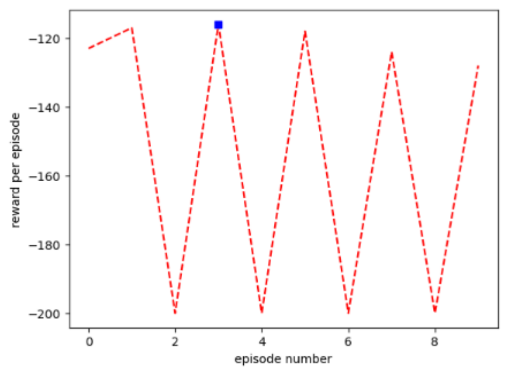
- Trial 1 has an average reward of approximately 0.0093 for the last 100 episodes.
- Trial 2 has a significantly higher average reward of approximately 0.4598 for the last 100 episodes.

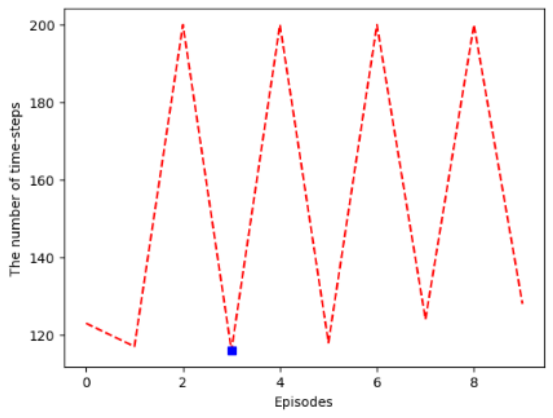
the optimal solution is the Second trial

Reason: This suggests that Trial 2, which has a deeper neural network architecture, a higher learning rate, a higher discount factor, a slower epsilon decay rate, and a lower epsilon minimum value, performs better than Trial 1 in terms of achieving higher cumulative rewards. The additional hidden layers and increased complexity of the model in Trial 2 seem to have a positive impact on the agent's performance, allowing it to learn more effectively and achieve higher rewards in the environment.

Test The Agent(Optimal solution = Trial 2)

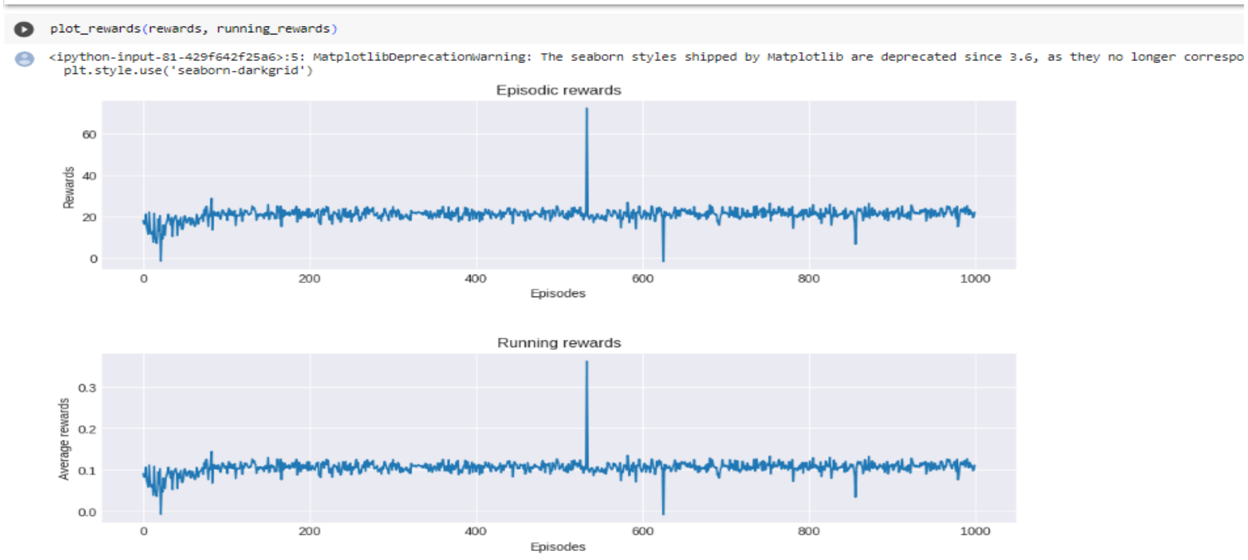
Best performance with the Second hyperparameters(model_weights_2.h5)





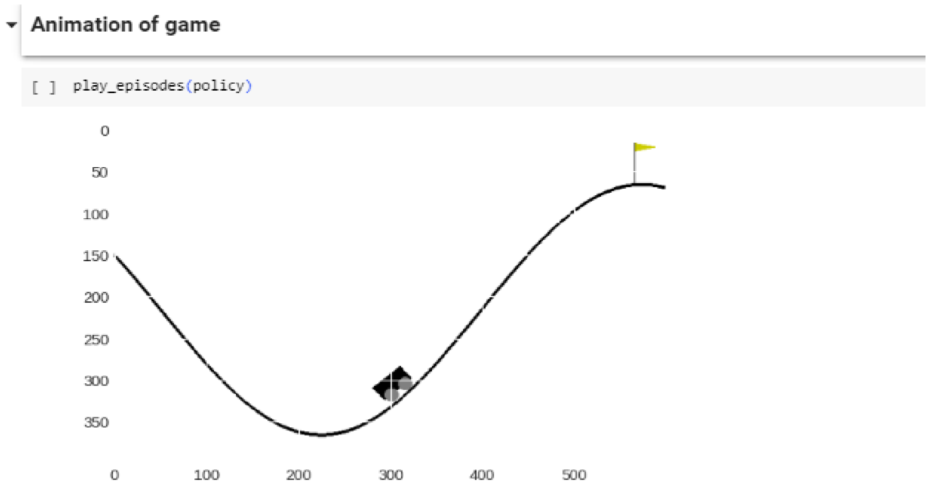
2- Second Methodology of using Deep Q-Learning (Shallow learning)

Trial 1



```
plot_rewards(rewards, running_rewards)
```

```
<ipython-input-81-429f642f25a6>:5: MatplotlibDeprecationWarning: The seaborn styles shipped by Matplotlib are deprecated since 3.6, as they no longer correspo
  plt.style.use('seaborn-darkgrid')
```



Animation of game

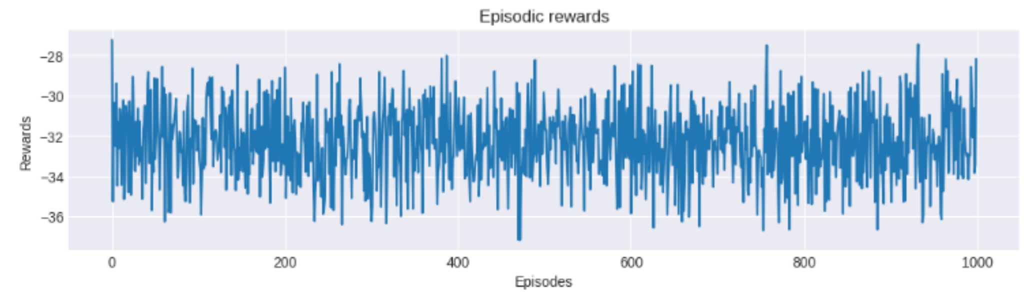
```
[ ]  play_episodes(policy)
```

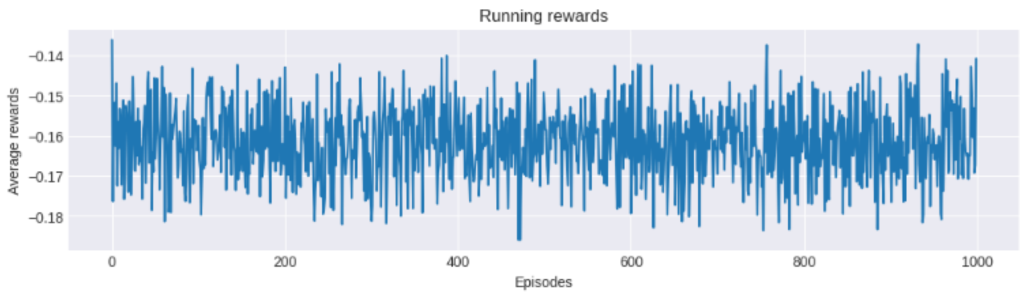
Trial 2

Plot the Rewards

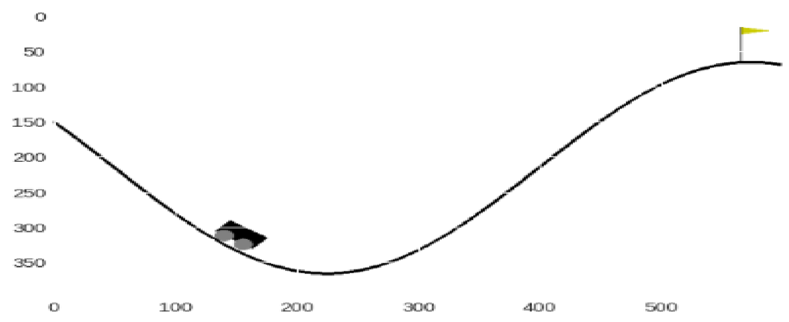
```
plot_rewards(rewards, running_rewards)
```

```
<ipython-input-81-429f642f25a6>:5: MatplotlibDeprecationWarning: The seaborn styles shipped by Matplotlib are deprecated since 3.6, as they no longer co
  plt.style.use('seaborn-darkgrid')
```





```
play_episodes(policy2)
```



5 – Conclusion

At the end from our work on this different techniques to learn our agent we found that the DQN was better than QLearning and no need to run more than 50 epoch to get a satisfying result. What we really need to work on is the hyperparameter tuning part that we expect that it will make more better results , but in our case the resources and time wear a big problem for us .