FLAPPY BIRD

TEAM MEMBERS:
MARIM AMER
MAWADA AMER
YOUMNA ALSAYED

Contents

Introduction

We will apply to the Flappy Bird game a Reinforcement Learning model that learns control policies directly from pipes and from feedback received when the bird hits the pipes or the ground. We will implement a DQN algorithm that uses a deep learning model to find the actions an agent can take at each state in a variant of DQN-Learning. We teach a Flappy Bird agent how to fly and go through pipes. If the Flappy Bird passes any pipe, the agent will receive a reward = +1; if the Flappy Bird dies, the reward will be zero. We will use the flappy-bird-gym library to use a flappy bird environment, but we will implement the DQN algorithm from scratch.

DQN (Deep-Q Networks)

DQN is a combination between Deep Learning and Reinforcement Learning. It combines the benefits of deep learning with the benefits of reinforcement learning (RL). Reinforcement learning focuses on teaching agents to take any action in a given environment at a specific stage to maximize rewards. By sensing rewards through interactions with the environment, reinforcement learning seeks to teach the model to improve itself and its choices.

DQN uses four approaches to solve unstable learning:

- Experience Replay
- Target Network
- Clipping Rewards
- Skipping Frames

Experience Replay

Experience Replay takes state transitions, rewards, and actions to perform DQ learning, and uses mini-batches to update neural networks.

The advantages of this approach are that it reduces the correlation between experiences when updating a deep neural network, speeds up learning with mini-batches, and reuses previous transitions to minimize catastrophic forgetting.

Target Network

An unstable target function makes training difficult, so for every specific step, the Target Network approach updates the target function's parameters and replaces them with the most recent network.

Clipping Rewards

Each game has a different scoring scale. To solve this problem, it uses the Clipping Rewards technique to clip scores, where all positive rewards are set at +1 and all negative rewards are set at -1.

Skipping Frames

The Skipping Frames technique reduces the computational cost and saves all experience by calculating Q values every n frame and using the previous n frames as inputs.

Code

Create environment

```python
import flappy_bird_gym
env = flappy_bird_gym.make("FlappyBird-v0")
env.seed(0)
print('State shape: ', env.observation_space.shape)
print('Number of actions: ', env.action_space.n)
```

```
State shape:  (2,)
Number of actions:  2
```

Create model class

DNN contains 3 layers (2 relu and 1 linear). We will use a non-linear function to map from state to action.

```python
class Create_model(nn.Module):
    """Create NN (Policy) Model"""

    def __init__(self, num_state, num_action, seed=1000):
        """
            num_state: Number of state
            num_action: Number of action
            seed: Generate random numbers
        """

        super(Create_model, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(num_state, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, num_action)

    def forward(self, state):
        """Create a network state -> action"""
        x = self.fc1(state)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        return self.fc3(x)
```

Experience Memory

Experience memory is used to save all experiences.

```
class Experience_memory:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, num_action, size_of_buffer, batch_size, seed=1000):
        """
            num_action: number of action
            size_of_buffer: maximum size of buffer
            batch_size: size of training batch
            seed: Generate random numbers
        """
        self.num_action = num_action
        self.memory = deque(maxlen=size_of_buffer)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def collect(self, state, action, reward, next_state, done):
        """Append new experience to memory"""
        exp = self.experience(state, action, reward, next_state, done)
        self.memory.append(exp)

    def sample(self):
        """Taking a random sample of memories from a large collection"""
        samples = random.sample(self.memory, k=self.batch_size)
        states = torch.from_numpy(np.vstack([s.state for s in samples if s is not None])).float()
        actions = torch.from_numpy(np.vstack([s.action for s in samples if s is not None])).long()
        rewards = torch.from_numpy(np.vstack([s.reward for s in samples if s is not None])).float()
        next_states = torch.from_numpy(np.vstack([s.next_state for s in samples if s is not None])).float()
        dones = torch.from_numpy(np.vstack([s.done for s in samples if s is not None]).astype(np.uint8)).float()

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of memory."""
        return len(self.memory)
```

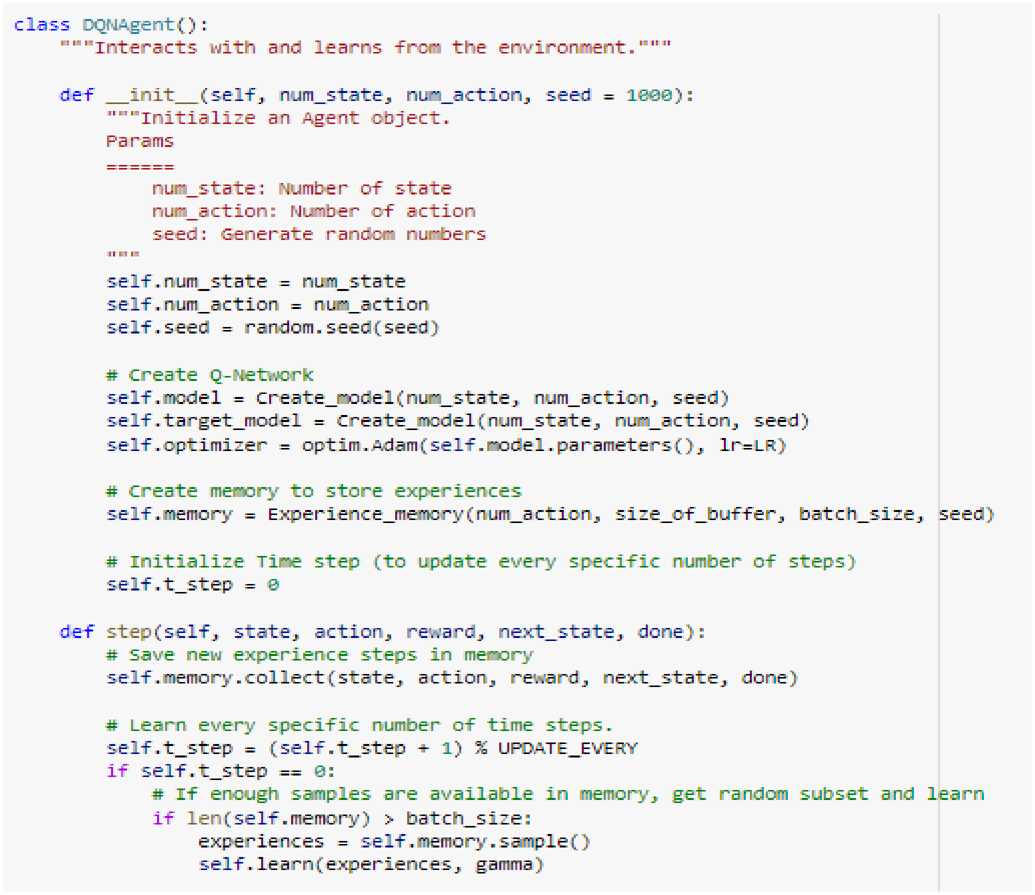
Initialize parameters

```
size_of_buffer = int(100000)  # buffer size
LR = 0.0005                   # learning rate
UPDATE_EVERY = 5              # how often to update the network
gamma = 0.999                 # discount factor
tau = 0.001                   # for soft update of target parameters
batch_size = 64               # minibatch size
```

DQN Agent

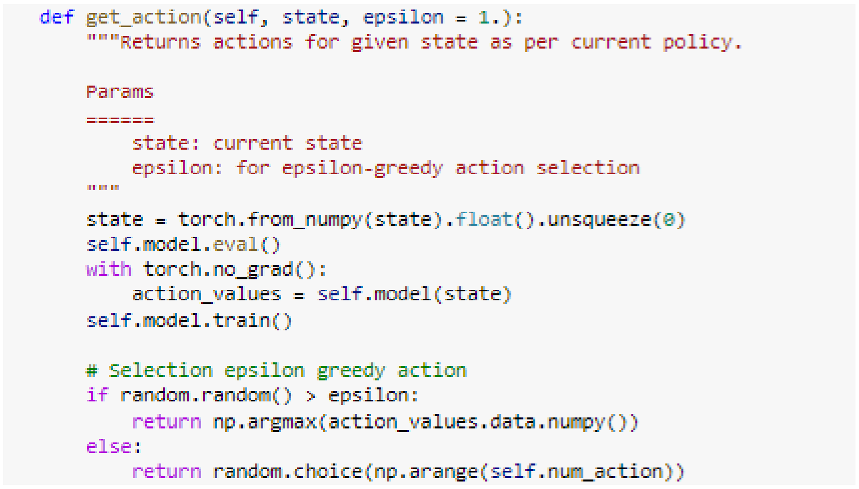
The agent has 4 functions:

1- step function that saves new experience steps in memory and learns every specific number of time steps.



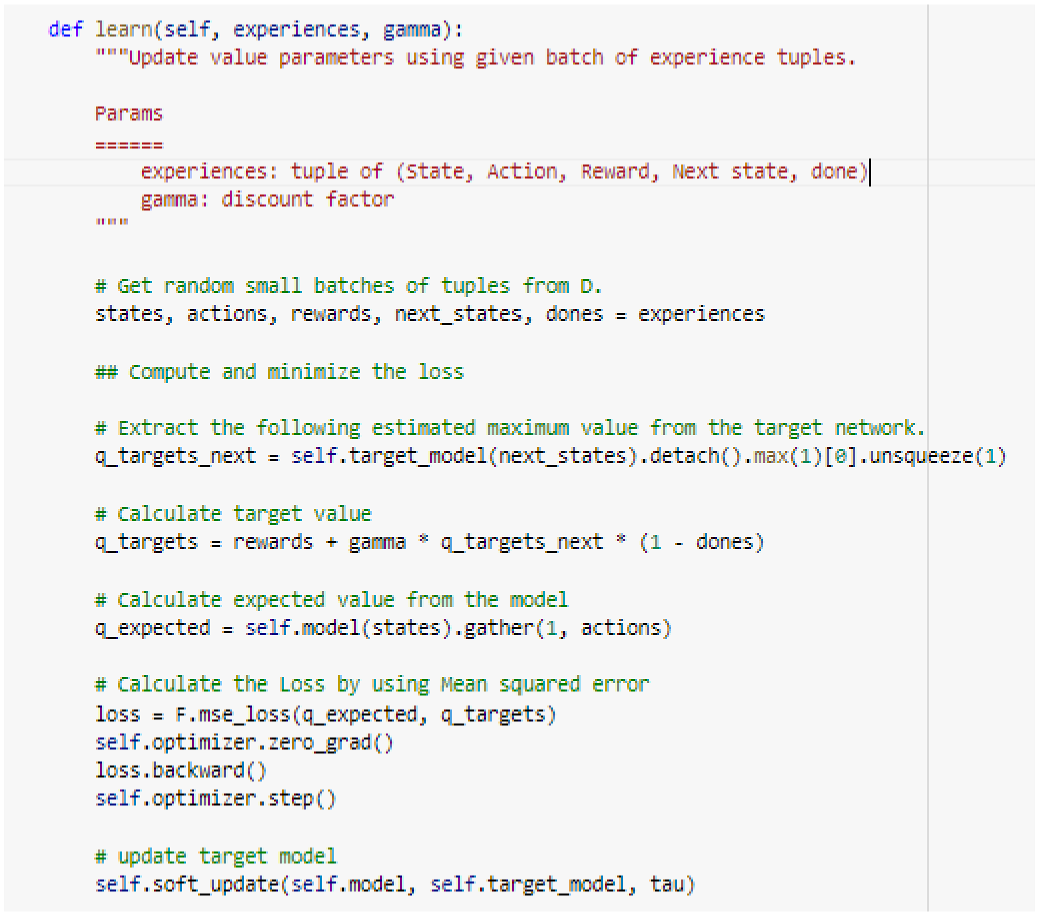
```python
class DQNAgent():
    """Interacts with and learns from the environment."""

    def __init__(self, num_state, num_action, seed = 1000):
        """Initialize an Agent object.
        Params
        ======
            num_state: Number of state
            num_action: Number of action
            seed: Generate random numbers
        """
        self.num_state = num_state
        self.num_action = num_action
        self.seed = random.seed(seed)

        # Create Q-Network
        self.model = Create_model(num_state, num_action, seed)
        self.target_model = Create_model(num_state, num_action, seed)
        self.optimizer = optim.Adam(self.model.parameters(), lr=LR)

        # Create memory to store experiences
        self.memory = Experience_memory(num_action, size_of_buffer, batch_size, seed)

        # Initialize Time step (to update every specific number of steps)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        # Save new experience steps in memory
        self.memory.collect(state, action, reward, next_state, done)

        # Learn every specific number of time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and learn
            if len(self.memory) > batch_size:
                experiences = self.memory.sample()
                self.learn(experiences, gamma)
```

2- get_action function that returns actions for a given state as per current policy by using both exploitations and explorations are used by the agent to take advantage of prior information and find new choices.



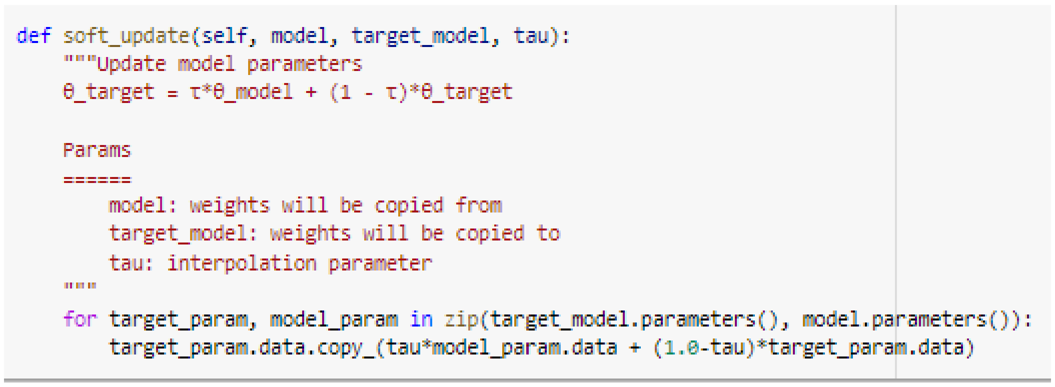
```python
    def get_action(self, state, epsilon = 1.):
        """Returns actions for given state as per current policy.

        Params
        ======
            state: current state
            epsilon: for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0)
        self.model.eval()
        with torch.no_grad():
            action_values = self.model(state)
        self.model.train()

        # Selection epsilon greedy action
        if random.random() > epsilon:
            return np.argmax(action_values.data.numpy())
        else:
            return random.choice(np.arange(self.num_action))
```

3- learn a function that updates value parameters (Q) using a given batch of experience.



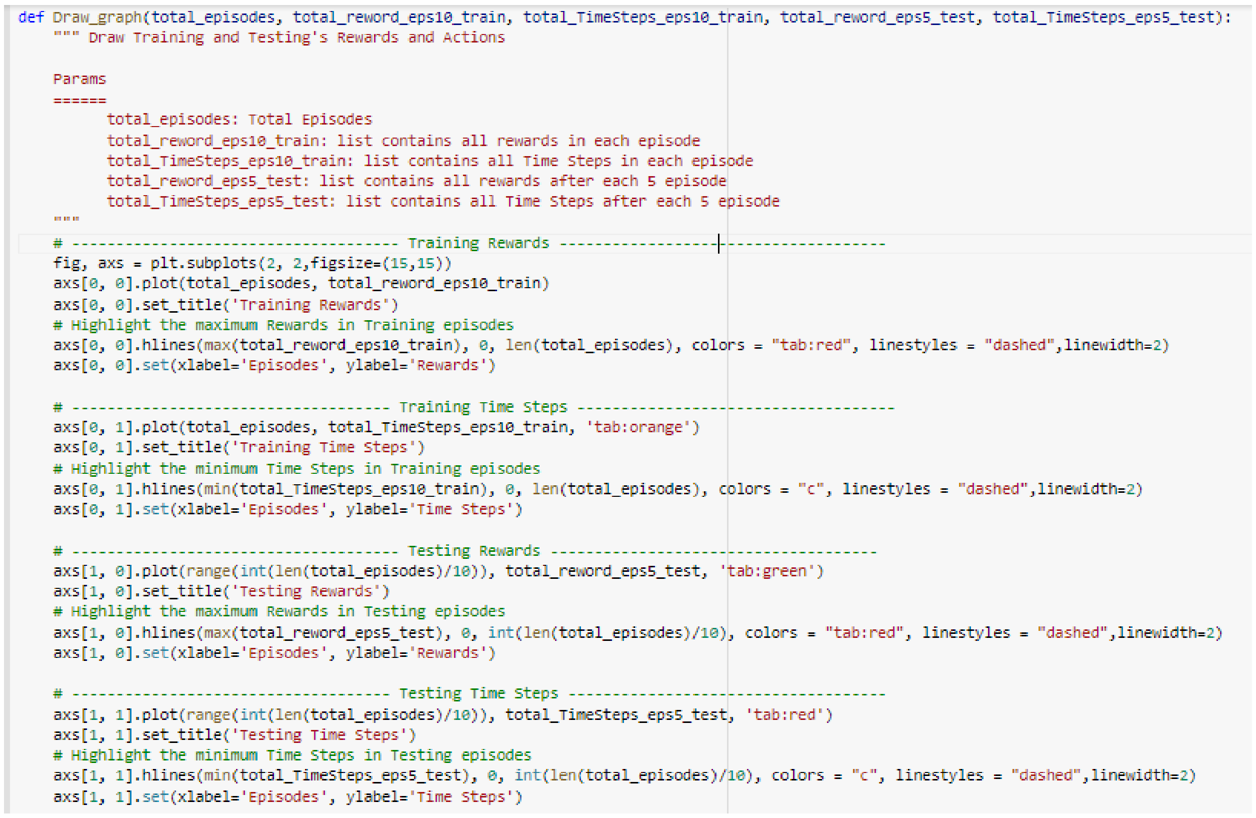
```
def learn(self, experiences, gamma):
    """Update value parameters using given batch of experience tuples.

    Params
    ======
        experiences: tuple of (State, Action, Reward, Next state, done)
        gamma: discount factor
    """

    # Get random small batches of tuples from D.
    states, actions, rewards, next_states, dones = experiences

    ## Compute and minimize the loss

    # Extract the following estimated maximum value from the target network.
    q_targets_next = self.target_model(next_states).detach().max(1)[0].unsqueeze(1)

    # Calculate target value
    q_targets = rewards + gamma * q_targets_next * (1 - dones)

    # Calculate expected value from the model
    q_expected = self.model(states).gather(1, actions)

    # Calculate the Loss by using Mean squared error
    loss = F.mse_loss(q_expected, q_targets)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # update target model
    self.soft_update(self.model, self.target_model, tau)
```

4- soft_update that Update model parameter (weights)



```
def soft_update(self, model, target_model, tau):
    """Update model parameters
    θ_target = τ*θ_model + (1 - τ)*θ_target

    Params
    ======
        model: weights will be copied from
        target_model: weights will be copied to
        tau: interpolation parameter
    """
    for target_param, model_param in zip(target_model.parameters(), model.parameters()):
        target_param.data.copy_(tau*model_param.data + (1.0-tau)*target_param.data)
```

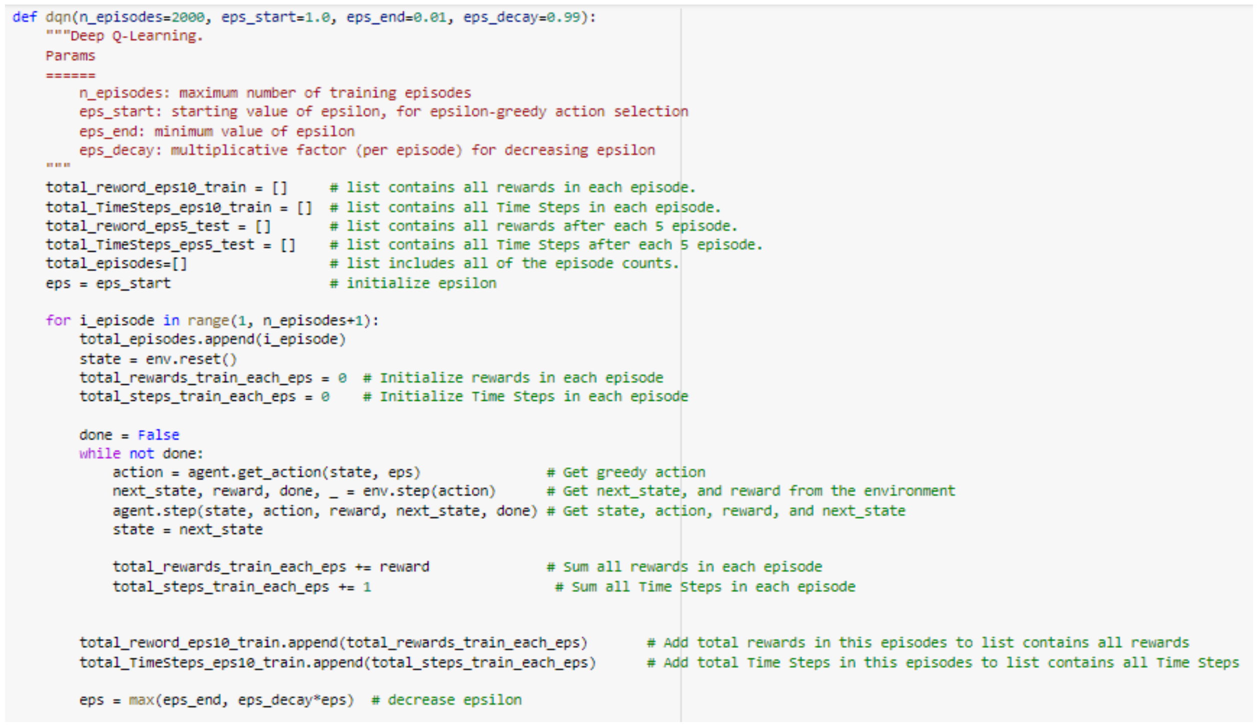
Draw function



```
def Draw_graph(total_episodes, total_reword_eps10_train, total_TimeSteps_eps10_train, total_reword_eps5_test, total_TimeSteps_eps5_test):
    """ Draw Training and Testing's Rewards and Actions

    Params
    ======
        total_episodes: Total Episodes
        total_reword_eps10_train: list contains all rewards in each episode
        total_TimeSteps_eps10_train: list contains all Time Steps in each episode
        total_reword_eps5_test: list contains all rewards after each 5 episode
        total_TimeSteps_eps5_test: list contains all Time Steps after each 5 episode
    """
    # ------------------------------------ Training Rewards ------------------------------------
    fig, axs = plt.subplots(2, 2,figsize=(15,15))
    axs[0, 0].plot(total_episodes, total_reword_eps10_train)
    axs[0, 0].set_title('Training Rewards')
    # Highlight the maximum Rewards in Training episodes
    axs[0, 0].hlines(max(total_reword_eps10_train), 0, len(total_episodes), colors = "tab:red", linestyles = "dashed",linewidth=2)
    axs[0, 0].set(xlabel='Episodes', ylabel='Rewards')

    # ------------------------------------ Training Time Steps ------------------------------------
    axs[0, 1].plot(total_episodes, total_TimeSteps_eps10_train, 'tab:orange')
    axs[0, 1].set_title('Training Time Steps')
    # Highlight the minimum Time Steps in Training episodes
    axs[0, 1].hlines(min(total_TimeSteps_eps10_train), 0, len(total_episodes), colors = "c", linestyles = "dashed",linewidth=2)
    axs[0, 1].set(xlabel='Episodes', ylabel='Time Steps')

    # ------------------------------------ Testing Rewards ------------------------------------
    axs[1, 0].plot(range(int(len(total_episodes)/10)), total_reword_eps5_test, 'tab:green')
    axs[1, 0].set_title('Testing Rewards')
    # Highlight the maximum Rewards in Testing episodes
    axs[1, 0].hlines(max(total_reword_eps5_test), 0, int(len(total_episodes)/10), colors = "tab:red", linestyles = "dashed",linewidth=2)
    axs[1, 0].set(xlabel='Episodes', ylabel='Rewards')

    # ------------------------------------ Testing Time Steps ------------------------------------
    axs[1, 1].plot(range(int(len(total_episodes)/10)), total_TimeSteps_eps5_test, 'tab:red')
    axs[1, 1].set_title('Testing Time Steps')
    # Highlight the minimum Time Steps in Testing episodes
    axs[1, 1].hlines(min(total_TimeSteps_eps5_test), 0, int(len(total_episodes)/10), colors = "c", linestyles = "dashed",linewidth=2)
    axs[1, 1].set(xlabel='Episodes', ylabel='Time Steps')
```

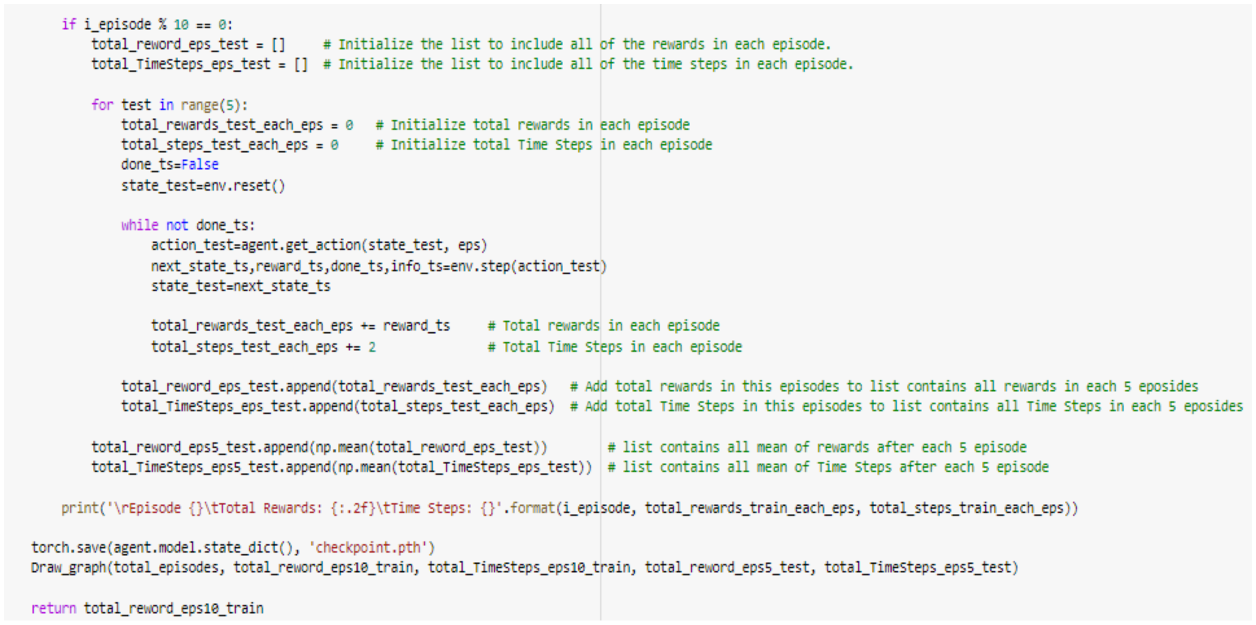
Train and Testing

1- Training



```
def dqn(n_episodes=2000, eps_start=1.0, eps_end=0.01, eps_decay=0.99):
    """Deep Q-Learning.
    Params
    ======
        n_episodes: maximum number of training episodes
        eps_start: starting value of epsilon, for epsilon-greedy action selection
        eps_end: minimum value of epsilon
        eps_decay: multiplicative factor (per episode) for decreasing epsilon
    """
    total_reword_eps10_train = []     # list contains all rewards in each episode.
    total_TimeSteps_eps10_train = []  # list contains all Time Steps in each episode.
    total_reword_eps5_test = []       # list contains all rewards after each 5 episode.
    total_TimeSteps_eps5_test = []    # list contains all Time Steps after each 5 episode.
    total_episodes=[]                 # list includes all of the episode counts.
    eps = eps_start                   # initialize epsilon

    for i_episode in range(1, n_episodes+1):
        total_episodes.append(i_episode)
        state = env.reset()
        total_rewards_train_each_eps = 0  # Initialize rewards in each episode
        total_steps_train_each_eps = 0    # Initialize Time Steps in each episode

        done = False
        while not done:
            action = agent.get_action(state, eps)            # Get greedy action
            next_state, reward, done, _ = env.step(action)   # Get next_state, and reward from the environment
            agent.step(state, action, reward, next_state, done) # Get state, action, reward, and next_state
            state = next_state

            total_rewards_train_each_eps += reward           # Sum all rewards in each episode
            total_steps_train_each_eps += 1                   # Sum all Time Steps in each episode


        total_reword_eps10_train.append(total_rewards_train_each_eps)      # Add total rewards in this episodes to list contains all rewards
        total_TimeSteps_eps10_train.append(total_steps_train_each_eps)     # Add total Time Steps in this episodes to list contains all Time Steps

        eps = max(eps_end, eps_decay*eps)  # decrease epsilon
```

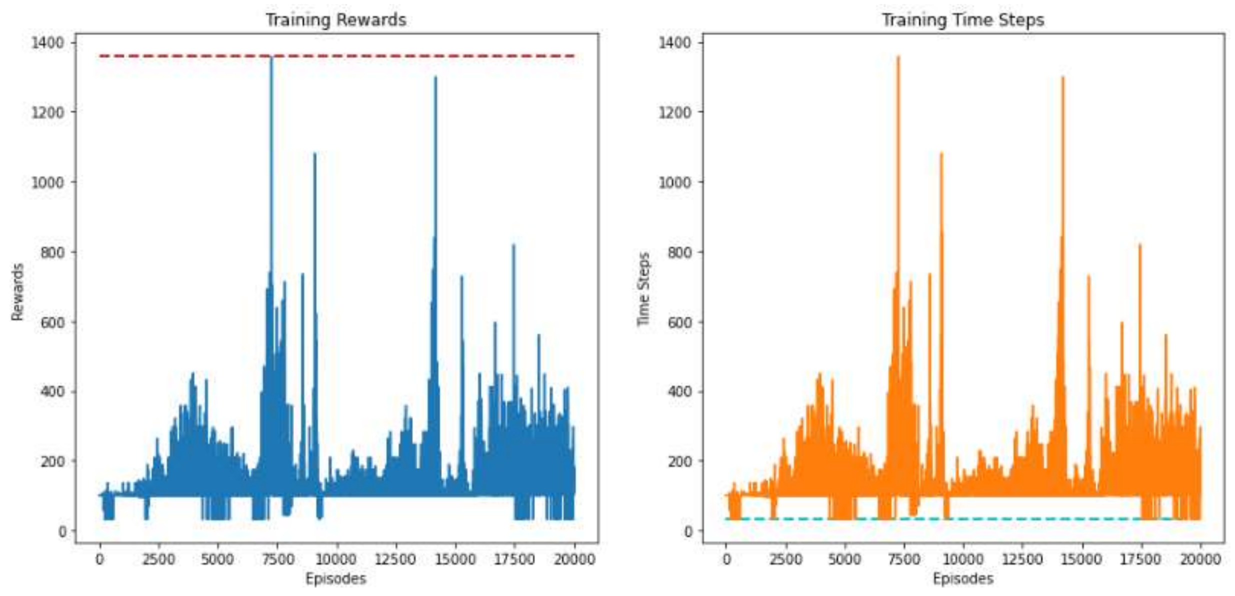
2- Testing


```
    if i_episode % 10 == 0:
        total_reword_eps_test = []     # Initialize the list to include all of the rewards in each episode.
        total_TimeSteps_eps_test = []  # Initialize the list to include all of the time steps in each episode.

        for test in range(5):
            total_rewards_test_each_eps = 0    # Initialize total rewards in each episode
            total_steps_test_each_eps = 0      # Initialize total Time Steps in each episode
            done_ts=False
            state_test=env.reset()

            while not done_ts:
                action_test=agent.get_action(state_test, eps)
                next_state_ts,reward_ts,done_ts,info_ts=env.step(action_test)
                state_test=next_state_ts

                total_rewards_test_each_eps += reward_ts     # Total rewards in each episode
                total_steps_test_each_eps += 2               # Total Time Steps in each episode

            total_reword_eps_test.append(total_rewards_test_each_eps)   # Add total rewards in this episodes to list contains all rewards in each 5 eposides
            total_TimeSteps_eps_test.append(total_steps_test_each_eps)  # Add total Time Steps in this episodes to list contains all Time Steps in each 5 eposides

        total_reword_eps5_test.append(np.mean(total_reword_eps_test))        # list contains all mean of rewards after each 5 episode
        total_TimeSteps_eps5_test.append(np.mean(total_TimeSteps_eps_test))  # list contains all mean of Time Steps after each 5 episode

    print('\rEpisode {}\tTotal Rewards: {:.2f}\tTime Steps: {}'.format(i_episode, total_rewards_train_each_eps, total_steps_train_each_eps))

torch.save(agent.model.state_dict(), 'checkpoint.pth')
Draw_graph(total_episodes, total_reword_eps10_train, total_TimeSteps_eps10_train, total_reword_eps5_test, total_TimeSteps_eps5_test)

return total_reword_eps10_train
```

Results

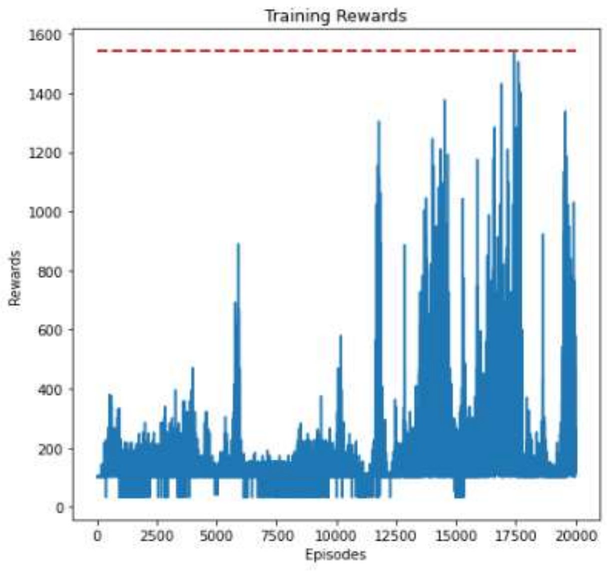
Training Performance

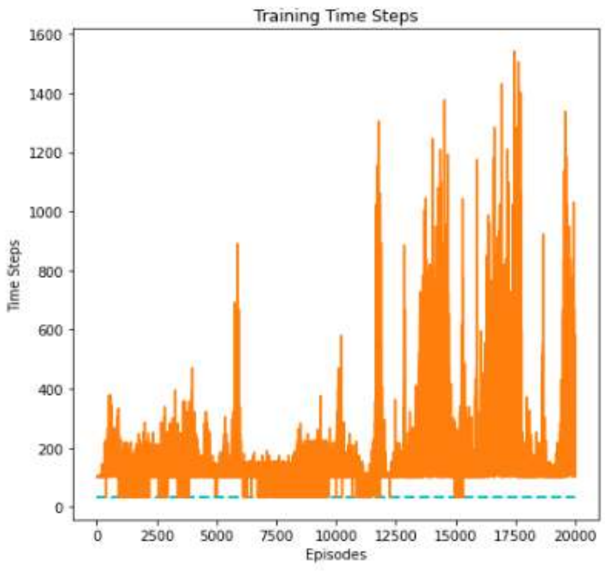
a. Start Epsilon = 1.0, End =0.01, and Decay Epsilon =0.99, Episodes = 20000



The maximum reward is approximate 1390 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 20. We will try to change Epsilon to improve learning.

b. Start Epsilon = 0.8, End =0.01, and Decay Epsilon =0.99, Episodes = 20000

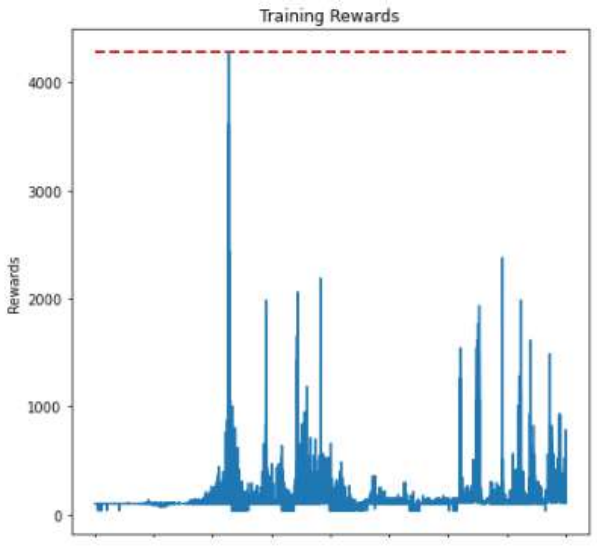


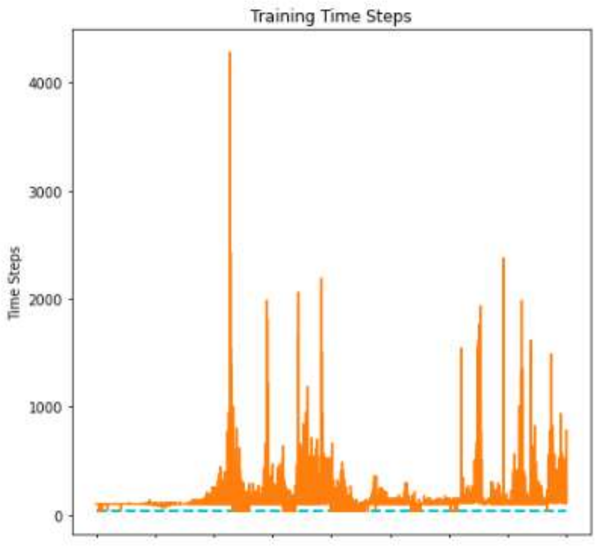


The maximum reward is approximate 1590 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 15.

This epsilon is better than the previous epsilon = 1 as in this trial the agent did not learn well from episodes 0 to 11000, but after that, it learned well. We will try to change Epsilon to improve learning more.

c. Start Epsilon = 0.5, End =0.001, and Decay Epsilon =0.99, Episodes = 20000

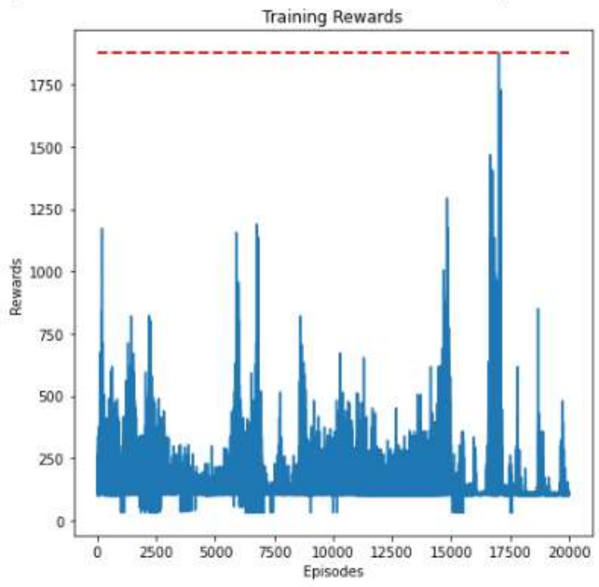


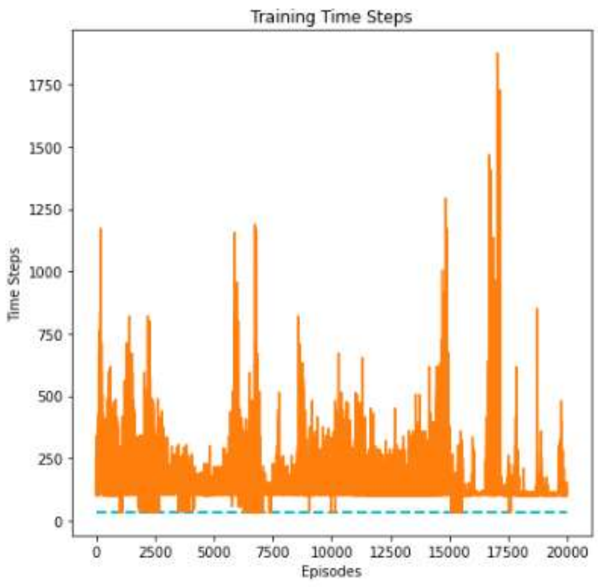


The maximum reward is approximate 4600 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 100.

The previous epsilon is better than this epsilon as in this trial the agent did not learn well.

d. Start Epsilon = 1, End =0.01, and Decay Epsilon =0.0005, Episodes = 20000



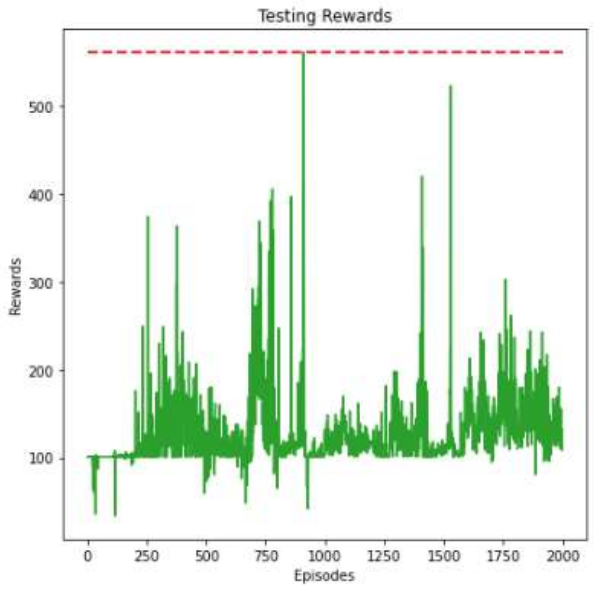


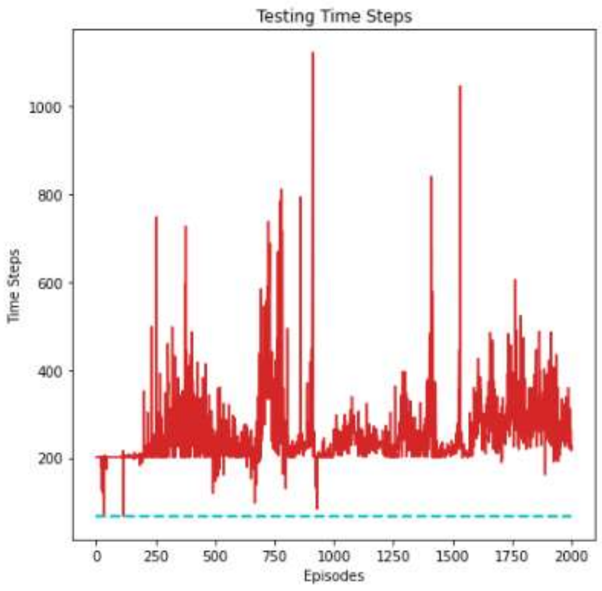
In this trial, we will try to change Decay Epsilon to improve the performance.

The maximum reward is approximate 1900 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 20.

Testing Performance

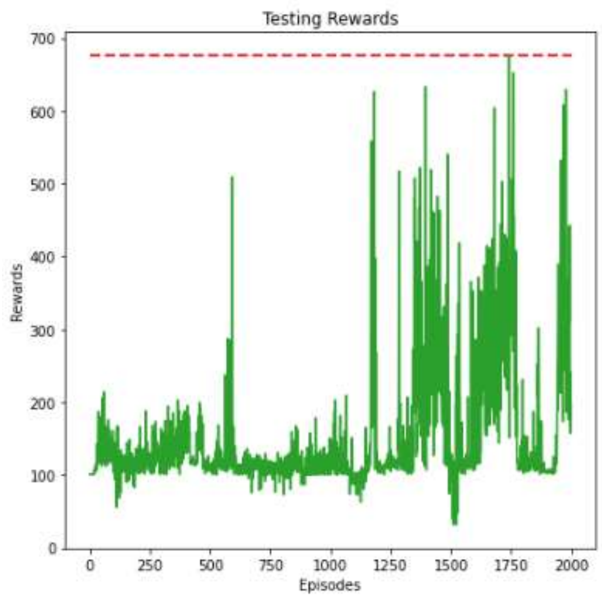
e. Start Epsilon = 1.0, End =0.01, and Decay Epsilon =0.99, Episodes = 2000

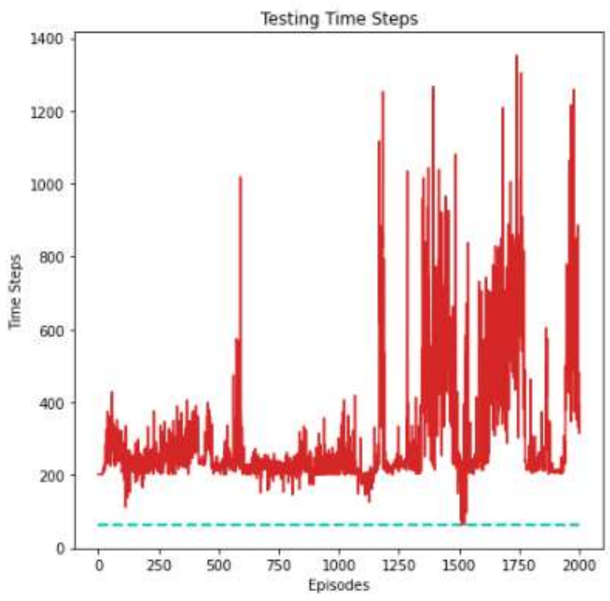




The maximum reward is approximate 600 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 60.

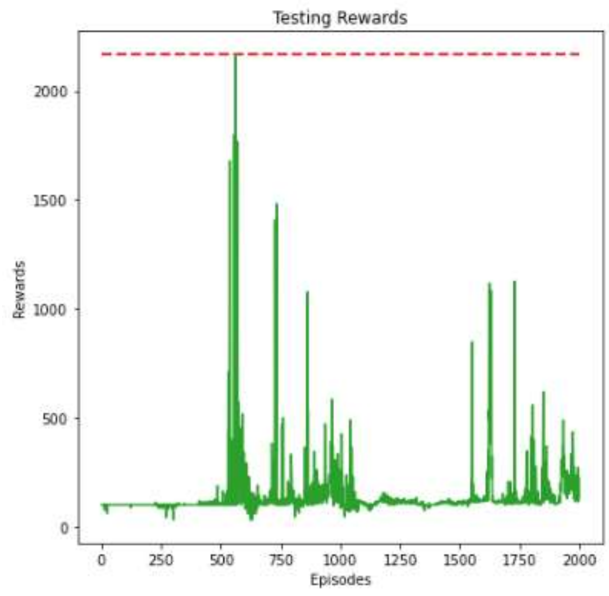
f. Start Epsilon = 0.8, End =0.01, and Decay Epsilon =0.99, Episodes = 2000

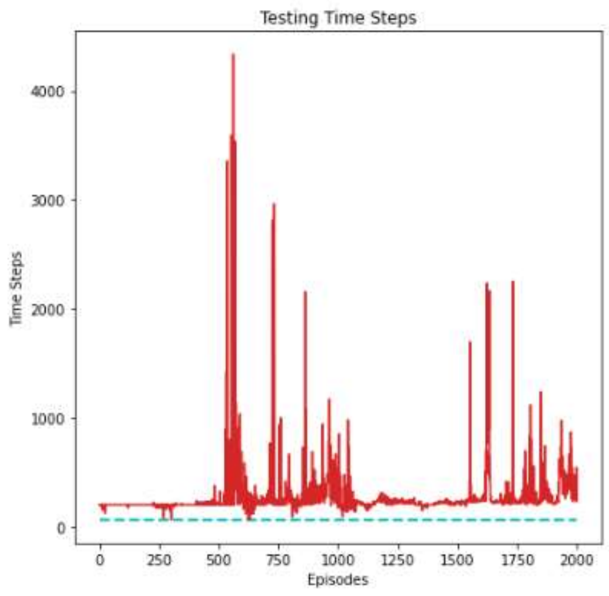




The maximum reward is approximate 690 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 60.

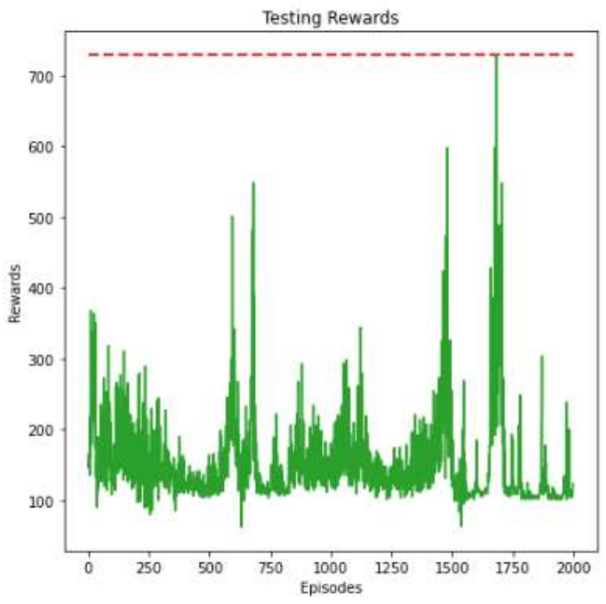
g. Start Epsilon = 0.5, End =0.001, and Decay Epsilon =0.99, Episodes = 2000

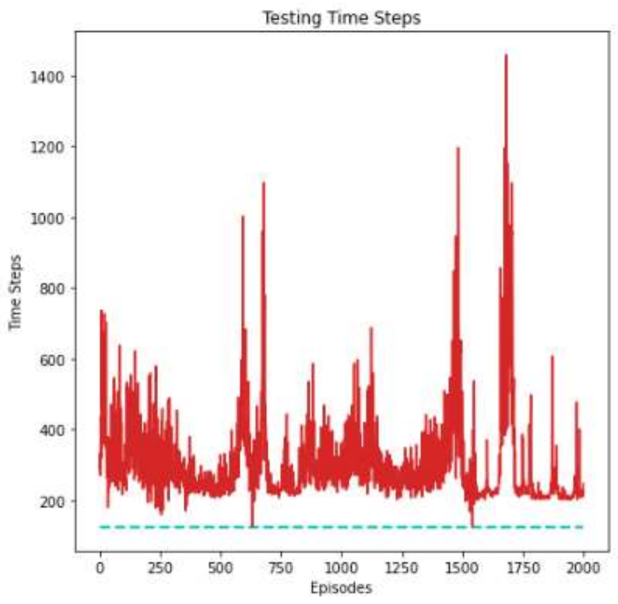




The maximum reward is approximate 2300 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 100.

h. Start Epsilon = 1, End =0.01, and Decay Epsilon =0.0005, Episodes = 2000





The maximum reward is approximate 750 rewards and the minimum number of Time Steps that the agent begins to learn is approximately 100.

Our Observations

Finally, we found the best performance in the model Start Epsilon = 0.8, End Epsilon = 0.01, Decay Epsilon = 0.99, Episodes = 20000, the reward is approximate 1590 rewards, and the time steps approximate 1590 steps because, at the beginning of the learning, the result was not good, but then the result improved until the percentage of non-learning decreased with time, and when we increased the number of episodes, the failure to learn decreases until it almost disappeared.

The number of rewards is equal to the number of time steps, as the reward that is taken when the birds pass the pipe is equal to one and the number of time steps in each step taken is equal to one.

The speed of convergence to the solutions is fast.

Reference

https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b