COMP532 ASSIGNMENT 2

DEEP REINFORCEMENT LEARNING

SUBMITTED BY

USMAN SHOUKAT	STUDENT ID: 201537600
JONE CHONG	STUDENT ID: 201533109
SAHIB BIR SINGH BHATIA	STUDENT ID: 201547831
PARTH KHANDELWAL	STUDENT ID: 201549274
ABDULAZIZ ALGAIN	STUDENT ID: 201560413

ABSTRACT

In this assignment, we used the openAI gym library to import a simple game called “Lunar Lander” to train our deep neural network to play it successfully.

Different deep learning model architectures were discussed and some experiments with the hyperparameters of the model were also done.

After doing the experiments and checking the results, the model with parameters mentioned below gave the best results for the lunar lander game.

1.learning rate: 0.001
2.epsilon decay: 0.0005
3.Number of Neurons: 512
4.Number of Hidden Layers:2

One of the primary challenges that we faced in this assignment was the amount of time taken by the model to train. This is because of lack of GPU capacity in our local machines resulting in training times to be around 4-5 hours for a model with a specific architecture.

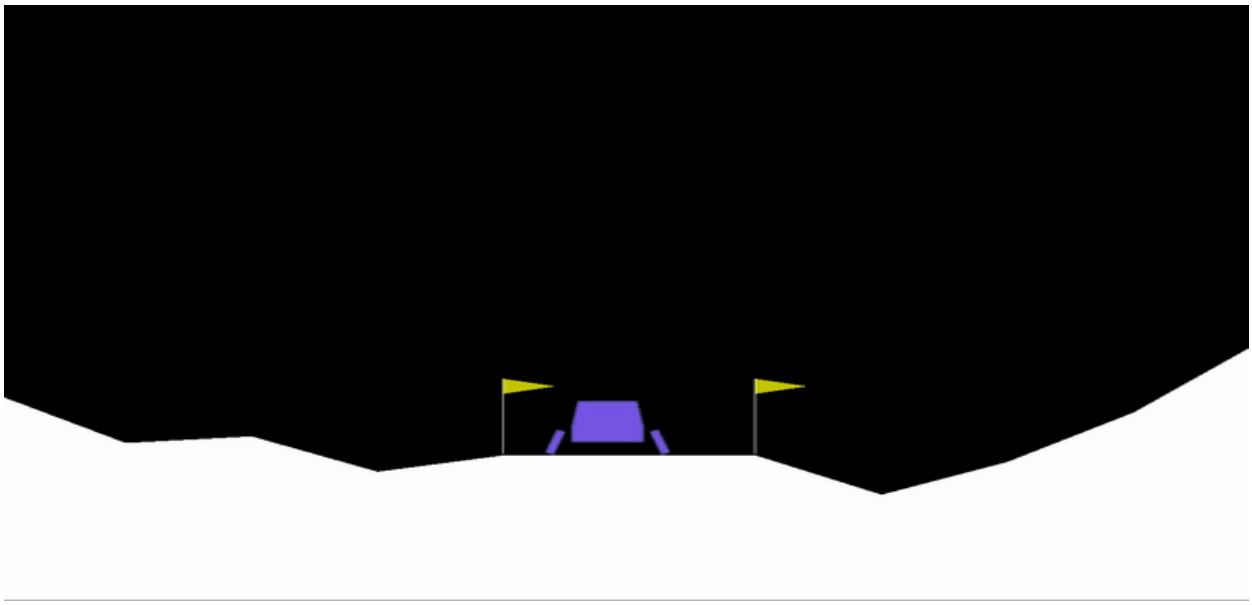

Fig1. Lunar Lander Game

INDEX

Importing openAI gym Game 4

Creating a Network 7

Connecting the game to our network 9

Different DNN Models 10

Implementing our Deep Reinforcement Model 16

Experiments 22

Experiment 1: Changing the Learning Rate 22

Experiment 2: Changing No. of Neurons 23

Experiment 3: Changing Epsilon Decay Value 24

Experiment 4: Changing No. of Hidden Layers 25

Conclusion 27

References 28

LIST OF FIGURES

Figure 1. Lunar Lander Game
Figure 2. Installing the openAI gym library
Figure 3. Visualization of the Lunar Lander Game
Figure 4. Importing the Lunar Lander Game
Figure 5. Our Neural Network
Figure 6. Class DeepQNetwork and forward Function
Figure 7. Connecting the Game to Network
Figure 8. Basic vs Double Q Learning Update
Figure 9. Double DQN
Figure 10. DQN vs DDQN scores for different Atari games
Figure 11. Double Deep Q Learning on Lunar-Lander V2
Figure 12 Duelling DQN
Figure 13. Agent Class and Initialiser Function
Figure 14. Store Transition Function and Choose Action Function
Figure 15. Learn Function
Figure 16. Train Function
Figure 17. Load Agent Function and Trained Agent Function
Figure 18. main function
Figure 19. Plot Function
Figure 20. Scores for different neurons in 3 hidden layers
Figure 21. Scores for different neurons in hidden layers
Figure 22. Scores for different learning rates
Figure 23. Exploration vs Exploitation
Figure 24. Scores for different decaying rates of epsilon

IMPORTING OPENAI GYM GAME

OpenAI Gym is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new Machine Learning algorithms (Reinforcement Learning)

In order to import openAI gym library in python we need to install it first. We can do it by running the following command in the terminal as shown in fig 2.

```
sahibbhatia@Sahibs-MacBook-Air ~ % pip install gym
Collecting gym
  Downloading gym-0.18.3.tar.gz (1.6 MB)
     |████████████████████████████████| 1.6 MB 2.8 MB/s
Requirement already satisfied: scipy in /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages (from gym) (1.7.0)
Requirement already satisfied: numpy>=1.10.4 in /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages (from gym) (1.20.1)
Collecting pyglet<=1.5.15,>=1.4.0
  Downloading pyglet-1.5.15-py3-none-any.whl (1.1 MB)
     |████████████████████████████████| 1.1 MB 5.5 MB/s
Requirement already satisfied: Pillow<=8.2.0 in /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages (from gym) (8.1.0)
Collecting cloudpickle<1.7.0,>=1.2.0
  Downloading cloudpickle-1.6.0-py3-none-any.whl (23 kB)
Using legacy 'setup.py install' for gym, since package 'wheel' is not installed.
Installing collected packages: pyglet, cloudpickle, gym
    Running setup.py install for gym ... done
Successfully installed cloudpickle-1.6.0 gym-0.18.3 pyglet-1.5.15
```

Fig 2. Installing openAI gym

The environment has four parameters namely:

- **Observation(**object**):** an environment-specific object representing your observation of the environment. For example, pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.
- **Reward(**float): amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward.
- **Done(**Boolean): whether it's time to reset the environment again. Most (but not all) tasks are divided up into well-defined episodes, and done being true indicates the episode has terminated. (For example, perhaps the pole tipped too far, or you lost your last life.)

- **Info(**dict**):** diagnostic information useful for debugging. It can sometimes be useful for learning (for example, it might contain the raw probabilities behind the environment's last state change). However, official evaluations of your agent are not allowed to use this for learning.

LUNAR LANDER GAME

As mentioned above the openAI gym environment help us to simulate where a lander needs to land at a specific location under low-gravity conditions, and has a well-defined physics engine implemented.

The main goal of the game is to direct the agent to the landing pad as softly and fuel-efficiently as possible. The state space is continuous as in real physics, but the action space is discrete.

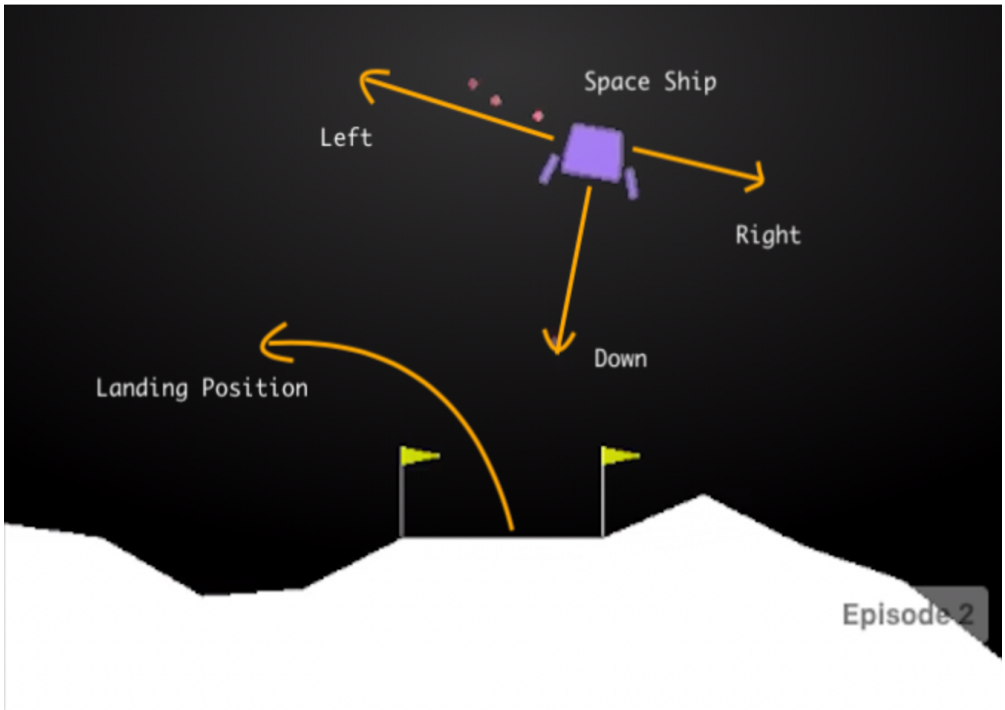

Fig.3 Visualization of the Lunar Lander Game

There are 8 state variables associated with the state space, as shown below:

- x coordinate of the lander
- y coordinate of the lander
- v_x, the horizontal velocity

- vy, the vertical velocity
- Θ, the orientation in space
- vθ, the angular velocity
- Left leg touching the ground (Boolean)
- Right leg touching the ground (Boolean)

All the coordinate values are given relative to the landing pad instead of the lower left corner of the window. The x coordinate of the lander is 0 when the lander is on the line connecting the centre of the landing pad to the top of the screen.

Therefore, it is positive on the right side of the screen and negative on the left. The y coordinate is positive at the level above the landing pad and is negative at the level below

There are four discrete actions available:

- [0]do nothing,
- [1[fire left orientation engine,
- [2]fire right orientation engine,
- and [3]fire main engine.

Firing the left and right engines introduces a torque on the lander, which causes it to rotate, and makes stabilizing difficult.

```
#Initalising the environment
env = gym.make('LunarLander-v2')
stateSize = env.observation_space.shape[0]
actionSize = env.action_space.n
```

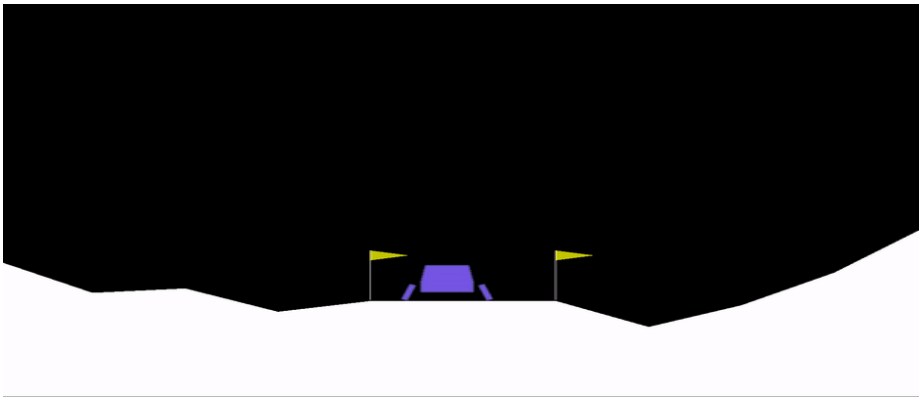

Fig 4. Importing the Lunar Lander Game

CREATING A NETWORK

We use a 4 layer neural network, as shown in Fig. 5 with 512 neurons in the hidden layers. We use ReLu activation in our network.

The learning rate: 0.001, epsilon decay: 0.0005 ,
Number of Neurons: 512 ,Number of Hidden Layers:2

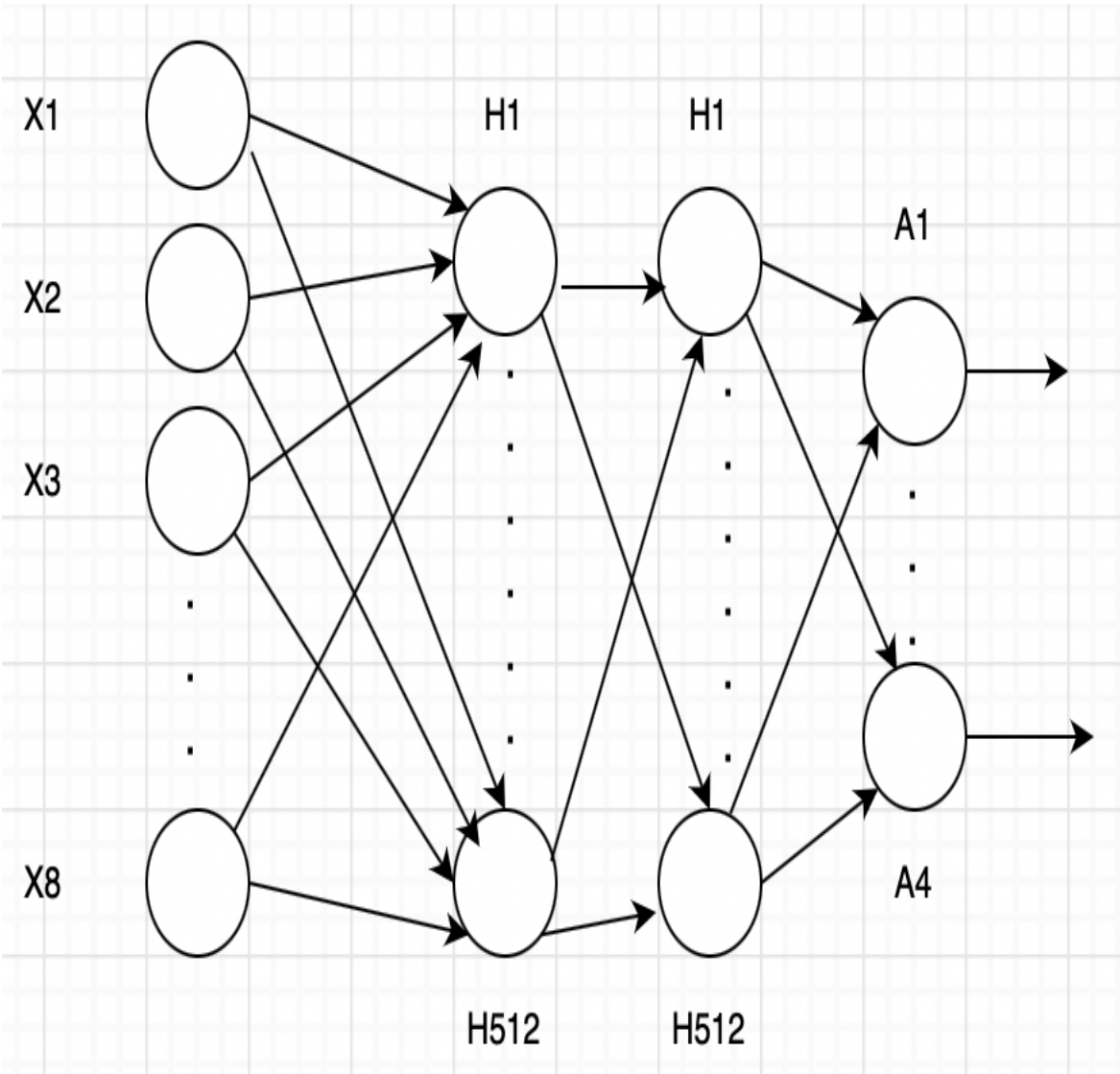

Fig. 5 Our Neural Network

```python
class DeepQNetwork(nn.Module):
    """
    This Class takes the architecture of network as the parameter and trains it

    Parameters:
        inputDimensions : Number of dimensions in the input layer
        fc1Dims : Number of dimensions in first hiddden layer
        fc2Dims : Number of dimensions in second hiddden layer
        numberOfActions : Number of dimensions of output layer
        type: whether to train the network or use only forward pass
    """
    def __init__(self, inputDimensions, fc1Dims, fc2Dims,
            numberOfActions, type = "training", learningRate = 0.001):
        super(DeepQNetwork, self).__init__()
        #Initialisation of parameters
        self.inputDimensions = inputDimensions
        self.fc1Dims = fc1Dims
        self.fc2Dims = fc2Dims
        self.numberOfActions = numberOfActions
        self.fc1 = nn.Linear(self.inputDimensions, self.fc1Dims)
        self.fc2 = nn.Linear(self.fc1Dims, self.fc2Dims)
        self.fc3 = nn.Linear(self.fc2Dims, self.numberOfActions)

        #Checking if the agent is in training mode or online mode
        if type == "training":
            self.optimizer = optim.Adam(self.parameters(), lr=learningRate)
            self.loss = nn.MSELoss()


    def forward(self, state):
        """
        This method takes an in for neural network and  passes it through the forward pass
        and at the end returns the actions from output layer
        """
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        actions = self.fc3(x)

        return actions
```

Fig.6 Class DeepQNetwork and forward Function

CONNECTING THE GAME TO OUR NETWORK

```
def trainTheAgent():
    """
    This method takes the hyper parameters from the user, initialised the agent and environment,
    pass the agent through the environment and help it learning. At the end it plots a graph to
    show the learning process of the agent
    """

    #Taking hyperparameters from the user
    print("\nEnter the following hyperparameters\n")
    gamma = float(input("Enter the value of gamma : "))
    epsDecay = float(input("Enter the value of Epsilon Decay : "))
    learningRate = float(input("Enter the Learning Rate : "))
    fc1Dims = int(input("Enter the dimensions of first fully connected hidden layer : "))
    fc2Dims = int(input("Enter the dimensions of second fully connected hidden layer : "))
    episodes = int(input("Enter the number of episodes : "))

    #Initalising the environment
    env = gym.make('LunarLander-v2')
    stateSize = env.observation_space.shape[0]
    actionSize = env.action_space.n

    #initialising the agent
    agent = Agent(gamma=gamma, epsilon=1.0, batchSize=64, numberOfactions=actionSize, epsMin=0.01,
                  inputDimensions=stateSize, learningRate=learningRate, epsDecay=epsDecay,
fc1Dims=fc1Dims, fc2Dims=fc2Dims)
    scores, epsHistory = [], []

    #Running through each episode
    for i in range(episodes):
        score = 0
        done = False
        observation = env.reset()
        while not done:
            action = agent.chooseAction(observation)  #chosing the action
            observation_, reward, done, info = env.step(action) #step into next state
            score += reward
            agent.storeTransition(observation, action, reward, #storing transition in the memory
                                  observation_, done)
            agent.learn() #learning through experience
            observation = observation_

        #taking account of learning
        scores.append(score)
        epsHistory.append(agent.epsilon)

        averageScore = np.mean(scores[-100:])

        print('Episode ', i, 'score %.2f' % score, 'average score %.2f' % averageScore,'epsilon
%.2f' % agent.epsilon)

    #plotting the graph
    x = [i+1 for i in range(episodes)]
    figName = 'lunar_lander.png'
    plotLearning(x, scores, epsHistory, figName)
    #Saving the agent if asked
    save = input("\nEnter 1 if you want to save the Agent otherwise press any other key : ")
    if save == "1":
        filename = input("Enter the file name : ")
        T.save(agent.qEvaluation.state_dict(), filename)
```

Fig.7 Connecting the Game to Network

DIFFERENT DNN MODELS

DEEP Q LEARNING METHOD[DQN]

The DQN method makes use of a multi-layer perceptron, called a Deep Q-Network (DQN), to estimate the Q values. The input to the network is the current state (8-dimensional in our case) and the outputs are the Q values for all state-action pairs for that state.

The Q-Learning update rule is as follows:

$$\mathbf{Q(s,a)=Q(s,a)+\alpha(r+\gamma \max Q(s',a')-Q(s,a))}$$

The optimal Q-value Q* (s, a) is estimated using the neural network with parameters θ. This becomes a regression task, so the loss function at iteration i is obtained by the temporal difference error:

$$L_i(\theta_i) = E_{(s,a)\sim\rho(.)}[(y_i - Q(s, a; \theta_i))^2]$$

Where , $y_i = E_{s'\sim E}[r+\gamma \max Q(s',a';\theta_{i-1})]$

Here, θ_{i-1} are the network parameters from the previous iteration, ρ(s,a) is a probability distribution over states s and actions a, and E is the environment the agent interacts with.

Gradient descent is then used to optimise the loss function and update the neural network weights. The neural network parameters from the previous iteration, θ_{i-1}, are kept fixed while optimizing $L_i(\theta_i)$.

Since the next action is selected based on the greedy policy, Q-learning is an off policy algorithm

DOUBLE DEEP Q LEARNING METHOD [DDQN]

note to marker [These are just theoretical proposals for the project, the method we chose was solely the DQN mentioned above]

Proposed Addition

A proposed addition to the DQN implemented in this project would be to introduce Double DQN (DDQN). In this section, we review the basis and reasoning for DDQN. Furthermore, we theorize the implementation and performance of the DDQN on the LunarLander-V2 based on previous work done.

Double Q learning is introduced by [5] for overcoming large estimations of Q-values (overly optimistic) in regular Q-learning. The overestimations of the Q-values are caused by the agents choosing the non-optimal values due to the maximum estimated Q-value.

The agent has no information on the environment and updates the Q(s,a) in each iteration. This results in noise that causes positive bias, as the maximum estimated Q-value is not guaranteed to be the best. When taking a non-optimal action based on the maximum Q-value, this problem is named an overestimation.

In the double Q learning algorithm, there are two action value functions for estimators, instead of one. This mitigates noise as the noise can be seen as uniform distribution between the two estimators, solving overestimations. The update rule is given by Fig 8.

Basic Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Double Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \boxed{Q'(s_{t+1}, \boxed{a})} - Q(s_t, a_t))$$

estimated/expected Q-value

$$\boxed{a} = \max_a Q(s_{t+1}, a)$$

$$q_{estimated} = \boxed{Q'(s_{t+1}, \boxed{a})}$$

Fig. 8 Basic vs Double Q Learning Update [6]

As seen in the equations above:

1. The Q function selects the action with maximum Q value of next state.
2. The Q' function calculates the Q value based on that selected action.
3. The Q function is then updated using the Q' function's expected Q value.

Basis of Double Q Network

The double Q network is an inspiration based on the double Q learning algorithm discussed and involves two different networks: a DQN and a target network.

Double Q Network

estimated/expected Q-value

$$Q_{qnet}(s_t, a_t) \leftarrow R_{t+1} + \gamma \boxed{Q_{tnet}(s_{t+1}, \boxed{a})}$$

$$\boxed{a} = \max_{a} Q_{qnet}(s_{t+1}, a)$$

$$q_{estimated} = \boxed{Q_{tnet}(s_{t+1}, \boxed{a})}$$

Fig.9 Double DQN [6]

In the equations above:

1. The DQN (Qqnet) selects the action with maximum Q value of next state.
2. The target network (Qtnet) calculates the Q value based on that selected action.
3. The Q value of the DQN is then updated using the target network's Q-value estimation.

$$\boxed{Q_{tnet}(s, a) = Q_{qnet}(s, a)}$$

4. The parameters of the target network are updated based on the DQN (as per number of iterations specified).
5. Update the parameters of DQN in the optimizer

Theoretical Implementation

According to [5], double Q networks were successfully used for performance improvement for openAI gym games. In our project, due to time constraints, we theorize the implementation of DDQN based on the previous work by [6],[7].

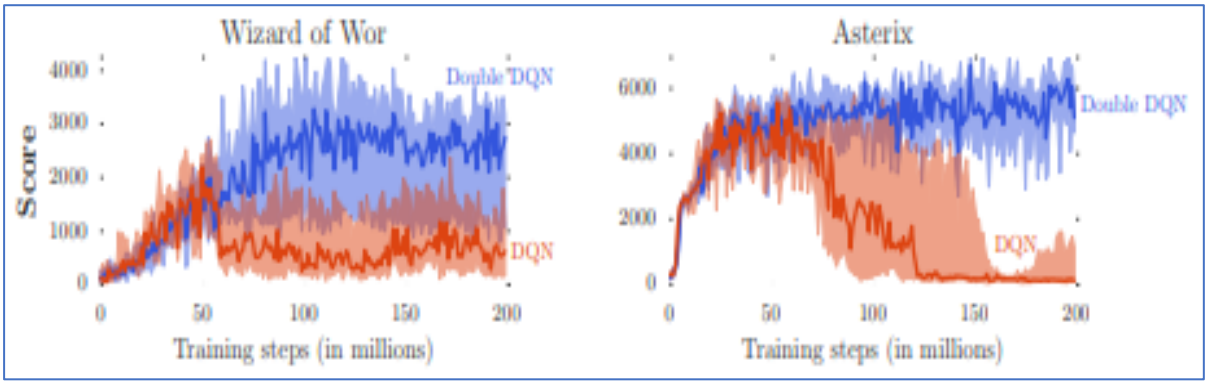

Fig.10 DQN vs DDQN scores for different Atari games [5]

In [6], a DDQN is implemented with Lunar-LanderV2, with a learning rate of 0.001, 1000 iterations, and using the ADAM optimizer.

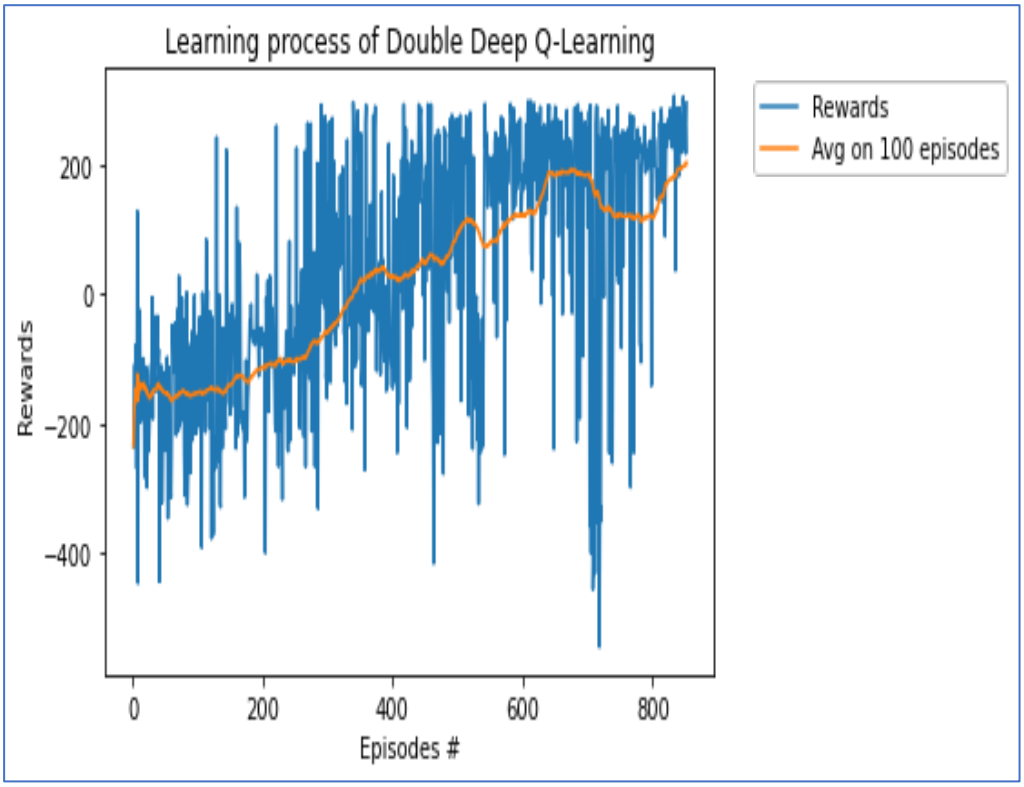

Fig.11 Double Deep Q Learning on Lunar-Lander V2 [6]

ACTOR-CRITIC METHOD

According to the tutorial documentation by TensorFlow [8]:

"Actor-Critic methods are temporal difference (TD) learning methods that represent the policy function independent of the value function, returning a probability distribution over actions that the agent can take based on the given state. A value function determines the expected return for an agent starting at a given state and acting according to a particular policy forever after.

In the Actor-Critic method, the policy is referred to as the actor that proposes a set of possible actions given a state, and the estimated value function is referred to as the critic, which evaluates actions taken by the actor based on the given policy.

Duelling DQN method

According to [9]:

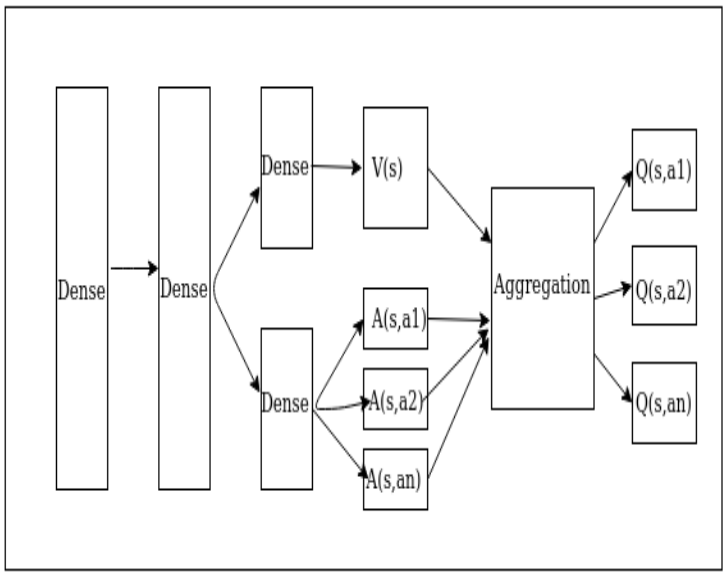

Fig.12 Duelling DQN [9]

In duelling DQN, there are two different estimates which are: the estimate for the value of a given state, which estimates how good it is for an agent to be in that state; and the estimate for the advantage of each action in a state.

IMPLEMENTING OUR DEEP REINFORCEMENT LEARNING MODEL

The following code implements our deep reinforcement model:

```
class Agent():
    """
    This class takes the hyperparameter for the agent and trains it
    Parameters:
        gamma : discount factor
        epsilon : proportion of exploration
        epsMin : minimum proportion of exploration
        epsDecay : decaying rate of proportion of exploration
        learningRate : learning rate
        inputDimensions : Number of dimensions in the input layer
        fc1Dims : Number of dimensions in first hiddden layer
        fc2Dims : Number of dimensions in second hiddden layer
        numberOfActions : Number of dimensions of output layer
        batchSize : size of the batch taken from the memory
        maxMemorySize : maximum size of the memory
        stateMemory : memory for states
        newStateMemory : memory for next states
        actionMemory : memory of actions
        rewardMemory : memory of rewards
        terminalMemory : memory of terminal states


    """
    def __init__(self, gamma, epsilon, learningRate, inputDimensions, batchSize, numberOfactions,
fc1Dims = 512, fc2Dims = 512,
            maxMemorySize=100000, epsMin=0.05, epsDecay=0.001):
        #Initialisation of parameters
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsMin = epsMin
        self.epsDecay = epsDecay
        self.leaarningRate = learningRate
        self.actionSpace = [i for i in range(numberOfactions)]
        self.memorySize = maxMemorySize
        self.batchSize = batchSize
        self.memoryCounter = 0
        self.iterationCounter = 0
        # self.replace_target = 100

        #Initailisation of the network
        self.qEvaluation = DeepQNetwork(learningRate = learningRate,
numberOfActions=numberOfactions, inputDimensions=inputDimensions,
                                    fc1Dims=fc1Dims, fc2Dims=fc2Dims)


        self.stateMemory = np.zeros((self.memorySize, inputDimensions), dtype=np.float32)
        self.newStateMemory = np.zeros((self.memorySize, inputDimensions), dtype=np.float32)
        self.actionMemory = np.zeros(self.memorySize, dtype=np.int32)
        self.rewardMemory = np.zeros(self.memorySize, dtype=np.float32)
        self.terminalMemory = np.zeros(self.memorySize, dtype=np.bool)
```

Fig.13 Agent Class and Initialiser Function

```python
def storeTransition(self, state, action, reward, state_, terminal):
    """
    This method stores the transitions in memory after each step
    parameters :
        state : current state
        action : action taken
        state_ : next state
        reward : reward collected
        terminal : whether the episode terminated or not (bool)
    """
    index = self.memoryCounter % self.memorySize
    self.stateMemory[index] = state
    self.newStateMemory[index] = state_
    self.rewardMemory[index] = reward
    self.actionMemory[index] = action
    self.terminalMemory[index] = terminal

    self.memoryCounter += 1

def chooseAction(self, observation):
    """
    This method takes the observation of the environment and returns the action to be taken
    """
    if np.random.random() > self.epsilon:
        state = T.tensor([observation])
        actions = self.qEvaluation.forward(state)
        action = T.argmax(actions).item()
    else:
        action = np.random.choice(self.actionSpace)

    return action
```

Fig.14 Store Transition Function and Choose Action Function

```python
    def learn(self):
        """
        This method holds the whole lerning process of the agent
        """
        if self.memoryCounter < self.batchSize:
            return

        #making the gradients equal to zero
        self.qEvaluation.optimizer.zero_grad()

        #Getting a batch from the memory
        maxMemory = min(self.memoryCounter, self.memorySize)

        batch = np.random.choice(maxMemory, self.batchSize, replace=False)

        batchIndex = np.arange(self.batchSize, dtype=np.int32)

        #Getting the batch of state, new state, reward, action, and termination information from
memory
        stateBatch = T.tensor(self.stateMemory[batch])
        newStateBatch = T.tensor(self.newStateMemory[batch])
        actionBatch = self.actionMemory[batch]
        rewardBatch = T.tensor(self.rewardMemory[batch])
        terminalBatch = T.tensor(self.terminalMemory[batch])


        #Getting the actions from the network
        qEvaluation = self.qEvaluation.forward(stateBatch)[batchIndex, actionBatch]
        qNext = self.qEvaluation.forward(newStateBatch)
        qNext[terminalBatch] = 0.0

        #Calculating the target Q value through Q learning
        qTarget = rewardBatch + self.gamma*T.max(qNext,dim=1)[0]

        # loss = self.qEvaluation.loss(qTarget, qEvaluation).to(self.qEvaluation.device)

        #optimising the network
        loss = self.qEvaluation.loss(qTarget, qEvaluation)
        loss.backward()
        self.qEvaluation.optimizer.step()

        self.iterationCounter += 1
        #Decaying the exploration rate
        self.epsilon = self.epsilon - self.epsDecay if self.epsilon > self.epsMin \
                       else self.epsMin
```

Fig. 15 Learn Function

```python
def trainTheAgent():
    """
    This method takes the hyper parameters from the user, initialised the agent and environment,
    pass the agent through the environment and help it learning. At the end it plots a graph to
    show the learning process of the agent
    """

    #Taking hyperparameters from the user
    print("\nEnter the following hyperparameters\n")
    gamma = float(input("Enter the value of gamma : "))
    epsDecay = float(input("Enter the value of Epsilon Decay : "))
    learningRate = float(input("Enter the Learning Rate : "))
    fc1Dims = int(input("Enter the dimensions of first fully connected hidden layer : "))
    fc2Dims = int(input("Enter the dimensions of second fully connected hidden layer : "))
    episodes = int(input("Enter the number of episodes : "))

    #Initalising the environment
    env = gym.make('LunarLander-v2')
    stateSize = env.observation_space.shape[0]
    actionSize = env.action_space.n

    #initialising the agent
    agent = Agent(gamma=gamma, epsilon=1.0, batchSize=64, numberOfactions=actionSize, epsMin=0.01,
                  inputDimensions=stateSize, learningRate=learningRate, epsDecay=epsDecay,
fc1Dims=fc1Dims, fc2Dims=fc2Dims)
    scores, epsHistory = [], []

    #Running through each episode
    for i in range(episodes):
        score = 0
        done = False
        observation = env.reset()
        while not done:
            action = agent.chooseAction(observation)  #chosing the action
            observation_, reward, done, info = env.step(action) #step into next state
            score += reward
            agent.storeTransition(observation, action, reward, #storing transition in the memory
                                  observation_, done)
            agent.learn() #learning through experience
            observation = observation_

        #taking account of learning
        scores.append(score)
        epsHistory.append(agent.epsilon)

        averageScore = np.mean(scores[-100:])

        print('Episode ', i, 'score %.2f' % score, 'average score %.2f' % averageScore,'epsilon
%.2f' % agent.epsilon)

    #plotting the graph
    x = [i+1 for i in range(episodes)]
    figName = 'lunar_lander.png'
    plotLearning(x, scores, epsHistory, figName)
    #Saving the agent if asked
    save = input("\nEnter 1 if you want to save the Agent otherwise press any other key : ")
    if save == "1":
        filename = input("Enter the file name : ")
        T.save(agent.qEvaluation.state_dict(), filename)
```

Fig.16 Train Function

```python
def loadAgent(stateSize, actionSize):
    """
    This method takes input and output dimension of the network, ask for the architecture of
    the network, loads the trained network and retrns it
    """
    print("Enter the architecture of the network\nThe hidden units for our trained model were 512
units each layer")
    fc1Dims = int(input("Enter the dimensions of first fully connected hidden layer : "))
    fc2Dims = int(input("Enter the dimensions of second fully connected hidden layer : "))
    filename = input("Enter the name of the file of trained agent : ")
    #initialising the network
    agent = DeepQNetwork(inputDimensions=stateSize ,fc1Dims=fc1Dims,
fc2Dims=fc2Dims,numberOfActions=actionSize, type = "trained" )
    #loading the trained network
    agent.load_state_dict(T.load(filename))
    return agent

def trainedAgent():
    """
    This method loads the trained agent and environment and shows its live performance
    """
    #initialisng the environment
    env = gym.make('LunarLander-v2')
    stateSize = env.observation_space.shape[0]
    actionSize = env.action_space.n
    #loading the agent
    agent = loadAgent(stateSize, actionSize)
    state = env.reset()
    state = np.reshape(state, [1, stateSize])
    score = 0
    #running the agent
    for t in range(500):
        env.render()
        action = np.argmax(agent.forward(Variable(T.Tensor(state))).data.numpy()[0]) #deciding the
action
        next_state, reward, done, info = env.step(action)
        score += reward
        next_state = np.reshape(next_state, [1, stateSize])
        state = next_state
        time.sleep(0.01)
        if done:
            break
    env.close()
    print("\nscore: {}".format(score), "\n")
```

Fig.17 Load Agent Function and Trained Agent Function

```python
def main():
    """
    This method gives the option either to train an agent or seee the performance of trained
agent,
    after taking the input it do thr desired action
    """
    while  True:
        menu = input("\n\nEnter from the following choices : \n1. Press 1 if you want to train an
agent\n2. Press 2 if you want to run a trained agent\n3. Press q/Q to quit : ")
        if menu == "q" or menu == "Q":
            break
        elif menu == "1":
            trainTheAgent()
        elif menu == "2":
            trainedAgent()
        else:
            print("Enter Valid Input : ")



if __name__ == '__main__':
  main()
```

Fig.18 main Function

```python
def plotLearning(x, scores, epsilons, figName):
    """
    This Method takes the scores and epsilon history as parameters and plot the their
    plots vs epsisode
    """

    #Computing the moving average
    N = len(scores)
    movingAverage = np.empty(N)
    for t in range(N):
            movingAverage[t] = np.mean(scores[max(0, t-20):(t+1)])

    #Plotting the Graphs
    fig, ax1 = plt.subplots()
    ax2 = ax1.twinx()
    ax1.plot(x, epsilons, label = "Epsilon", color = "r")
    ax2.plot(x, movingAverage, label = "Scores", color = "g")
    ax1.set_xlabel("Episode", color="r")
    ax1.set_ylabel("Epsilon", color="r")
    ax1.tick_params(axis='x', colors="r")
    ax1.tick_params(axis='y', colors="r")
    ax2.axes.get_xaxis().set_visible(False)
    ax2.yaxis.tick_right()
    ax2.set_ylabel('Score', color="g")
    ax2.yaxis.set_label_position('right')
    ax2.tick_params(axis='y', colors="g")
    plt.legend()
    plt.savefig(figName)
```

Fig.19 Plot Function

EXPERIMENTS

EXPERIMENT 1: CHANGING THE NUMBER OF HIDDEN LAYERS

We decided to experiment with the number of hidden layers .For this we changed the number of neurons from 64 to 512 neurons in the 3 hidden layers and checked the scores over the episodes as shown in fig 20.

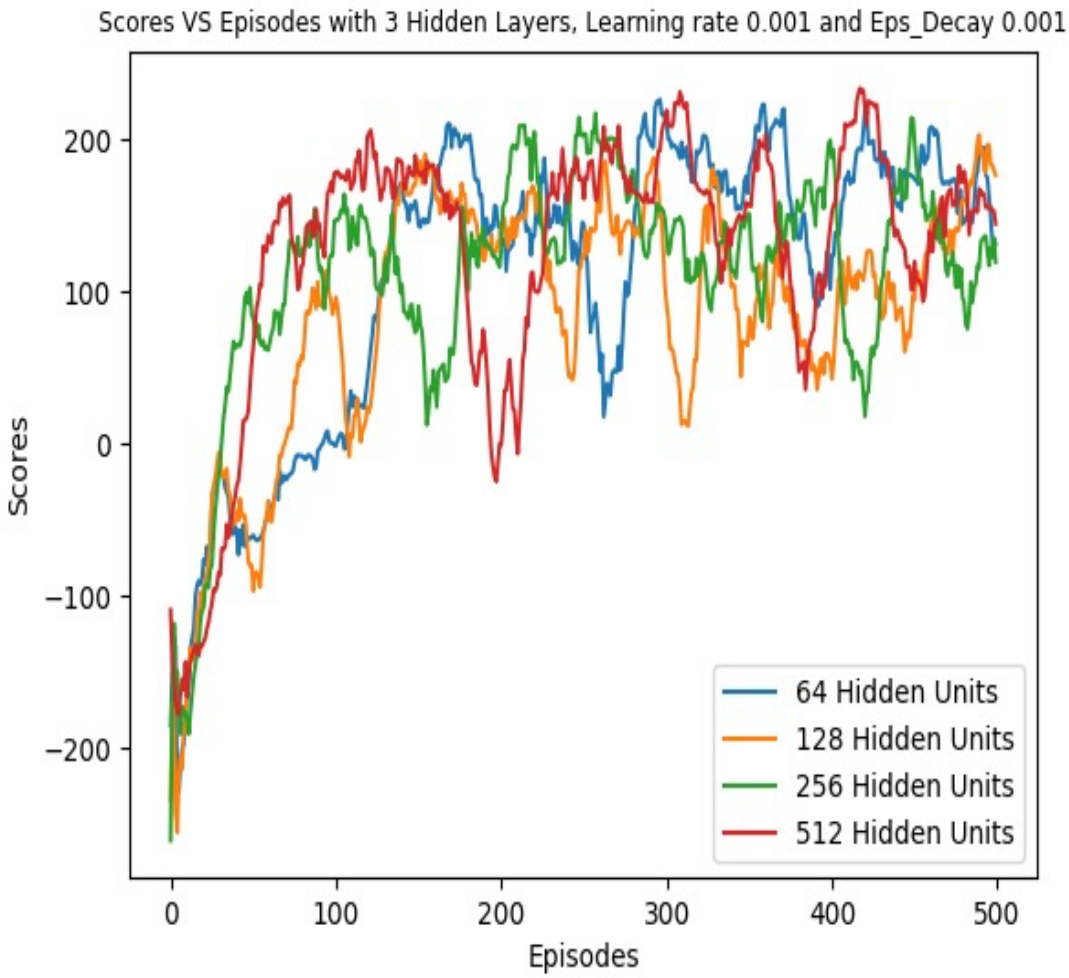

Fig.20 Scores for different neurons in 3 hidden layers

As shown in Fig 20 we can clearly see that the scores for 3 hidden layers are less than that the scores with 2 hidden layers. Thus, we can say that by increasing the hidden layers of our network the performance of the network decreases in case of the Lunar Lander game.

EXPERIMENT 2: CHANGING NUMBER OF NEURONS IN HIDDEN LAYER

Next, we decided to experiment with the number of neurons in the 2 hidden layers. For this we changed the number of neurons from 64 to 512 neurons in the hidden layers and checked the scores over the episodes as shown in fig 21.

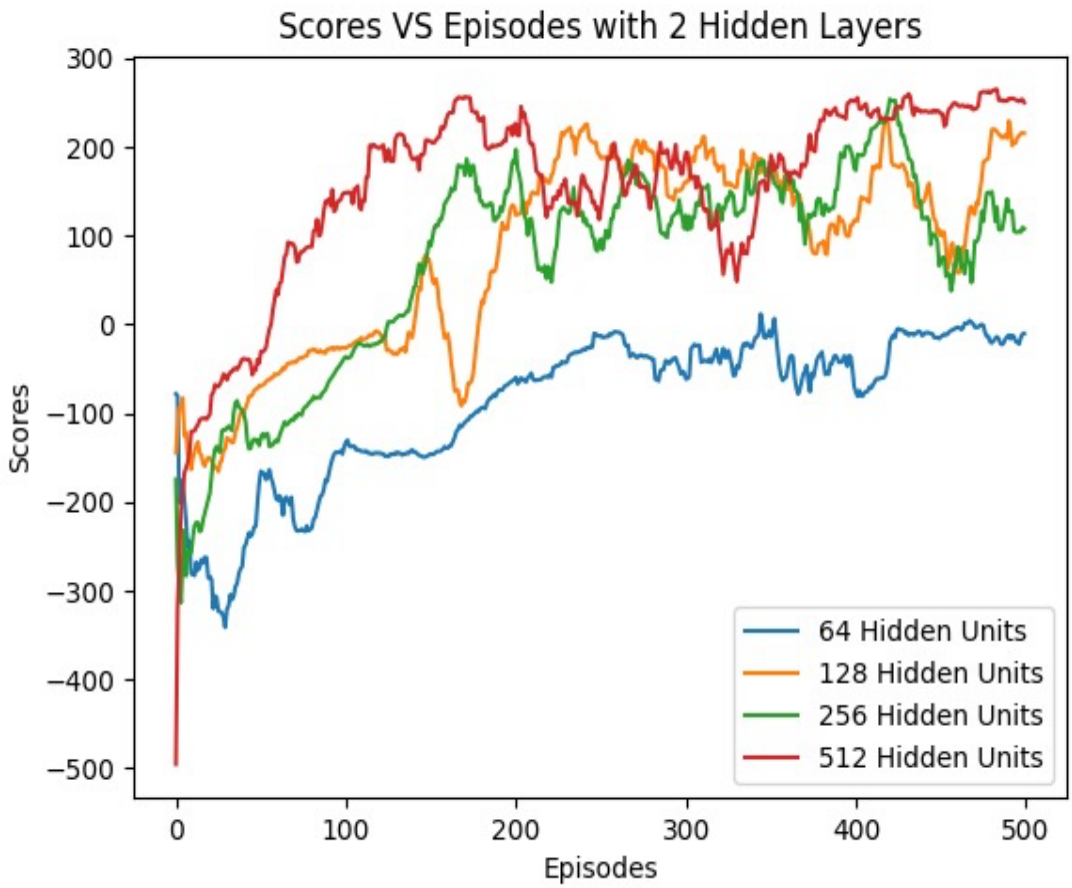

Fig.21 Scores for different neurons in the hidden layers

As shown in Fig 21 ,we can clearly see that as the number of neurons increased in the hidden layer the score of the network also increased with best score for 512 neurons and worst score for 64 neurons.

One downside of increasing the number of neurons in the model was the training time with the neural network of 512 neurons taking up to 5 hours to learn. The learning process can be made faster if we have more GPU power.

EXPERIMENT 3: CHANGING LEARNING RATE

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. We experimented with learning rate to check the influence of it on the scores over the episodes.

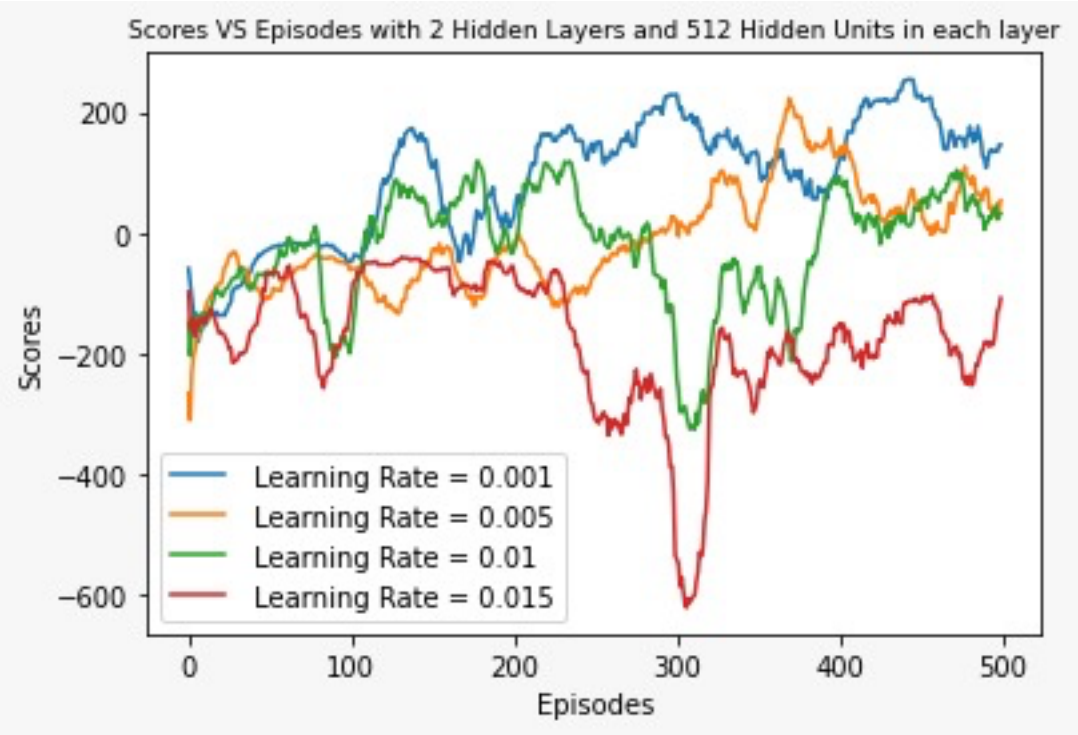

Fig.22 Scores for different learning rates

As shown in Fig.22 , we can clearly see that the best scores were obtained for learning rate = 0.001 and worst scores were obtained when we increased the learning rate to 0.01.

In conclusion, we can say that it is true that by increasing the learning rate the model learns quickly but it doesn't learn effectively.

EXPERIMENT 4: CHANGING THE VALUE OF EPSILON DECAY

Our model follows the epsilon greedy action selection meaning the agent uses both exploitations to take advantage of prior knowledge and exploration to look for new options

The epsilon-greedy approach selects the action with the highest estimated reward most of the time. The aim is to have a balance between exploration and exploitation.

Exploration allows us to have some room for trying new things, sometimes contradicting what we have already learned.

With a small probability of epsilon, we choose to explore, i.e., not to exploit what we have learned so far. In this case, the action is selected randomly, independent of the action-value estimates.

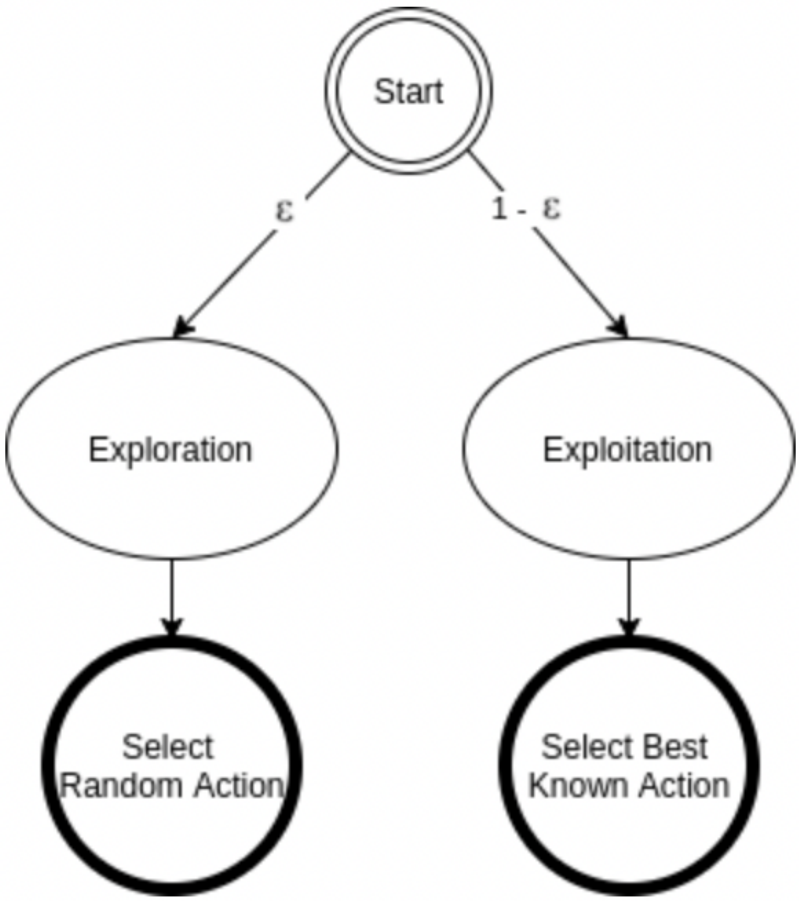

Fig.23 Exploration vs Exploitation

If epsilon is set to 0, we never explore but always exploit the knowledge we already have. On the contrary, having the epsilon set to 1 force the algorithm to always take random actions and never use past knowledge. Usually, epsilon is selected as a small number close to 0.

We experimented with different decaying values of epsilon in order to check its influence on the score over the episodes as shown in Fig24.

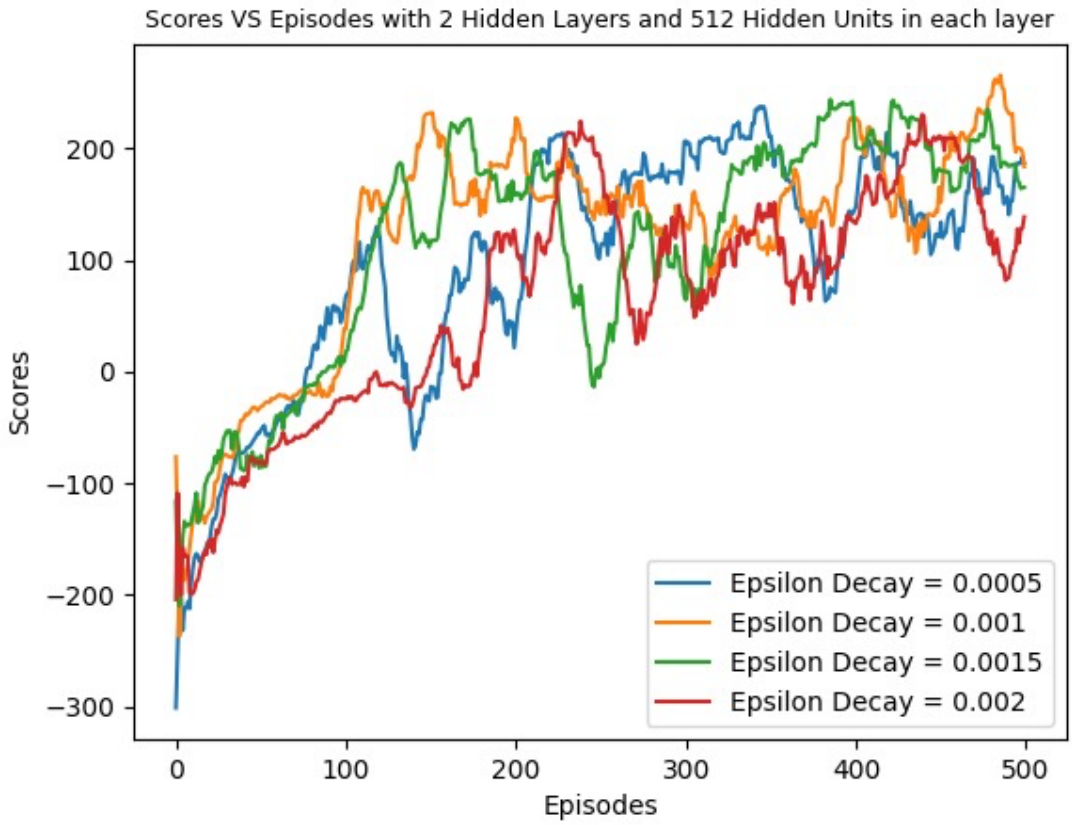

Fig.24 Scores for different decaying rates of epsilon

From the above figure, we can clearly say that the scores for different values of epsilon decay were similar, and the best score was achieved for the epsilon decay value of 0.001 and the decay value of 0.0005 gave us the most range of scores.

CONCLUSION

In conclusion, we were able to train our agent successfully to play the Lunar Lander Game. After experimenting with multiple parameters we found a set of parameters which were the best possible for our model.

These are:

1. learning rate: 0.001
2. epsilon decay: 0.0005
3. Number of Neurons in each hidden layer: 512
4. Number of Hidden Layers:2

The above mentioned parameters gave us the average score of around 230 which is good because according to the openAI gym documentation the average score for the lunar lander game is 200 and our average score was above the mentioned average score.

The code for our agent is attached in the zip folder with filename GP_1qlearningLunarLander.py
and,
there are two mp4 files namely Gp1_LunarLanderTraining.mp4
and,
Gp1_LunarLanderTesting.mp4 which as the name suggests are the training and testing videos of our agent.

REFERENCES

[1] 'Develop & Compare Reinforcement Learning Algorithms with OpenAI Gym', Analytics Vidhya, Sep. 08, 2020. https://www.analyticsvidhya.com/blog/2020/09/developing-new-machine-learning-using-openai-gym/ (accessed Jul. 22, 2021).

[2] OpenAI, 'Gym: A toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com (accessed Jul. 22, 2021).

[3] S. Gadgil, Y. Xin, and C. Xu, 'Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning', arXiv:2011.11850 [cs], Nov. 2020, Accessed: Jul. 22, 2021. [Online]. Available: http://arxiv.org/abs/2011.11850

[4] Baeldung, 'Epsilon-Greedy Q-learning | Baeldung on Computer Science', Dec. 18, 2020. https://www.baeldung.com/cs/epsilon-greedy-q-learning (accessed Jul. 22, 2021).

[5] H. van Hasselt, A. Guez, and D. Silver, 'Deep reinforcement learning with double Q-Learning', in Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, Arizona, Feb. 2016, pp. 2094–2100.

[6] 'Deep Q-Learning, Part2: Double Deep Q Network, (Double DQN) | by Amber | Medium'. https://medium.com/@qempsil0914/deep-q-learning-part2-double-deep-q-network-double-dqn-b8fc9212bbb2 (accessed Jul. 22, 2021).

[7] C. Yoon, 'Double Deep Q Networks', Medium, Jul. 17, 2019. https://towardsdatascience.com/double-deep-q-networks-905dd8325412 (accessed Jul. 22, 2021).

[8] 'Playing CartPole with the Actor-Critic Method | TensorFlow Core', TensorFlow. https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic (accessed Jul. 22, 2021).

[9] A. Suran, 'Dueling Double Deep Q Learning using Tensorflow 2.x', Medium, Jul. 11, 2020. https://towardsdatascience.com/dueling-double-deep-q-learning-using-tensorflow-2-x-7bbbcec06a2a (accessed Jul. 22, 2021).

[10] https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code