Module-3

1) Implement a Q-learning algorithm for a simple grid world environment. Explain how the Q-values are updated and analyze the convergence behaviour of your implementation

Q-learning is a model free reinforcement learning algorithm used to learn the optimal action-value function for an environment. It allows an agent to learn by interacting with the environment and updating Q-values based on rewards recieved

Implementation:

```
import numpy as np, random
Q = np.zeros((16,4))
alpha = 0.1
gamma = 0.9
epsilon = 0.1
def step(s, a):
    r, c = divmod(s,4)
    if a==0 and r>0: r-=1
    if a==1 and r<3: r+=1
    if a==2 and c>0: c-=1
    if a==3 and c<3: c+=1
    ns = r*4+c
    reward = 1 if ns==15 else 0
    return ns, reward
for _ in range(500):
    s = 0
    while s!=15:
        a = random.randint(0,3) if random.random()<epsilon
        else np.argmax(Q[s])
```

```
ns, r = step(s, a)
Q[s,a] += alpha * (r + gamma * np.max(Q[ns]) - Q[s,a])
s = ns
```

Q-value update:

Q-values are updated using:

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

- The term $r + \gamma \max Q(s')$ is the target value
- α controls how fast learning happens

Convergence behaviour:

- Q-values gradually stabilize after multiple episodes
- The learned policy becomes the shortest path to goal.
- Because the grid world is small and deterministic, Q learning converges to the optimal Q^*.

2) Develop a deep Q-network (DQN) to solve a basic control problem (eg. Cartpole). Discuss how the neural network approximates the Q function & evaluate its performance to standard Q-learning.

A DQN uses a neural network to approximate the Q function instead of Q-table, allowing it solve environments with continuous state spaces like Cartpole.

Implementation:

```
import gym, torch, torch.nn as nn, torch.optim as optim
import random, numpy as np
from collection import deque
env = gym.make("Cartpole-v1")
class DQN(nn.Module):
    def __init__(self):
        super().__init__()
```

```
self.net = nn.Sequential( nn.Linear(4, 64), nn.Relu(), nn.Linear
                                                    (64,2))

def forward(self, x):
    return self.net(x)

q_net = DQN()
target_net = DQN()
target_net.load_state_dict(q_net.state_dict())
optimizer = optim.Adam(q_net.parameters(), lr = 1e-3)
memory = deque(maxlen = 20000)
gamma = 0.99
epsilon = 1.0
def select_action(state):
    if random.random() < epsilon:
        return env.action_space.sample()
    with torch.no_grad():
        return q_net(torch.FloatTensor(state)).argmax().item()
for episode in range(200):
    s = env.reset()
    done = False
    while not done:
        a = select_action(s)
        s2, r, done, _ = env.step(a)
        memory.append((s,a,r,s2,done))
        s = s2
    if len(memory) > 1000:
        batch = random.sample(memory, 64)
        s_b, a_b, r_b, s2_b, d_b = zip(*batch)
```

```
with torch.no_grad():
    target = r_b + gamma * target_net(s2_b).max(1,
                                    keepdim=True)[0] * (1-d_b)
q_values = q_net(s_b).gather(1, a_b)
loss = nn.MSEloss()(q_values, target)
optimizer.zero_grad()
loss.backward()
optimizer.step()
if episode %.10 == 0:
    target_net.load_state_dict(q_net.state_dict())
    epsilon = max(0.05, epsilon * 0.97)
```

The NN takes the 4 dimensional cartpole state & outputs 2 Q-values & backpropogation minimizes:

$$(r + \gamma \max Q_{target}(s') - Q(s,a))^2$$

Performance vs Standard Q-learning :-

- Standard Q-learning requires a discrete state space fails on Cart pole.
- DQN learns directly from continuous states & typically solves Cartpole within 200-300 episodes
- DQN also provides smoother and more stable learning.

3) Design an experiment to compare double DQN with vanilla DQN on a provided environment & Interpret the differences in their learning curves & stability.

Double DQN vs Vanila DQN

- Environment : CartPole-v1
- Models compared : a) Vanila DQN b) Double DQN

Both use : same neural network, replay buffer, batch size, optimizer, ϵ greedy schedule & training steps

Only difference : target calculation.

Vanilla DQN target :

```
with torch.no_grad():
    target = r + gamma * target_net(s2).max(1, keepdim=True)[0] * (1-done)
```

Double DQN target :

```
with torch.no_grad():
    a_max = online_net(s2).argmax(1, keepdim=True)
    target = r + gamma * target_net(s2).gather(1, a_max) * (1-done)
```

Learning Curve:

DQN :-

- Learns quickly at first
- Shows spikes & drops
- Sometimes collapses after good performances

Double DQN:

- Slightly slower start
- Much smoother learning
- Higher & more stable final reward.

Interpretation:-

Double DQN provides more stable training and better long-term performances bcz it uses the online network to choose the action and the target network to evaluate it. This experiment shows that double DQN > DQN.

4) Extend the Q learning agent to use a dueling network architecture. Analyze how this architecture improves the agent's policy learning.

A dueling network is an improvement and standard Q-networks, instead of predicting Q-values directly, it

splits the network into value and Advantage streams:

$$Q(s,a) = V(s) + A(s,a) - \max A(s,a)$$

Implementation:

```
import toch.nn as nn
import torch
class DuelingDQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.shared = nn.Sequential(nn.Linear(state_dim,
                                              128),
        nn.ReLU(),)
        self.value= nn.Sequential(nn.Linear(128,1)
        self.adv= nn.Sequential(nn.Linear(128, actiondim)
    def forward(self,x):
        h = self.shared(x)
        V = self.value(h)
        A = self.adv(h)
        return V+A-A.max(dim=1, keepdim=True)[0]
```

How it improves Policy learning:

- Learns State value Independently
- Advantage Stream Learns Action Differences
- Faster and more stable learning
- Better policy in Hard states

A dueling network improves Q-learning by seperating state value and action advantage, resulting in:

faster policy learning.

5. Use the concepts of multi step learning & hierarchal learning propose and implement an improvement to a base R.L agent. Diss the effects.

Improve a basic Q-learning agent by adding:

1. n-step learning and hierarchal choice

1. Add n-step buffer + return

```
n_step_buffer.append((s, a, r, s_next, done))

if len(n_step_buffer) == n or done:
    R = 0
    for i, (_, _, ri, _, di) in enumarate(n_step_buffer):
        R += (gamma ** i) * ri
        if di: break
    s0, a0, _, _, d0 = n_step_buffer[0]
    s_n = n_step_buffer[-1][3]
    replay_buffer.append((s0, a0, R, s_n, d0))
    n_step_buffer.pop(0)

target = R + (gamma **n) * maxQ(s_n) * (1-done)
```

Heirarchal option Selection:

```
if steps_left == 0:
    option = random_choice([0, 1])
    steps_left = k
if option == 0:
    action = argmax_Q(s)
else:
    action = random_action()

steps_left -= 1
```

n-step learning :-

Faster reward propogation → quicker learning

Lower variance than 1-step TD → more stable updates.

hierarichal learning :-

High level decision reduce action noise → smoother behaviour.

Improves generalization to longer horizon tasks.