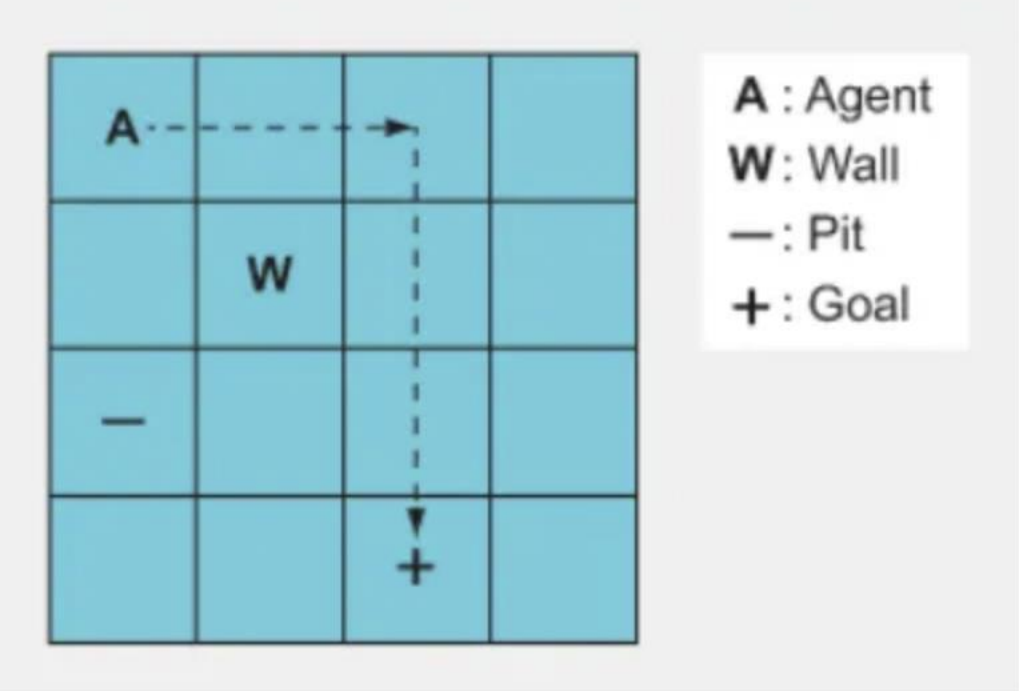
Project 2 Report

1. Maze Description: Design your own grid world example and describe it at the beginning of the report.

A:

It is a 4x4 matrix.

The agent, marked as 'A' in the picture showing the simplified form of Gridworld, has to find the shortest path to the destination tile, denoted by '+', while avoiding obstacles, signified by '–' and Wall W.



In Gridworld, the state of the game is represented by a tensor that describes where each element is located on the grid. The goal is to teach a neural network to play Gridworld from scratch with proficiency. The agent is aware of the board's current configuration. It has four options for possible movements: up, down, left, or right. The agent's rewards reflect the consequences of its actions: a regular motion earns a reward of -1, but actions that cause it to collide with a wall or fall into a pit cost it a reward of -10. If the objective is accomplished, a positive reward of +10 is awarded.

2. Problem Formulation: Define your states, actions, and rewards.

A:

States:

- Player: (0,3)
- Goal: (0,0)
- Pit: (0,1)
- Wall: (1,1)

```
#Setup static pieces
self.board.components['Player'].pos = (0,3) #Row, Column
self.board.components['Goal'].pos = (0,0)
self.board.components['Pit'].pos = (0,1)
self.board.components['Wall'].pos = (1,1)
```

Action:

There are four actions at every state.

- Up
- Down
- Left
- Right

Rewards:

- Step: -1
- Pit: -10
- Goal: 10

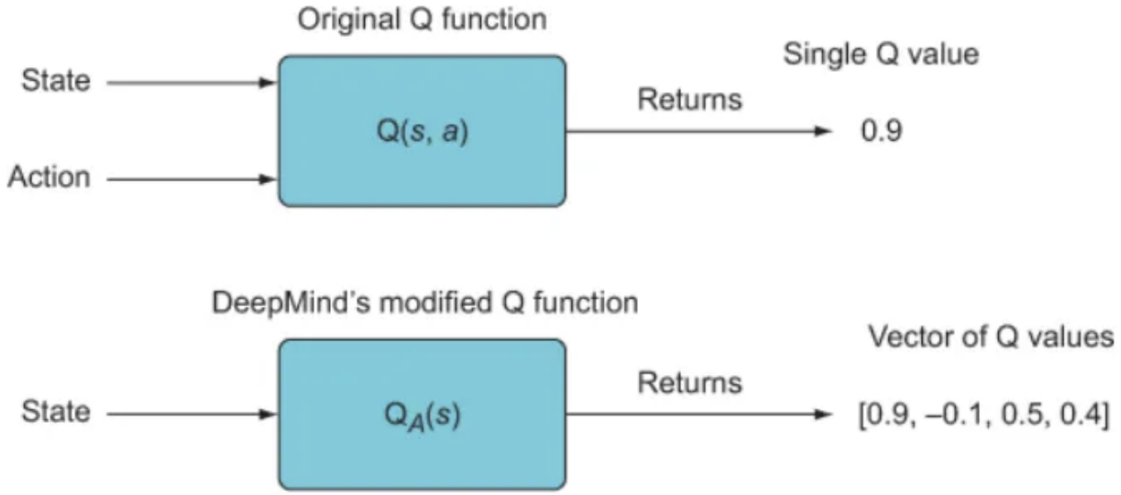
3. Q Network Design: Design and implement your Q network.

A:

Getting the Q value:

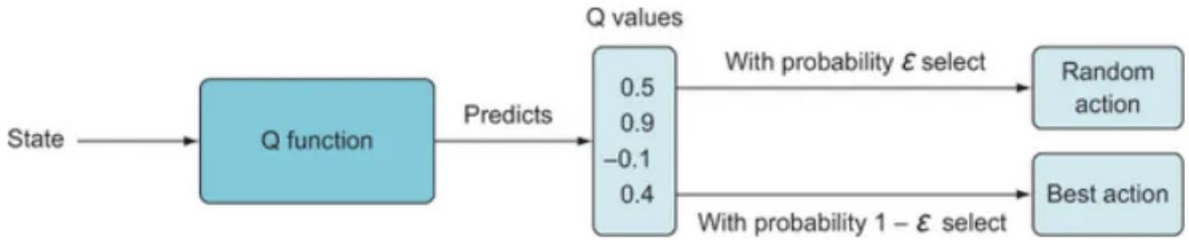
The network gets the current state S(t) and a probable action in the normal Q-learning method, and it outputs a single predicted value. The method would necessitate the network to develop four distinct predictions, each with a different input action, due to our situation's four possible actions. This would result in a large increase in processing demands. DeepMind suggested a change to simplify this: the network generates a

probability distribution covering every action that might be taken in a single forward pass. As a result, a four-element vector indicating the likelihood of each potential action is the network output. Next, in any given situation, the course of action with the highest probability is the most advantageous. This creative modification is explained in the accompanying figure.



Select the best action:
Using the epsilon-greedy technique, an action is chosen once the network's anticipated probability distribution for each state has been obtained. With this method, an epsilon value represents the probability of choosing to explore at random instead of following the network's projected probabilities. In contrast, the action associated with the greatest predicted Q value is selected with a probability of (1 - epsilon), striking a balance between the utilization of known information and the necessity for exploration.

We will eventually arrive at a new state S(t+1) with the observed reward R(t+1) once the action A(t) has been chosen. The expected value we should have received at S(t) is predicted by Q(S(t),A(t)).We now wish to post an update on our network indicating the actual benefit it obtained for following its recommendation.



Get the reward at (t+1):

The network is then run for the next state, (s(t+1)), in order to get the expected value, or (Q(s(t+1), a)). This value makes the best course of action clear, which is the largest anticipated Q value among all feasible actions for the new state.

Update the network:
Using a loss function such as mean-squared error, we will proceed with a single training iteration in order to minimize the difference between our network's forecast and the desired prediction.

4. Pseudo Code: Provide the pseudo code in the report.

A:

Pseudo code:

1. Create a loop to iterate through a specified number of epochs.
2. At the beginning of each epoch, initialize the state of GridWorld.
3. Establish a while loop to track the progress of the game episode.
4. Execute the Q-network forward pass to obtain the evaluation Q value for the current state.
5. Apply the epsilon-greedy strategy to select an action.
6. Perform the action determined in the previous step to transition to a new state, s', and receive a reward, $r(t+1)$.
7. Conduct another forward pass of the Q-network at the new state, s', to find the maximum Q value, denoted as maxQ.
8. Compute the target value for training the network as $r(t+1) + \gamma \times \text{maxQ}(A(s(t+1)))$, where γ is a discount factor ranging between 0 and 1. If the action taken concludes the game, rendering $s(t+1)$ non-existent, the term $\gamma \times \text{maxQ}(A(s(t+1)))$ is considered invalid and set to 0, making the target simply $r(t+1)$.
9. Since the network produces four outputs and we only need to update the output associated with the executed action, adjust the target output vector to match the initial output vector, with the exception of modifying the element corresponding to our chosen action to reflect the value derived from the Q-learning equation.
10. Train the model using this adjusted target for the step and repeat from step 2 to 9 for each iteration.

Model code:

```
import numpy as np
import torch
from Gridworld import Gridworld
```

```
import random
from matplotlib import pylab as plt

l1 = 64
l2 = 150
l3 = 100
l4 = 4

model = torch.nn.Sequential(
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3,l4)
)
loss_fn = torch.nn.MSELoss()
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

gamma = 0.9
epsilon = 1.0
```

Training code:

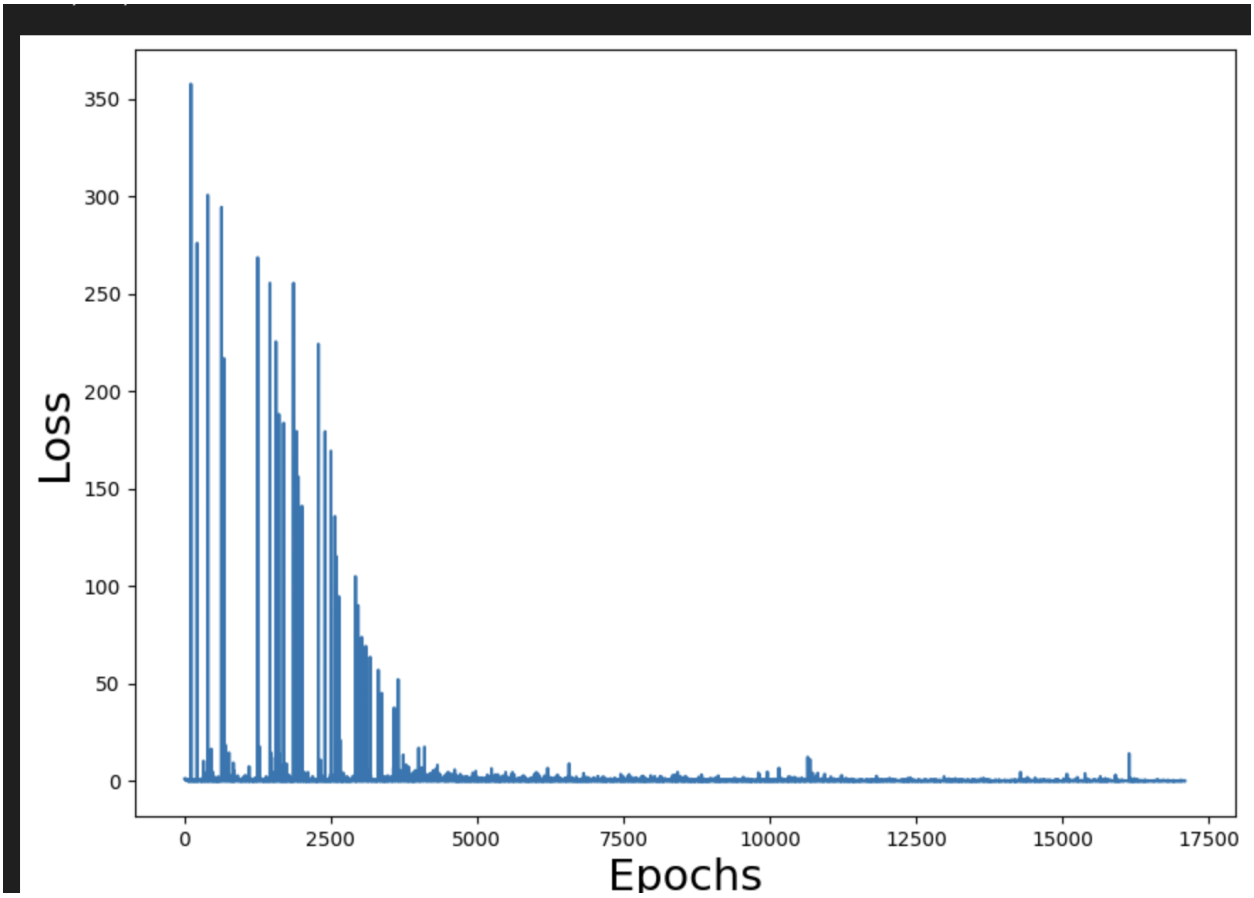
```
epochs = 1000
losses = []
for i in range(epochs):
    game = Gridworld(size=4, mode='static')
    state_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
    state1 = torch.from_numpy(state_).float()
    status = 1
    while(status == 1):
        qval = model(state1)
        qval_ = qval.data.numpy()
        if (random.random() < epsilon):
            action_ = np.random.randint(0,4)
        else:
            action_ = np.argmax(qval_)

        action = action_set[action_]
        game.makeMove(action)
        state2_ = game.board.render_np().reshape(1,64) + np.random.rand(1,64)/10.0
        state2 = torch.from_numpy(state2_).float()
        reward = game.reward()
        with torch.no_grad():
```

```
            newQ = model(state2.reshape(1,64))
        maxQ = torch.max(newQ)
        if reward == -1:
            Y = reward + (gamma * maxQ)
        else:
            Y = reward
        Y = torch.Tensor([Y]).detach()
        X = qval.squeeze()[action_]
        loss = loss_fn(X, Y)
        optimizer.zero_grad()
        loss.backward()
        losses.append(loss.item())
        optimizer.step()
        state1 = state2
        if reward != -1:
            status = 0
    if epsilon > 0.1:
        epsilon -= (1/epochs)
```

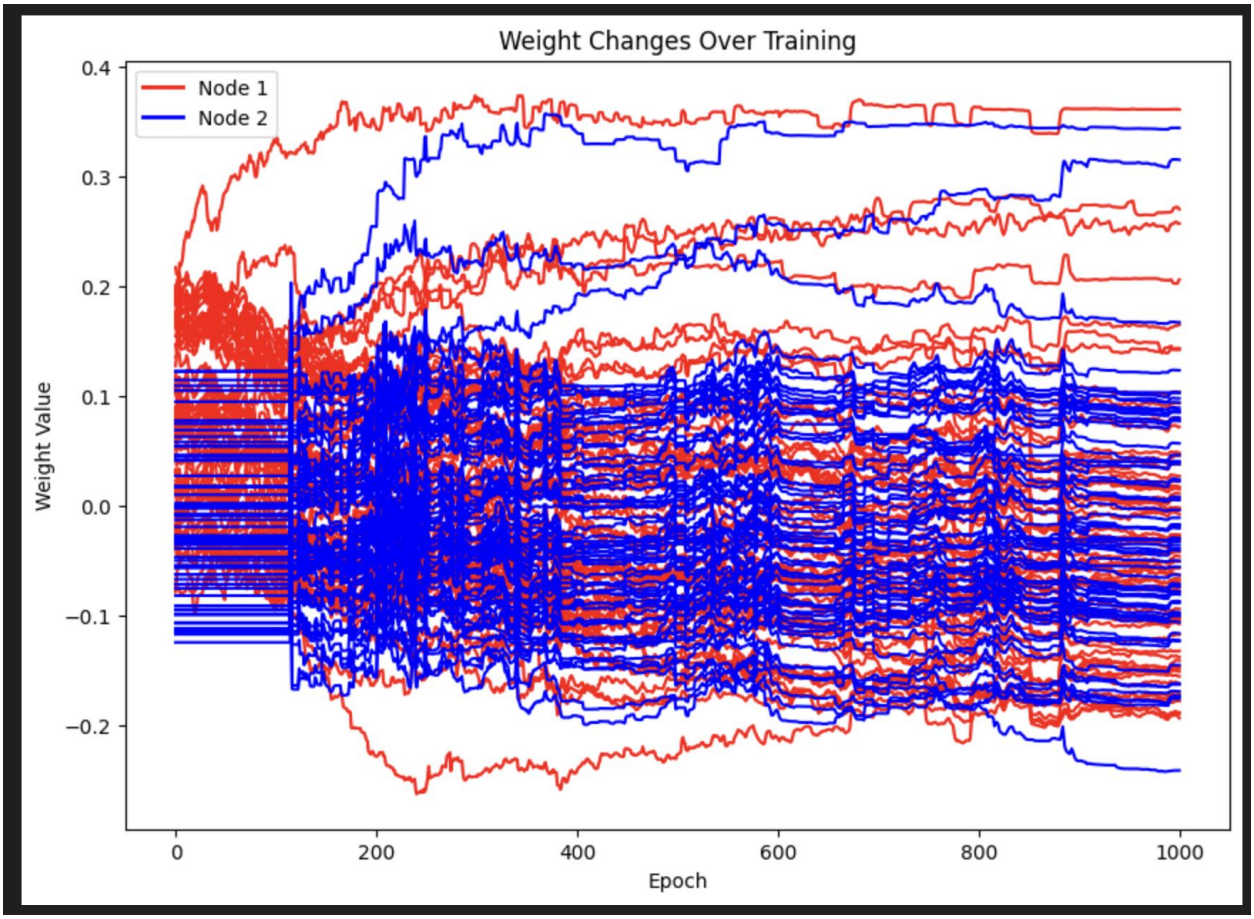
5. Results and Discussions: Show the convergence process of mean square error (objective function) and the weights trajectories.

A:

Loss Function:



The weights trajectories:



6. Reference: cite all your reference here.

https://towardsdatascience.com/part-1-building-a-deep-q-network-to-play-gridworld-deepminds-deep-q-networks-78842007c631
https://github.com/NandaKishoreJoshi/Reinforcement_Lerning/blob/main/RL_course/Ch3_Gridworld/Gridworld.py