Project 2: Report

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

A: Maze Design:

Maze consist of 5 rows and 4 columns with 2 walls

A screen shot of a computer

Description automatically generated

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

A: States:

* + Initial: on co-ords (3,0) which is in yellow colour.
  + Goal: in (0,4) which is in green colour.

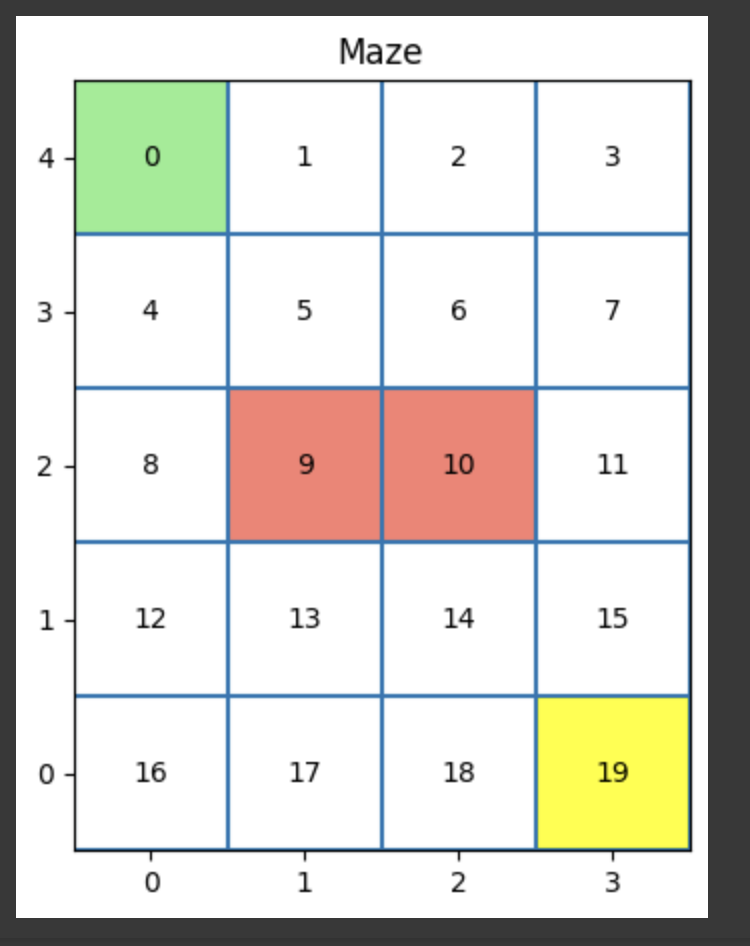
Action:

There are four actions at every state.

* + Up
  + Down
  + Left
  + Right

Rewards:

* + Step: 0
  + Wall: -5
  + Goal: 10



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

A:

## Q Network Design:

There are four layers in Q network.

* Layer 1: Input Layer: 2 nodes, which take (x,y) coordinates as input.
* Layer 2: Hidden Layer: 150 nodes.
* Layer 3: Hidden Layer: 100 nodes.
* Layer 4: Output Layer: 4 nodes, each correspondence to each action

With learning rate = 1e-3 and adam optimizer is used.

A screenshot of a computer program

Description automatically generated

## Implementation:

This model is implemented in the function “update\_Q\_table\_Q\_Network”. In the below line.

new\_state\_Q\_values = self.model(torch.from\_numpy(np.array([new\_state\_x, new\_state\_y])).float())

Which returns 4 outputs, which are the Q values for each action. For that, the Q values of each action will fetch the maximum of Q values and updates the Q table.

A screen shot of a computer program

Description automatically generated

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

A: For each episode, the model will evaluate and generate the 4 Q values for each action, taking the position of the new state as input.

A diagram of a light source

Description automatically generated

Update\_Q\_table\_Q\_Network function helps us to update the Q table using the neural network model and return the loss.

def update\_Q\_table\_Q\_Network(self, new\_state: int):

"""

Function that applies the RL update function

"""

# Getting the next\_state's reward

reward = self.reward\_dict[new\_state]

# Saving the current Q value

current\_Q = self.Q[self.past\_state][self.past\_action]

# If the new state is the terminal state or the wall state, then the max\_Q is 0

max\_Q = 0

Y = reward

# Else we get the max Q value for the new state

if new\_state != self.goal\_state:

with torch.no\_grad():

new\_state\_x, new\_state\_y = self.get\_state\_coords(new\_state)

new\_state\_Q\_values = self.model(torch.from\_numpy(np.array([new\_state\_x, new\_state\_y])).float())

max\_Q = torch.max(new\_state\_Q\_values) #M

Y = reward + (self.gamma \* max\_Q)

# Updating inplace the Q value

self.Q[self.past\_state][self.past\_action] = current\_Q + self.alpha \* (reward + self.gamma \* max\_Q - current\_Q)

Y = torch.Tensor([Y]).detach()

X = torch.from\_numpy(self.Q[self.past\_state]).squeeze()[self.past\_action]

loss = self.loss\_fn(X, Y) #P

clear\_output(wait=True)

self.optimizer.zero\_grad()

loss = Variable(loss, requires\_grad = True)

loss.backward()

self.optimizer.step()

return loss

Model code:

l1 = 2

l2 = 150

l3 = 100

l4 = self.num\_actions

self.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)

)

Training episodes code:

def train\_episodes\_Q\_Network(self, num\_episodes: int):

"""

Function that trains the agent for one episode

"""

# Calculating the episode number to end the training

end\_episode = self.current\_episode + num\_episodes - 1

# Creating a dir to store the images

os.makedirs(self.path\_to\_images\_V, exist\_ok=True)

os.makedirs(self.path\_to\_images\_Q, exist\_ok=True)

# If the directory for the first episode is not empty, we delete the files inside it

if len(os.listdir(self.path\_to\_images\_V)) > 0:

for file in os.listdir(self.path\_to\_images\_V):

os.remove(os.path.join(self.path\_to\_images\_V, file))

if len(os.listdir(self.path\_to\_images\_Q)) > 0:

for file in os.listdir(self.path\_to\_images\_Q):

os.remove(os.path.join(self.path\_to\_images\_Q, file))

episode\_weights = {}

# Moving the agent until we reach the goal state

while self.current\_episode != end\_episode:

print(f"Episode {self.current\_episode}")

print(self.reward\_dict)

self.move\_agent\_Q\_Network(num\_episodes)

plot\_V\_Table(

S=self.S,

Q=self.Q,

goal\_coords=self.goal\_coords,

wall\_coords=self.wall\_coords,

start\_coords=self.start\_coords,

title=f"V Table at Episode {self.current\_episode}",

filename=f"{self.path\_to\_images\_V}/episode\_{self.current\_episode}.png"

)

plot\_Q\_Table(

S=self.S,

Q=self.Q,

goal\_coords=self.goal\_coords,

wall\_coords=self.wall\_coords,

start\_coords=self.start\_coords,

title=f"Q Table at Episode {self.current\_episode}",

filename=f"{self.path\_to\_images\_Q}/episode\_{self.current\_episode}.png"

)

self.capture\_weight\_stats()

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

**Loss Function:**

A graph with a line

Description automatically generated

6. Reference: cite all your references here.

<https://towardsdatascience.com/part-2-building-a-deep-q-network-to-play-gridworld-catastrophic-forgetting-and-experience-6b2b000910d7>

<https://github.com/Eligijus112/rl-snake-game>