Lesson 28 Reinforcement Learning Last Update: June 2, 2021

Conventions used in these lessons:

bold - headings, keywords, code

italics - code syntax

underline - important words

Machine Learning

Machine learning refers to a class of algorithms that promises to improve automatically based on experience

We can broadly divide machine learning into three categories depending upon the feedback available for the algorithm to learn over time:

- Supervised Learning: The objective of supervised learning is to learn a function that can map the input to output, exploiting from a labeled set of training data.
- Unsupervised Learning: In contrast, unsupervised learning is about learning undetected patterns in the data, through exploration without any pre-existing labels.
- Reinforcement Learning: Finally, the goal of reinforcement learning is to maximize the cumulative reward by taking actions in an environment, balancing between exploration and exploitation.

Neural networks consist of many simple processing nodes that are interconnected and loosely based on how a human brain works**.** We typically arrange these nodes in layers and assign weights to the connections between them. The objective is to learn these weights through several iterations of feed-forward and backward propagation of training data through the network.

We typically construct these networks to solve sophisticated problems and categorize them as deep learning. When we apply deep learning in the context of reinforcement learning, we often term that as deep reinforcement learning.

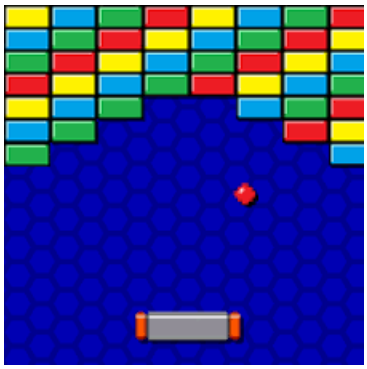
Reinforcement Learning

Reinforcement learning is about an autonomous agent taking suitable actions to maximize rewards in a particular environment. Over time, the agent learns from its experiences and tries to adopt the best possible behavior**.**

Reinforcement learning is an autonomous, self-teaching system that essentially learns by trial and error. It performs actions with the aim of maximizing rewards, it is learning by doing in order to achieve the best outcomes. This is similar to how we learn things like riding a bicycle where in the beginning we fall off lots of times and make many erratic moves, but over time we use the feedback of what we learned and use this knowledge to fine-tune our actions and eventually learn how to ride a bicycle. The same is true when computers use reinforcement learning, they try different actions, learn from the feedback whether that action delivered a better result, and then reinforce the actions that worked. A reinforcement learning system will rework and modifying its algorithms autonomously over many iterations until it makes decisions that deliver the best result.

A good example of using reinforcement learning is a robot learning how to walk. The robot first tries a large step forward and falls. The outcome of a fall with that big step is a data point the reinforcement learning system responds to. Since the feedback was negative, a fall, the system adjusts the action to try a smaller step. The robot is able to move forward. This is an example of reinforcement learning in action. Another good example is a robot vacuum cleaner navigating a room for cleaning, that would bump into many walls and furniture obstacles before it learned all the paths.

One of the most fascinating examples of reinforcement learning in action is was Google's Deep Mind tool applied to the classic Atari computer game Break Out. The goal (or reward) was to maximize the score and the actions were to move the bar at the bottom of the screen to bounce the playing ball back up to break the bricks at the top of the screen.



The algorithm makes lots of mistakes at the beginning but quickly improves to where it would beat even the best human players.

Deep Learning

Deep learning is essentially an autonomous, self-teaching system in which use existing data to train algorithms to find patterns and then use that to make predictions about new data. For example, you might train a deep learning algorithm to recognize cats on a photograph. You would do that by feeding it millions of images that either contains cats or not. The program will then establish patterns by classifying and clustering the image data (e.g. edges, shapes, colors, distances between the shapes, etc.). Those patterns will then inform a predictive model that is able to look at a new set of images and predict whether they contain cats or not, based on the model it has created using the training data.

Deep learning algorithms use layers of artificial neural networks which mimic the network of neurons in our brain. This allows the algorithm to perform various cycles to narrow down patterns and improve the predictions with each cycle.

A great example of deep learning in practice is Apple's Face ID. When setting up your phone you train the algorithm by scanning your face.

Each time you log on using e.g. Face ID, the TrueDepth camera captures thousands of data points which create a depth map of your face and the phone's inbuilt neural engine will perform the analysis to predict whether it is you or not.

Deep Reinforcement Learning

Deep learning and reinforcement learning are both systems that learn autonomously. The difference between them is that deep learning is learning from a training set and then applying that learning to a new data set, while reinforcement learning is dynamically learning by adjusting actions based in continuous feedback to maximize a reward.

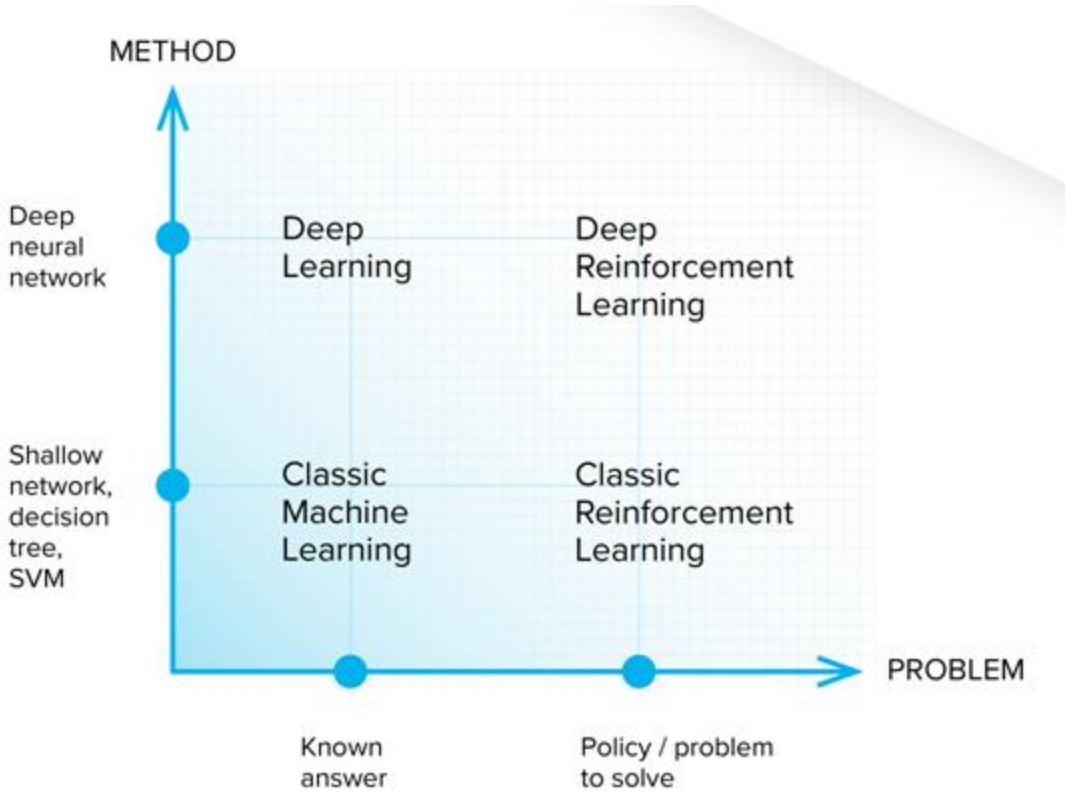
By combining deep learning with reinforcement aids the reinforcement to remember and predict the patterns to a higher degree of accuracy.

Methods to Solve Problems

If we know the answer to a problem we can use classical machine learning techniques such as classifications or we can use Deep Neural Networks,

Classical machine learning is much faster but less accurate. Deep Neural networks rake much longer to train but needs lots of training data but more accurate.

If we have a problem with no known outcome then classical reinforcement is much faster but less accurate. Whereas deep reinforcement learning is much slower but more accurate. Evert thing in life always has trades offs.



Reinforcement Learning Example

We want to navigate a mouse in a maze from start to finish containing obstacles to avoid. The mouse may move up, down, left or right.

S			
	O	O	
A	O	O	
			G

S = Start
G = Goal
O = Obstacle
A = Agent

reinforcement terminology:

The maze is known as the environment
The mouse is known as the Agent
The up, down left and right movements are known as Actions
Each square travelled to is known as a State

Reinforcement theory:

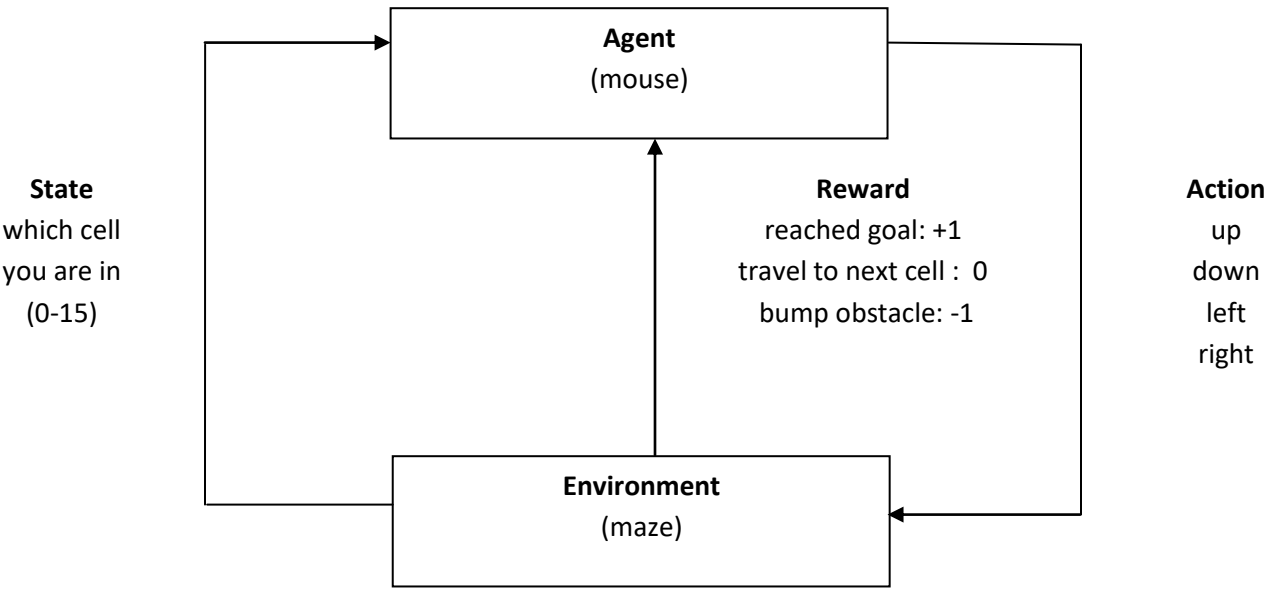
Reinforcement learning is about an autonomous agent taking suitable actions to maximize rewards in a particular environment. Over time, the agent learns from its experiences and tries to adopt the best possible behavior.

In the case of reinforcement learning, we limit human interaction to changing the environment states, and the system of rewards and penalties. This environment setup is known as the **Markov Decision Process**. (MDP)

Environment for our maze example

The Agent mouse moves to different states accordingly to the up, down left or right action taken in the maze environment. If the agent reached the goal it gets an a reward . if the agent hits an obstacle it gets punished if the agent moves to a free cell then a small reward or no reward may be made.

Training Environment:



The maze is the environment that the autonomous agent is travelling to the goal a piece of cheese, Each up, down left and right movement is an action to get to the next cell, When the goal is reached the mouse is reward, if the modes bump into obstacles then the mousse receives a penalty if the mode successful y gets to another cell when the modes does not gain or looses any points. The mouse may make many trial and error journeys until it knows the path inside out by heart.

Markov Decision Processes

The Markov Decision Processes uses states, action, rewards and mechanism to move from once state to another depending on the action taken.

A Markov Decision Process (MDP) model contains:
- A set of possible world states S
- A set of possible actions A
- A real valued reward function R(s,a)
- A description T of each action's effects in each state.

We assume in the Markov Property the effects of an action taken in a state depend only on that state, and not on the prior history.

Representing Actions

Deterministic Actions:

- T : For each state and action we specify a new state.

Stochastic Actions:
- T: For each state and action we specify a probability distribution over next states. Represents the distribution P(s' |s,a).

Representing Solutions
A policy π is a mapping from S to A

Following a Policy

Following a policy π:
1. Determine the current states
2. Execute action π(s)
3. Goto step 1.

Assumes full observability: the new state resulting from executing an action will be known to the system

Value Functions

A value function : represents the expected objective value obtained following policy from each state in S .

Value functions partially order the policies,
•but at least one optimal policy exists, and
•all optimal policies have the same value function, v*

Bellman Equations

Bellman equations relate the value function to itself via the problem dynamics. For the discounted objective function,

$$V_{\pi}(s) = R(s, \pi(s)) + \sum_{s' \in S} T(s, \pi(s), s') \cdot \gamma \cdot V_{\pi}(s')$$

$$V^*(s) = \mathbf{MAX}_{a \in A} \left(R(s, a) + \sum_{s' \in S} T(s, a, s') \cdot \gamma \cdot V^*(s') \right)$$

In each case, there is one equation per state

Types of Environment

Reinforcement learning primarily consists of two types of environments:

- Deterministic: This refers to the case where both **the state transition model and reward model are deterministic functions**. Simply put, an agent can expect the same reward and next state if it repeats an action in a particular state.

- Stochastic: Stochastic refers to something that has a random probability of occurrence. Within such an environment, if **an agent takes action in a state repeatedly, they cannot be guaranteed to receive the same reward or the next state.**

Types of Reinforcement Learning

There are generally two types of reinforcement learning:

- Model-Based: In a model-based algorithm, the **agent uses experience to construct an internal model of the transitions and immediate outcomes in the environment,** and refers to it to choose appropriate action. The agent can externally receive the state transition and reward probability functions.

- Model-Free: In contrast, in a model-free algorithm, the **agent uses experience to learn the policy or value function directly without using a model** of the environment. Here, the agent only knows about the possible states and actions in an environment and nothing about the state transition and reward probability functions.

Value Functions and Policy

The reward is immediate feedback that an agent receives from the environment for an action that it takes in a given state. Moreover, the agent receives a series of rewards in discrete time steps in its interactions with the environment.

The objective of reinforcement learning is to maximize this cumulative reward, which we also know as value. The strategy that an agent follows is known as policy, and the policy that maximizes the value is known as an optimal policy.

Reinforcement learning is a branch of machine learning dedicated to training agents to operate in an environment, in order to maximize their utility in the pursuit of some goals.

Its underlying idea, states Russel, is that intelligence is an emergent property of the interaction between an agent and its environment. This property guides the agent's actions by orienting its choices in the conduct of some tasks.

We can say, analogously, that intelligence is the capacity of the agent to select the appropriate strategy in relation to its goals. Strategy, a teleologically-oriented subset of all possible behaviors, is here connected to the idea of "policy".

A policy is, therefore, a strategy that an agent uses in pursuit of goals. The policy dictates the actions that the agent takes as a function of the agent's state and the environment.

Formally**,** the notion of value in reinforcement learning is presented as a value function**:**

Here, the function takes into account the discounted future rewards starting in a state under a given policy**.** We also know this as the state-value function of this policy. The equation on the right side is what we call a Bellman equation, which is associated with optimality conditions in dynamic programming.

3.5. Q-value and Q-learning

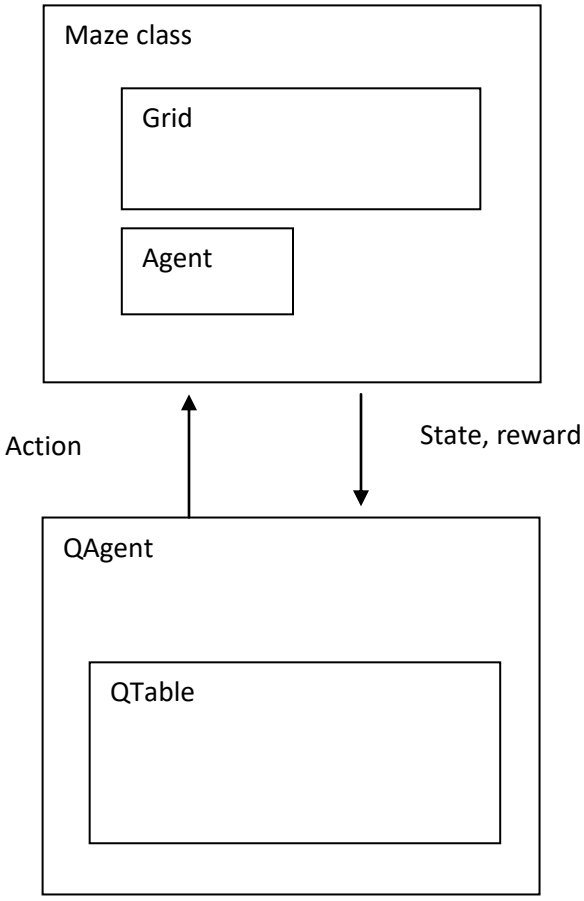
Q-value is a measure of the long-term return for an agent in a state under a policy, but it also takes into account the action an agent takes in that state. The basic idea is to capture the fact that the same action in different states can bare different rewards:

Here the function creates a map of the state and action pairs to the rewards. We also know this as the action-value function for a policy.

Q-value is a measure we use in Q-learning, which is one of the main approaches we use toward model-free reinforcement learning**.** Q-learning emphasizes how useful a given action is in gaining some future reward in a state under a policy.

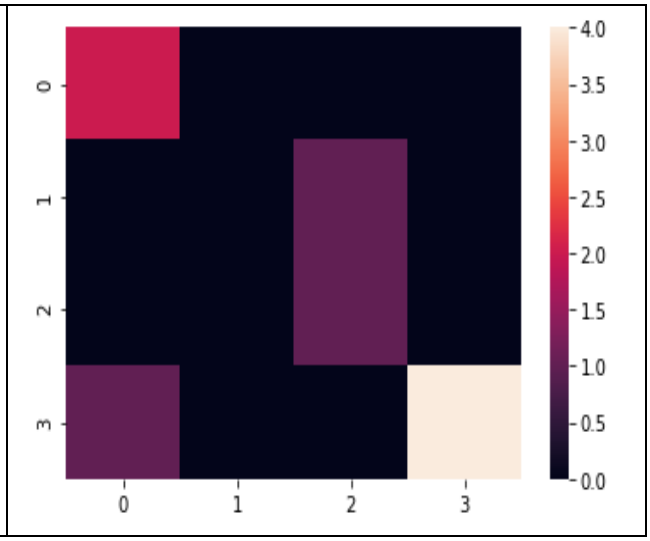
4. Implementing Reinforcement Learning

Now that we've covered enough basics, we should be able to attempt to implement reinforcement learning. We'll be implementing the q-learning algorithm for this tutorial.



1. Test Environment

START			
		WALL	
		WALL	
			GOAL



The rules of this game are:

- The grid consists of 16 tiles set up 4×4
- There is a starting tile (S) and a goal tile (G)
- Some tiles are walls
- Movement of the agent is largely uncontrollable
- The agent is rewarded for finding a walk able path from the starting tile to the goal tile

We'll be using this environment to test the reinforcement learning algorithm we're going to develop in the subsequent sections.

Maze class

The maze class store the 2d grid maze, and the agent coordinates. We display the maze in a heat map. A step function is used to update the agent position based on the actions received, The agent current position is the current state. For operation convenience. the state is stored as row and column coordinates by state number. Here is the maze code:

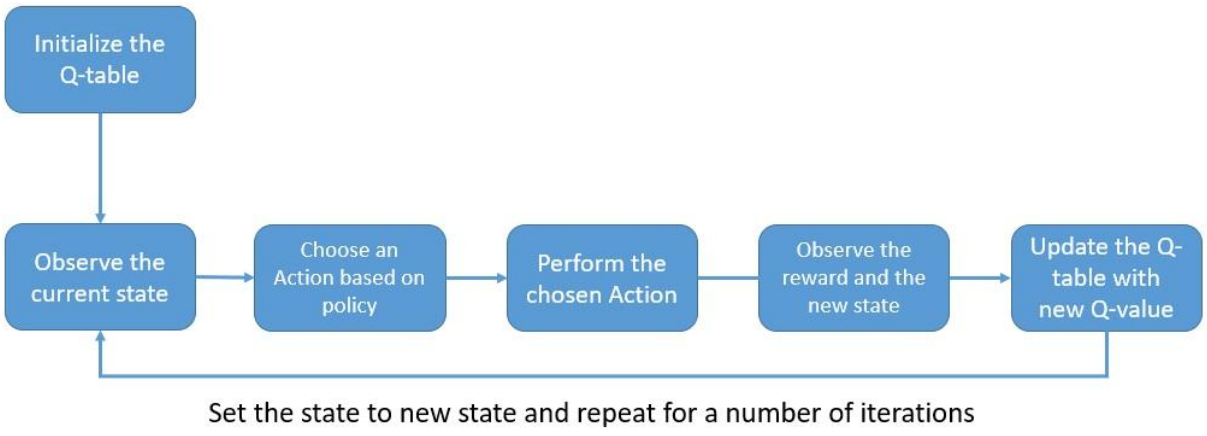
```
"""
Qtable_maze.py
Reinforcement learning
Finding a goal in A Maze
"""
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns
import time

# index constants
ROW = 0
COL = 1

# maze constants
OPEN = 0
WALL = 1
OCCUPIED = 2
START = 3
GOAL = 4

# actions
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3

"""
maze class
stores maze grid and agent coordinates
"""
class Maze:

    # initialize maze
    def __init__(self,grid,start_row,start_col,goal_row, goal_col):

        self.grid = grid
        self.rows = len(grid)
        self.cols = len(grid[0])
        self.start_row=start_row
        self.start_col=start_col
        self.goal_row=goal_row
        self.goal_col=goal_col
```

```
    self.agent_row = 0
    self.agent_col = 0

# reset maze
def reset(self):

    self.agent_row = 0
    self.agent_col = 0
    return (0,0)  # return start state

# return next state, rewards, done for input action
def step(self,action):

    # inialize
    done = False
    r = 0

    # save agent coordinance
    prev_agent_row,prev_agent_col = self.agent_row, self.agent_col

    # move agent
    if (action == UP):
        self.agent_row = max(self.agent_row-1,0)
    if (action == DOWN):
        self.agent_row = min(self.agent_row +1,self.rows-1)
    if (action == LEFT):
        self.agent_col = max(self.agent_col -1,0)
    if (action == RIGHT):
        self.agent_col = min(self.agent_col +1,self.cols-1)

    # check if agent done
    if (self.isDone( self.agent_row, self.agent_col)):
        done = True

        # goal found
        if ((self.agent_row == self.goal_row) and (self.agent_col == self.goal_col)):
            print('Target reached')
            r = 1

        # hit a wall or start
        else:
            print('hit a wall')
            self.agent_row,self.agent_col = prev_agent_row, prev_agent_col
            r = -1
```

```
        # return next state, reward, done flag and next state as a number
        next_state = (self.agent_row,self.agent_col)
        return(next_state,r,done)

    # return true if wall, start or goal found
    def isDone(self,row,col):
        done = False
      if(self.grid[row][col] == START or self.grid[row][col] == WALL or self.grid[row][col] == GOAL ):
            done = True
        return done

    # render maze as a heatmap
    def render(self):

        cells = np.copy(self.grid)
        r,c = self.agent_row,self.agent_col
        cells[r][c] = OCCUPIED
        sns.heatmap(cells)
        plt.show()
```

Q-learning Process

Here is the Q-learning process



The q-values are stored and updated in a q-table, which has dimensions matching the number of actions and states in the environment. This table is initialized with zeros at the beginning of the process.

As evident from the diagram above, the q-learning process begins with choosing an action by consulting the q-table. On performing the chosen action, we receive a reward from the environment and update the q-table with the new q-value. We repeat this for several iterations to get a reasonable q-table.

4.4. Choosing an Action

We saw earlier that **when the process starts, the q-table is all zeros**. Hence the action an agent chooses cannot be based on the q-table, it has to be random. However, as the q-table starts to get updated, the agent makes the selection of the action based on the maximum q-value for a state.

This may potentially lock the agent in repeating some of the initial decisions that were not optimal. Essentially, the agent moves from exploration to exploitation of the environment too soon. Therefore, it is necessary to introduce an action selection policy called the **e-greedy**.

Here we sample a random number, and if it happens to be less than , the action is chosen randomly. This **allows the agent random exploration, which can be very useful, especially in the initial iterations**. Of course, we slowly decay the impact of this parameter to lean on the side of exploitation as learning matures.

4.5. Updating the Q-value

The calculation of q-value follows a Bellman equation. Basically, we **keep adding a temporal difference to the current q-value for a state and action pair**.

$$Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha * \{r_t + \gamma * max_a Q(s_{t+1}, a) - Q^{old}(s_t, a_t)\}$$

o basically, we **keep adding a temporal difference to the current q-value for a state and action pair**. There are two important parameters here which are important to choose wisely:

important parameters:

- Learning rate: This is a parameter we can use to control the pace at which our algorithm can learn. We set it between o and 1 with an effective value of 0, meaning no learning at all.

- Discount factor: We saw earlier that a future reward has less importance for actions in the present. We model this using a discount factor, again set between 0 and 1.

4.6. Setting-up the Q-learning Algorithm

Actions UP, DOWN, LEFT, RIGHT

States 0 to 15 (each square cell is a state)

Qtable

Each row is state where each column is a Action

The Qtable is a 2 dimensional array accessed by row and column rather than state number. Using row and column indexes is much easier to implement since the agent is moving in x and y coordinates. The x and y agent coordinates represent the states 0 to 15.

State	row	column	UP	DOWN	LEFT	RIGHT
0	0	0				
1	0	1				
2	0	2				
3	0	3				
4	1	0				
5	1	1				
6	1	2				
7	1	3				
8	2	0				
9	2	1				
10	2	2				
11	2	3				
12	3	0				
13	3	1				
14	3	2				
15	3	3				

QAgent CLASS

QAgent class holds the Qtable and all the bellman parameters. The maze class supplies the state to the Qtable, The QAgent supplies the actions to the maze class calculates the next state for the QAgent class. Here is the QAgent class.

QAgent parameters
gamma=0.9
alpha=0.1
epsilon=0.1
num_episodes=1000

$$Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha * \{r_t + \gamma * max_a Q(s_{t+1}, a) - Q^{old}(s_t, a_t)\}$$

- Learning rate: This is a parameter we can use to control the pace at which our algorithm can learn. We set it between 0 and 1 with an effective value of 0, meaning no learning at all.
- Discount factor: We saw earlier that a future reward has less importance for actions in the present. We model this using a discount factor, again set between 0 and 1.

QAgent code

```
"""
QAgent class
stores QTable and bellman equation parameters
contains train and test methods
"""

class QAgent:

    # initialize
    def __init__(self,gamma=0.9,alpha=0.1,epsilon=0.1,num_episodes=500,maze=None):

        # qtable
        self.q_table = [[[0,0,0,0] for c in range(maze.cols)] for r in range(maze.rows)]
        self.actions = [UP,DOWN,LEFT,RIGHT]
        self.maze = maze
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.num_episodes = num_episodes
        self.action_list=['UP','DOWN','LEFT','RIGHT']
```

```
# main learning algorithm.
# loops for 'number of episodes'
# agent tries to make many movements as possible
# either randomly or based on the Q values
def train(self):

    # for each episode
    for episode in range(self.num_episodes):

        #self.maze.render()

        # start
        done = False
        state,state_number = self.maze.reset()

        # for many movements as possible
        while not done:

            # 10% choose an action randomly where Q values are 0
            if (np.random.uniform()<self.epsilon)
                 or (sum(self.q_table[state[ROW]][state[COL]]) == 0):
                action = np.random.choice(self.actions)
            # use max Q value
            else:

                # get actions for this state
                actions = self.q_table[state[ROW]][state[COL]]

                # action = max Q value (argmax = lst.index(max(lst)))
                action = actions.index(max(actions))

            # print action
            print(self.action_list[action],action)

            # get next state
            next_state,reward,done,state_number = self.maze.step(action)

            print( next_state,reward,done,state_number)

            # get current Q value
            current_Q = self.q_table[state[ROW]][state[COL]][action]

            # get next state max Q value
```

```
            next_Q = max(self.q_table[next_state[ROW]][next_state[COL]])

            # update qtable with bellman's equations and its future rewards
            self.q_table[state[ROW]][state[COL]][action]
                    += self.alpha * (reward + self.gamma * next_Q - current_Q)

            # update state
            state = next_state


    # test predictions
    def test(self):

        state, state_number = self.maze.reset()
        self.maze.render()
        time.sleep(1)
        done = False

        # loop to goal found
        while not done:

            print(state)

            # get next state
            state = (self.maze.agent_row,self.maze.agent_col)

            # get next best possible move
            actions = self.q_table[state[ROW]][state[COL]]
            action = actions.index(max(actions))

            print(self.action_list[action],action)
            state,reward,done,state_nunber = self.maze.step(action)

            # display maze
            self.maze.render()
            time.sleep(1)

        # display maze
        print(state)
        self.maze.render()
        time.sleep(1)
```

Running the program:

Here is the code to run the program:

```
#######
# Run #
#######

# maze grid
grid = [[3,0,0,0],[0,0,1,0],[0,0,1,0],[1,0,0,4]]
print(grid)

# make maze
maze = Maze(grid,0,0,3,3)

# QAgent parameters
gamma=0.9
alpha=0.1
epsilon=0.1
num_episodes=1000

# make QAgent
q = QAgent(gamma,alpha,epsilon,num_episodes,maze)

# train
print("training")
q.train()

# print q table
print(q.q_table)

# test
print("testing")
q.test()
print("done")
```

to do:

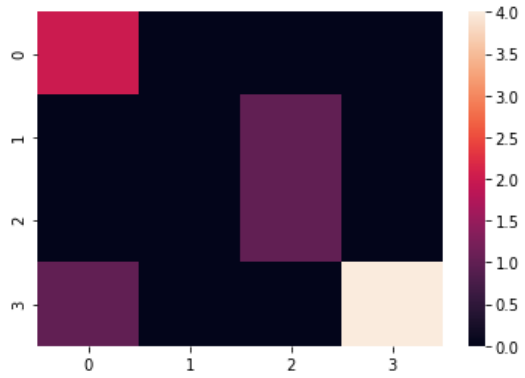
Type in or copy paste the above code and run it

You should get something like this.
Training

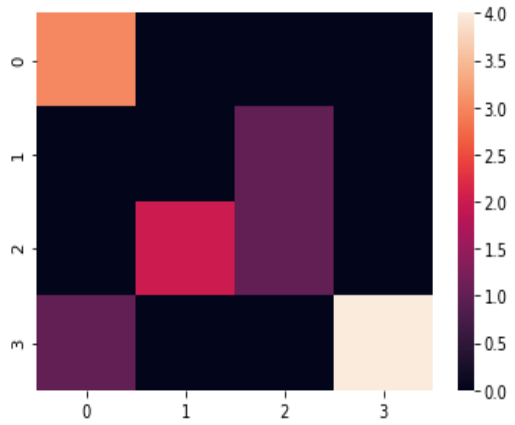
DOWN 1
(1, 1) 0 False 5
DOWN 1
(2, 1) 0 False 9
DOWN 1
(3, 1) 0 False 13
RIGHT 3
(3, 2) 0 False 14
RIGHT 3
Target reached
(3, 3) 1 True 15

[[[0.4050321248408491, 0.31774591770007055, 0.3254037304544567, 0.5904899999999977], [0.48706275491288104, 0.6560999999999979, 0.46951005532682855, 0.07260564993864563], [0.0, 0.0, 0.29634285134312316, 0.0], [0.0, 0.0, 0.0, 0.0]], [[0.034327486518508685, 0.00805924436478477, 0.05446749993875941, 0.6431604905943425], [0.44981478808352726, 0.7289999999999983, 0.3886574298716119, 0.5684926405608114], [0, 0, 0, 0], [0.0, 0.0, 0.0, 0.0]], [[0.37107067748621947, 0.0, 0.0, 0.0], [0.4464662577844367, 0.8099999999999987, 0.14550627451519554, 0.5031096461043003], [0, 0, 0, 0], [0.0, 0.1, 0.0, 0.0]], [[0, 0, 0, 0], [0.44101601504639876, 0.5266730305480968, 0.3794655349291217, 0.899999999999999], [0.4216785634828184, 0.6766716000541159, 0.5399264356454585, 0.9999999999999996], [0, 0, 0, 0]]]

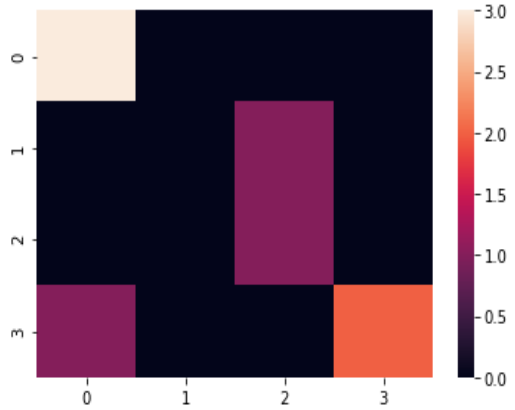
Testing



(0, 0)
DOWN 1
(1, 0)
RIGHT 3
(1, 1)
DOWN 1
(2, 1)



DOWN 1
(3, 1)
RIGHT 3
(3, 2)
RIGHT 3
Target reached
(3, 3)



DEEP LEARNING REINFORCEMENT HOMEWORK 1

Change the algorithm so when the agent bombs into the wall will shoes another action rather than restarting from state 0. Also keeps track of visited cells so it does not choose the cell again, also when the agent can travel to the goal without bumping into a wall exit training, so that the training does not rely on the number of episodes. When the training exits print out the number of episodes.

REINFORCEMENT LEARNING HOMEWORK 2

Apply Reinforcement learning for the stock prediction program of previous lessons. You may need two QTables, one for buying and cashing out and another one for selling and cashing out.

The actions would be holding buy/sell and cash out. There are manypossibiites for states

(1) a moving n* n grid, where each cell in the grid is a state, you may want a grid to represent a day of trading.

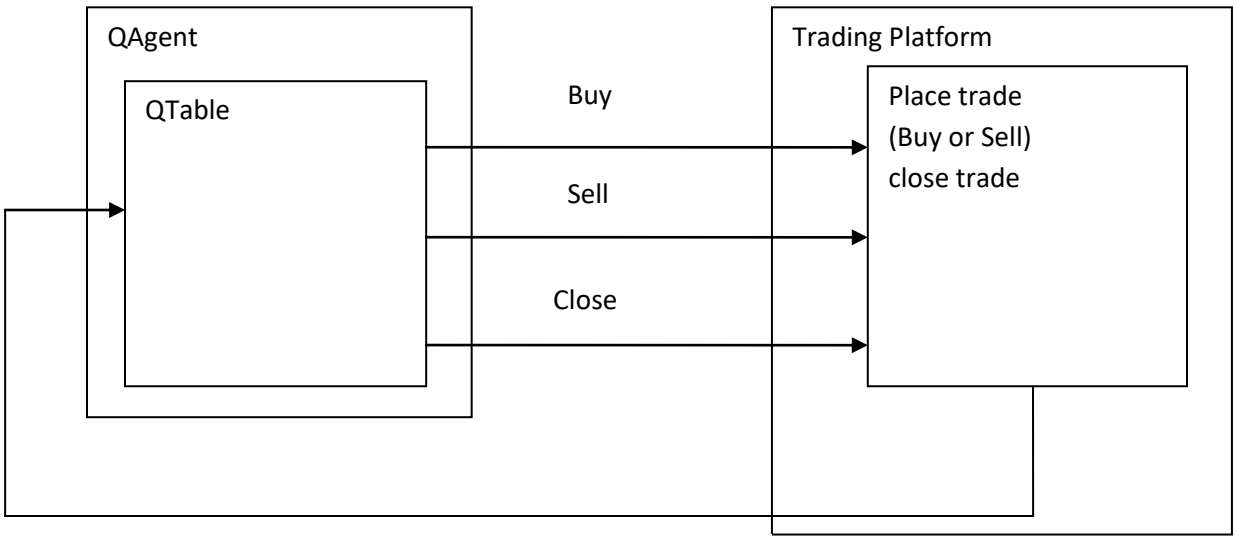
(2) moving window of prices, where each difference of price is mapped into a state number. Where -1 is a price going down, 0 piece not changing and 1 price inceasing. 3^8 = 6561 states.

(3) the price between the minimum and maximum can also represent a state.

This would be a vertical displacement of states.

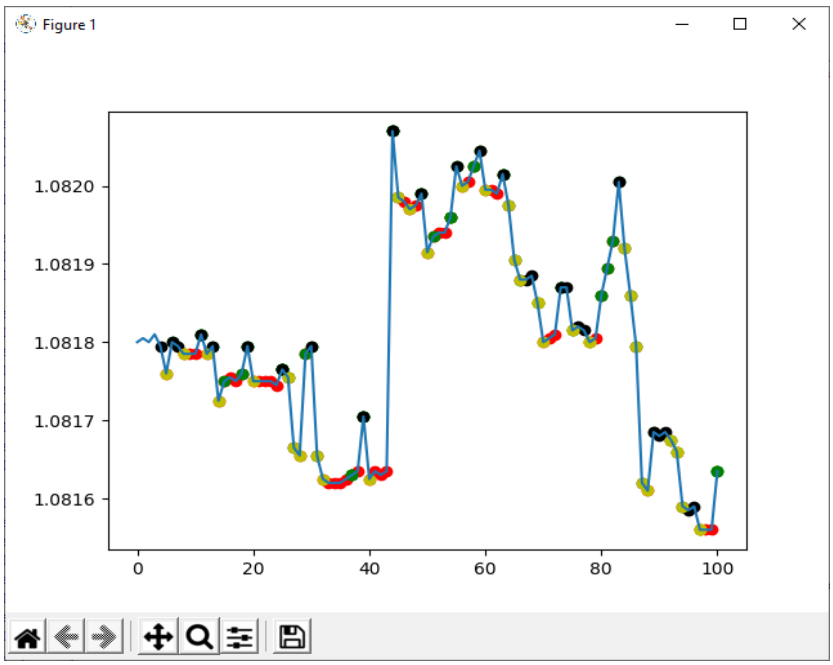
TheBuying is when the prices are changing direction. The environment is buying and selling and cashing in stocks. The system is rewarded when it is making money.

Call your py file deep_reinforcement_homework2.py

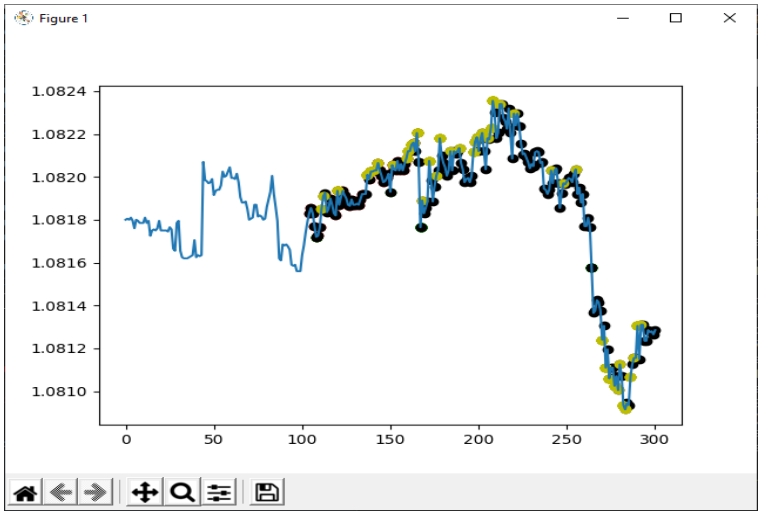


You may get something like this:

Training:



Running



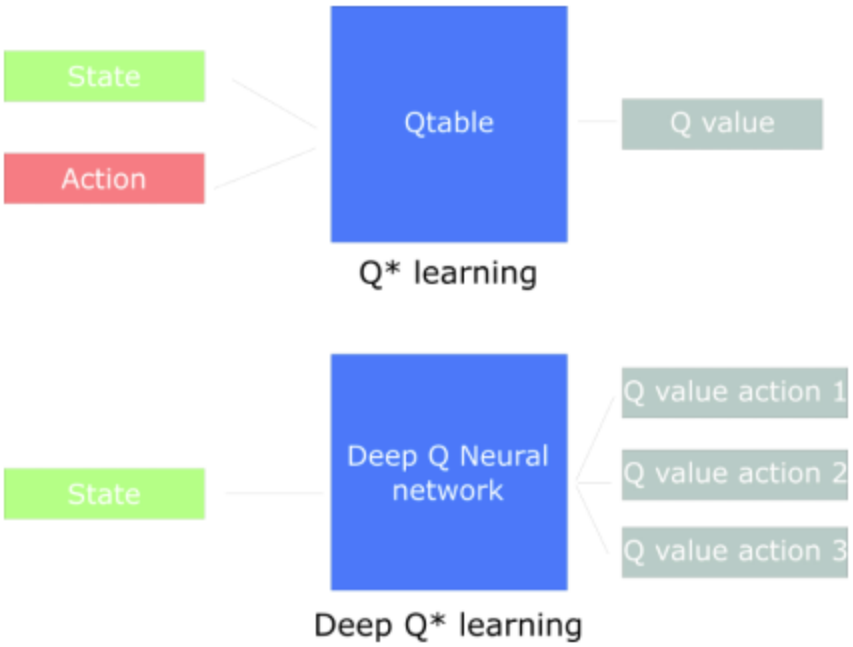
PART 2

REPLACING THE QTABLE WITH A NEURAL NETWORK

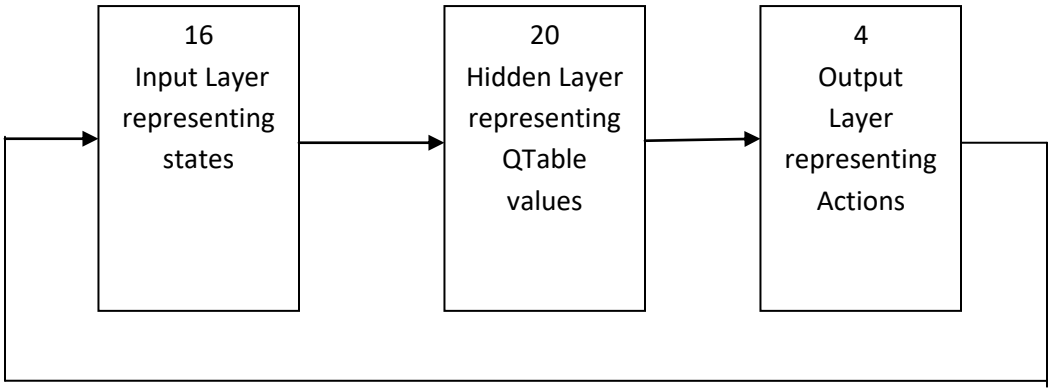
We can replace the QTable with a neural network. The advantage is more accuracy and less memory usage. The disadvantage is slower training operation, but once we are trained the response time is faster.

Difference between Reinforcement Learning and Deep Reinforcement learning

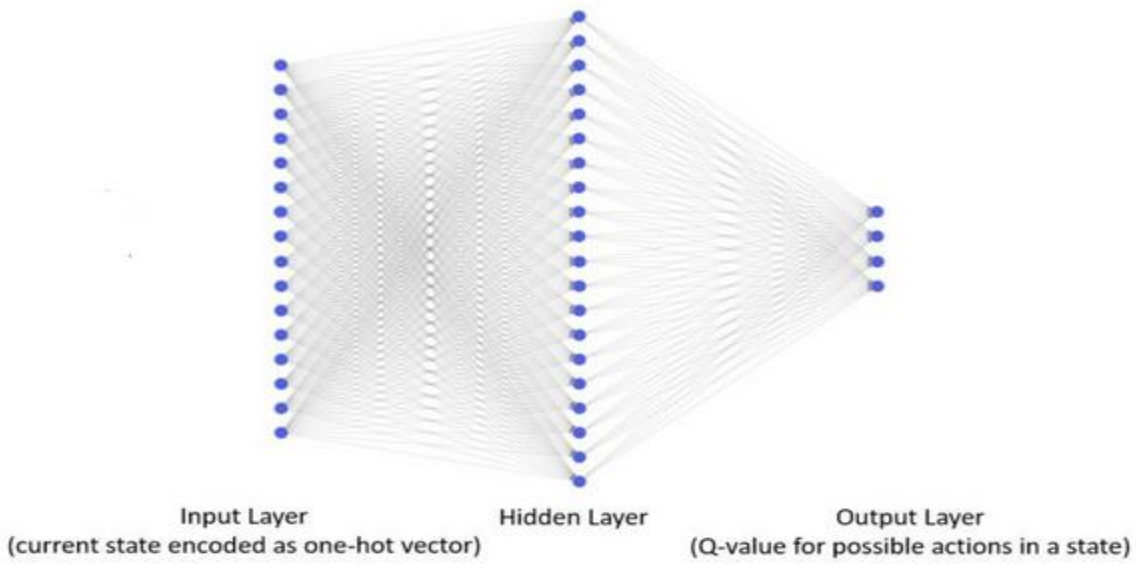
Q learning is good for small applications but not for large applications millions of states may be needed. Q learning can't be used in unknown states because it can't infer the Q value of new states from the previous ones. Therefore we approximate the Q values using some machine learning model. We replace the QTable with a Neural networks. The network receives the state as an input and outputs the Q values for all possible actions. The largest value output of the neural network is our next action.



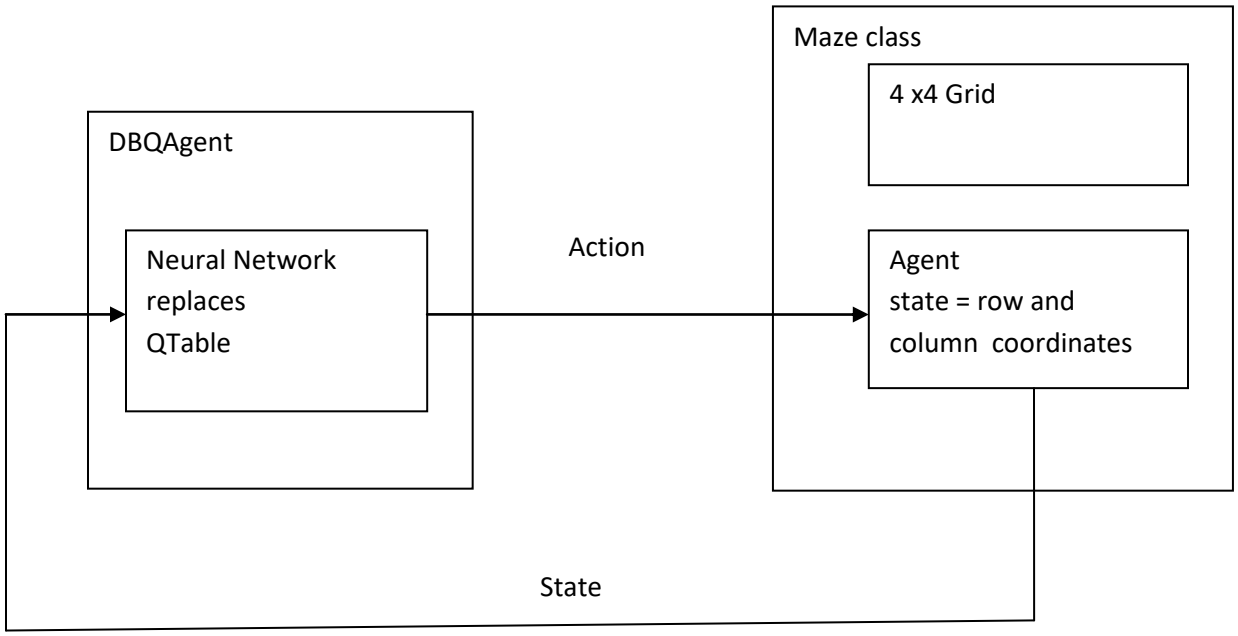
Our Neural Network will have 16 inputs each one representing one of the 16 states, a hidden layer of 20 nodes and 4 output nodes representing the QTable values for each UP, DOWN, LEFT and RIGHT action.



Our Neural Network layers may look like this:



We will have two classes a Maze class to represent our grid environment that stores the agent and state. And a DBQAgent class the stores the neural network that predicts the best action for the current state.
DBQ = Deep Based Q



We are using the **Keras** neural network that uses the **Tensor Flow** backend. We use the sequential model so we can stack many layers together.

```
self.model = Sequential()
```

We first add the input layer where the number of inputs is specified from maze observation space of 16 states.

```
self.model.add(InputLayer(batch_input_shape=(1,self.maze.observation_space())))
```

Next we add the hidden layer of 20 nodes using **relu** activation. An activation is the level a neuron fires that becomes the neuron output. An **relu** activation just allows positive values, and negative values are set to 0;

```
self.model.add(Dense(20, activation='relu'))
```

Finally we add the output layer using we use linear activation. Linear activation has no activation level at all and just outputs the actual value from the neural network.

```
self.model.add(Dense(len(self.actions), activation='linear'))
```

The model compile function sets up the model with losses, optimizer and metrics

```
self.model.compile(loss='mse', optimizer='adam', metrics=['mae'])
```

For the model we are using:

Loss = mse loss
Optimizer= adam optimizer
Metrics = mae metrics

Loss is the model objective function, for loss we are using **mse.**
Mse computes the mean squared error between actual and predictions.

Metrics is what is to be evaluated by the model during training and testing
We are using **mae** which calculates the **Mean Absolute Error**:

The **optimizer** is used to the model in response to the output of the loss function. During training the weights of the model are adjusted to try to minimize that loss function, and make our predictions as correct and optimized as possible. The loss function guides the optimizer when it's moving in the right or wrong direction to adjust the weights of the model.

We are using the **Adam** optimizer. The Adam optimizer is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

The Adam optimizer *is* computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters.

DBQgent paramaters

The DBQagent receives the following paamaters:

```
discount_factor = 0.95
eps = 0.5
eps_decay_factor = 0.999
num_episodes=500
maze=Maze
```

The discount factor allows a future reward to have less importance for actions in the present. The discount factor is set between 0 and 1.

The eps allows random actions at the beginning and as the system evolves the eps value is decreased by the eps_decay factor to allow maximum Q values to be used. Eps has a starting value of .5 and eps_decay_factor is set at .999.

Learning rate is used to control the pace at which our algorithm can learn. We set it between 0 and 1 where a value of 0, meaning no learning at all.

Our DBQagent is similar to the QTAgent the only difference is the neural network has replaced the QTable

Selecting states

Each state is represents by a identity matrix this is known as 1 hot decoder.

State	One hot encoding	UP	DOWN	LEFT	RIGHT
0	1000000000000000				
1	0100000000000000				
2	0010000000000000				
3	0001000000000000				
4	0000100000000000				
5	0000010000000000				
6	0000001000000000				
7	0000000100000000				
8	0000000010000000				
9	0000000001000000				
10	0000000000100000				
11	0000000000010000				
12	0000000000001000				
13	0000000000000100				
14	0000000000000010				
15	0000000000000001				

The state bits are selected with slicing:

```
bits = np.identity(self.maze.observation_space())[state:state + 1]))
```

The self.maze.observation_space() is 16 and the state is the state number. If the state is 6 then [state:state + 1] then the bits selected would be

0000010000000000

Each state would produce 4 Q values from the neural network model.

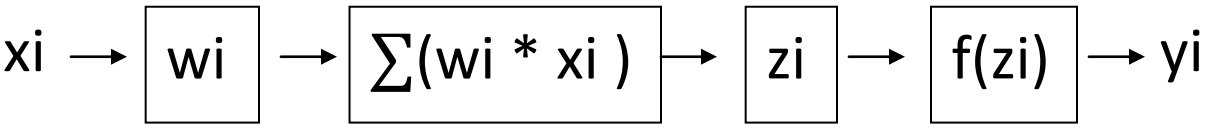
.544432	.65443	.54432	.77665

We can select the index of the Qvalue having the max value using np.argmax

```
action = np.argmax(
        self.model.predict(np.identity(self.maze.observation_space())[state:state + 1]))
```

Updating the Qvalues

Neural networks work on mathematical calculations where internal values are stored as weight in a matrix. The inputs of the neural network are multiplied and summed with the internal weights to calculate an output. The output are sent to a activation function that fires the neuron. The internal weights are used to train the neural network to give accurate outputs accordingly to the inputs.

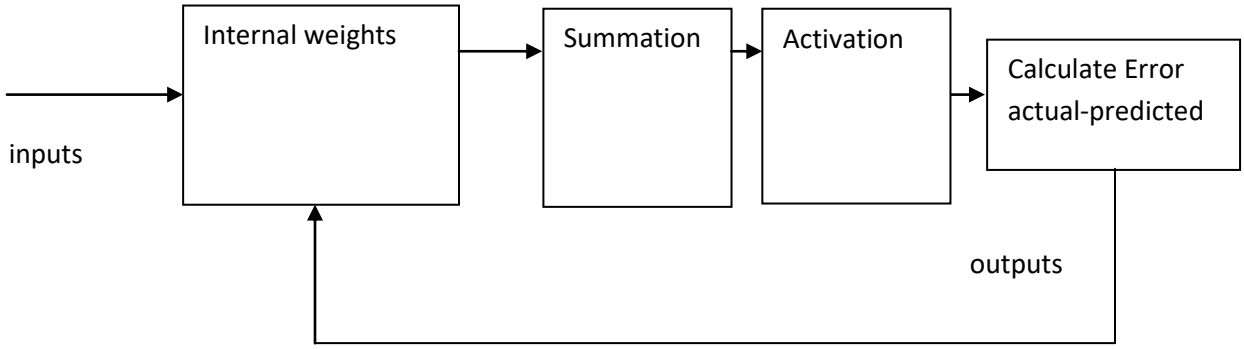


inputs weights summation sum activation output

The output of the neural network is compared to an actual value. An error is calculated from the actual and predicted values.

error = actual – predicted

The error value are used to adjust the internal weights of the neural network as to the minimize output error.



In our situation our predicted output is one of the actions and the actual value is a valid path to reach the goal and if found.

When the goal is found we increase the Q action value for that state.

Neural networks work by iteratively updating the weights and biases of the model to reduce the error in predictions that it is making. It is necessary for us to be able to calculate the model error at any point in time. The loss function enables us to do tune the model. We are using the mean-squared-error loss function.

We use the bellman equation from part1 reinforcement learning to calculate the next Q value

$$\text{Next value} = \text{reward} + \text{discount\_factor} * \max_{a'} Q'(s', a'))$$

The bellman equation adds the reward to the learning curve times the max QTValue.

The mean-squared-error loss function measures the square value of the difference between the prediction and the target:

$$\text{Loss} = [\text{predicted value} - \text{actual value}]^2$$

Using the bellman equation and state and action values

$$\text{loss} = [(\text{reward} + \text{discount\_factor} * \max_{a'} Q'(s', a')) - Q(s,a)]^2$$

The output of the loss function is to feed back the backward loss value through the network and update the weights**.** This is called back propagation. Usually stochastic gradient descent is used for back propagation. We are using **Adam** (derived from Adaptive Moment Estimation) a back propagation which **is** more efficient with lesser memory requirements

Training

W e first loop for number of episodes

```
for i in range(self.num_episodes):
```

we initialize the environment that return the state 0

```
state = self.maze.reset()
```

We then decrease the eps. The eps is to select random actions at the start of training.

```
self.eps *= self.eps_decay_factor
```

We loop until a goal is found or a wall is hit

```
done = False

    while not done:
```

If a random number is less than the **eps** then we chose a random action or it chooses the max action from the neural network Qvalue..

```
if np.random.random() < self.eps:

  action = np.random.randint(0, len(self.actions))

else:

  action = np.argmax(

    self.model.predict(np.identity(self.maze.observation_space())[state:state + 1]))

print(state, self.action_list[action])
```

We get the next state from for Agent calculated movement from the present state and action from the QTable for the present state.

```
reward, done,new_state = self.maze.step(action)
```

we get the predicted from the neural network for the present state

```
predict = self.model.predict(

  np.identity(self.maze.observation_space())[new_state:new_state + 1])
```

We now calculate the new Q target value using bellman equation and the max Q value of the predicted Qvalue for the present state.

```
target = reward + self.discount_factor * np.max(predict)
```

Again we obtain the Q values for this state from the model

```
target_vector = self.model.predict(
    np.identity(self.maze.observation_space())[state:state + 1])[0]
```

We update the model target Qvalues with the calculated target

```
target_vector[action] = target
```

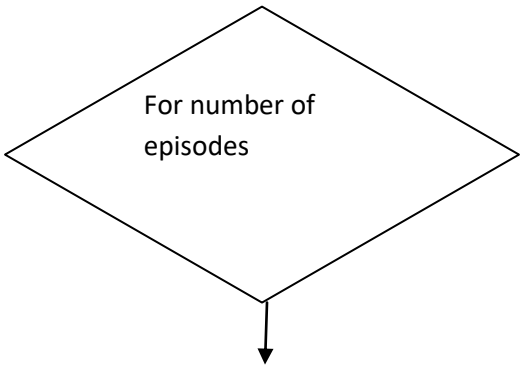
We now train the neural network with the fit method.

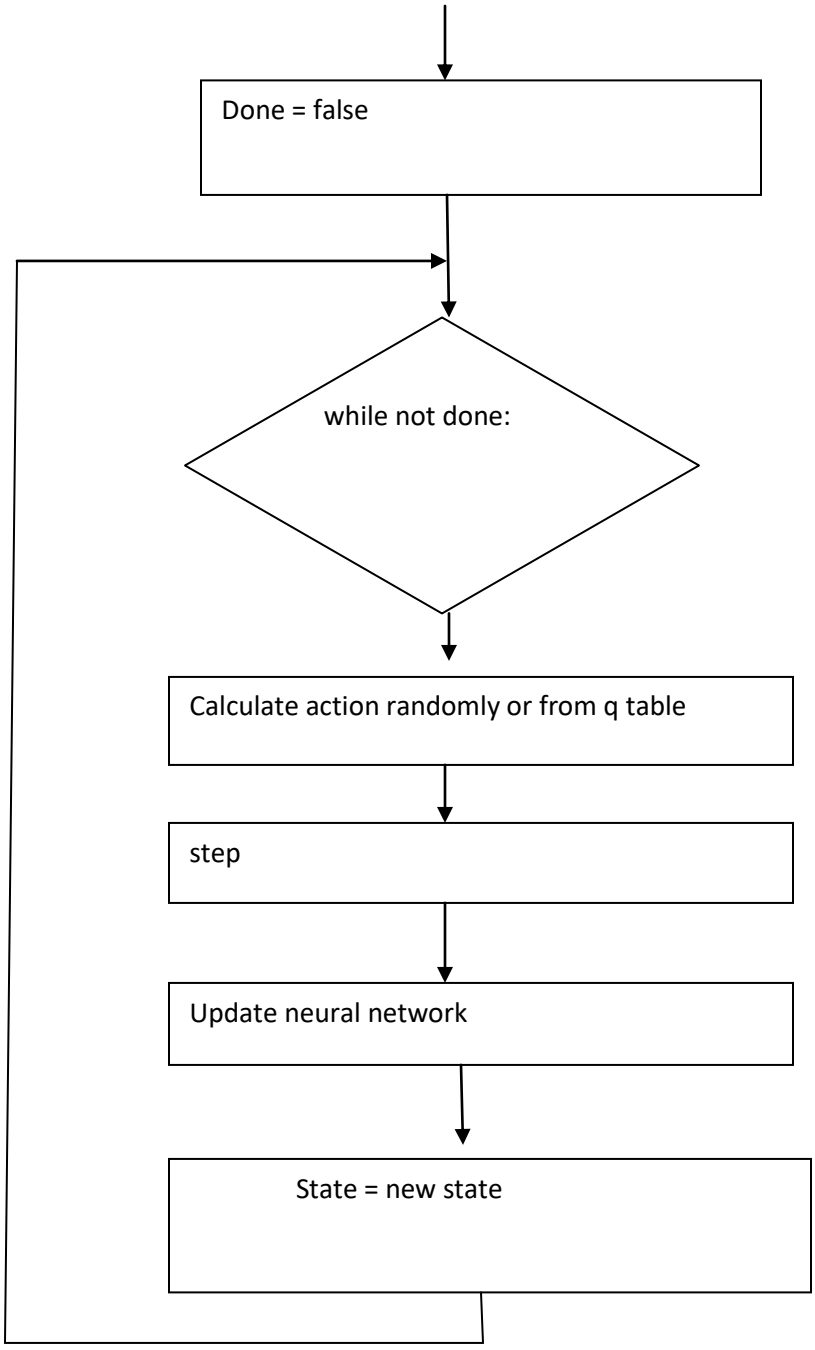
```
self.model.fit(
  np.identity(self.maze.observation_space())[state:state + 1],
  target_vector.reshape(-1,len(self.actions)), epochs=1, verbose=0)
```

The new state now is the present state

```
state = new_state
```

Here is the training flow chart





Testing

We first reset the environment to state 0

```
done = False
state = self.maze.reset()
```

we then get the prediction for current state

```
action = np.argmax(
  self.model.predict(np.identity(self.maze.observation_space())[state:state + 1]))
```

We get next state from the agent calling the step function

```
new_state = reward, done= self.maze.step(action)
```

when the new state is found we are done

Here is the complete program:

```
"""
DBQ_maze.py

deep reinforcement learning Q neural network maze
"""

import numpy as np
from keras.models import Sequential
from keras.layers import InputLayer
from keras.layers import Dense
import matplotlib.pyplot as plt
import seaborn as sns
import time


# grid contents
ROW = 0
COL = 1
OPEN = 0
WALL = 1
OCCUPIED = 2
START = 3
GOAL = 4
```

```
# actions
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3

# maze contains grid and agent
class Maze:

    # initialize maze
    def __init__(self,grid,start_row,start_col,goal_row, goal_col):

        self.grid = grid
        self.rows = len(grid)
        self.cols = len(grid[0])
        self.start_row=start_row
        self.start_col=start_col
        self.goal_row=goal_row
        self.goal_col=goal_col
        self.agent_row = 0
        self.agent_col = 0

    # reset environment
    def reset(self):

        self.agent_row = 0
        self.agent_col = 0
        state_number= self.agent_row*self.rows + self.agent_col
        return state_number

    # return number of states
    def observation_space(self):
        return self.rows * self.cols;

    # get next state
    def step(self,action):

        # initialize
        done = False
        r = 0
        prev_agent_row,prev_agent_col = self.agent_row, self.agent_col
```

```
    # get next move
    if (action == UP):
        self.agent_row = max(self.agent_row-1,0)
    if (action == DOWN):
        self.agent_row = min(self.agent_row +1,self.rows-1)
    if (action == LEFT):
        self.agent_col = max(self.agent_col -1,0)
    if (action == RIGHT):
        self.agent_col = min(self.agent_col +1,self.cols-1)

    # wall hit, or goal found
    if (self.isDone( self.agent_row, self.agent_col)):
        done = True
        # goal found
        if ((self.agent_row == self.goal_row) and (self.agent_col == self.goal_col)):
            print('Target reached')
            r = 1

        # wall hit
        else:
            print('hit a wall')
            self.agent_row,self.agent_col = prev_agent_row, prev_agent_col


    # store next state
    next_state= self.agent_row*self.rows + self.agent_col

    # return agent state, reward, done, and number state
    return(r,done,next_state)


# return true if start, wall or goal found
def isDone(self,row,col):
    done = False
    if(self.grid[row][col] == START or self.grid[row][col] == WALL
                or self.grid[row][col] == GOAL ):
        done = True
    return done
```

```
    # render maze as a heatmap
    def render(self):

        cells = np.copy(self.grid)
        r,c = self.agent_row,self.agent_col
        cells[r][c] = OCCUPIED
        sns.heatmap(cells)
        plt.show()


# QAgent containing neural network
class DBQAgent:

    # initialize environment
    def __init__(self,discount_factor = 0.95,eps = 0.5,eps_decay_factor =
0.999,num_episodes=500,maze=None):

        # store values
        self.actions = [UP,DOWN,LEFT,RIGHT]
        self.maze = maze
        self.discount_factor =discount_factor
        self.eps = eps
        self.eps_decay_factor = eps_decay_factor
        self.num_episodes = num_episodes
        self.action_list=['UP','DOWN','LEFT','RIGHT']

        # initialize keras neural network
        self.model = Sequential()
        self.model.add(InputLayer(batch_input_shape=(1,self.maze.observation_space())))
        self.model.add(Dense(20, activation='relu'))
        self.model.add(Dense(len(self.actions), activation='linear'))
        self.model.compile(loss='mse', optimizer='adam', metrics=['mae'])

    # train neural network
    # for each action a new state is produced
    def train(self):

        action = 0
        for i in range(self.num_episodes):

            # initialize
            state = self.maze.reset()
            self.eps *= self.eps_decay_factor
            done = False
```

```
        while not done:

            # use random
            if np.random.random() < self.eps:
                action = np.random.randint(0, len(self.actions))

            # use max Qvalue from neural network for this state
            else:
                action = np.argmax(
                    self.model.predict(np.identity(self.maze.observation_space())[state:state + 1]))
            print(state, self.action_list[action])

            # get next state from Agent movement
            reward, done,new_state = self.maze.step(action)

            # make prediction
            predict = self.model.predict(
                np.identity(self.maze.observation_space())[new_state:new_state + 1])

            # calculate actual target for reward
            target = reward + self.discount_factor * np.max(predict)

            # make prediction for this state
            target_vector = self.model.predict(
                    np.identity(self.maze.observation_space())[state:state + 1])[0]

            # update target with actual target
            target_vector[action] = target

            # train with updated target
            self.model.fit(
                np.identity(self.maze.observation_space())[state:state + 1],
                target_vector.reshape(-1,len(self.actions)), epochs=1, verbose=0)

            # state is new state
            state = new_state

    # test predictions
    def test(self):

        done = False
        state = self.maze.reset()
        self.maze.render()
        print(state)
```

```
        while not done:

            # get next prediction
            action = np.argmax(
                self.model.predict(np.identity(self.maze.observation_space())[state:state + 1]))

            # get next state from prediction
            reward, done, new_state= self.maze.step(action)
            state = new_state
            print(state,q.action_list[action])
            self.maze.render()
            time.sleep(1)

# start

# make grid
# 0 - path, 1 = wall, 2 = occupied, 3 = start and 4 = goal
grid = [[3,0,0,0],[0,0,1,0],[0,0,1,0],[1,0,0,4]]
print(grid)

# make maze
maze = Maze(grid,0,0,3,3)

# make agent
discount_factor = 0.95
eps = 0.5
eps_decay_factor = 0.999
num_episodes=500
q = DBQAgent(discount_factor,eps,eps_decay_factor,num_episodes,maze)

# train environment
print("training;")
q.train()
print("Q Values")
for state in range(maze.observation_space()):
    print(state,q.model.predict(np.identity(maze.observation_space())[state:state + 1])[0])

# test environment
print("testing")
q.test()
print("done")
```

Todo

Type in or copy and paste in the above program and run it,

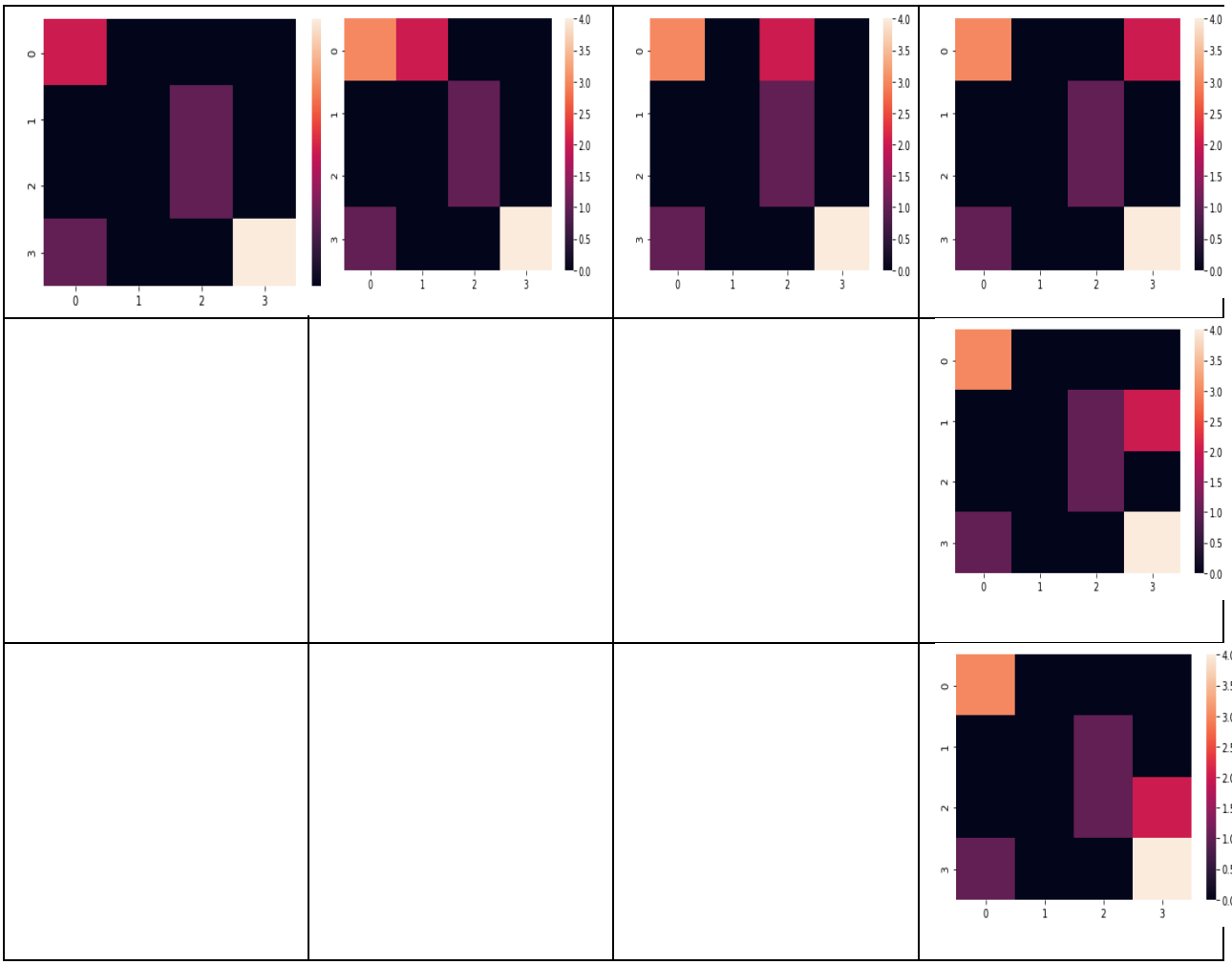
You should get something like this:

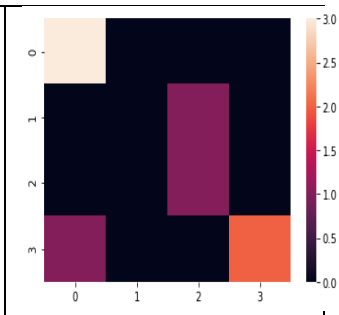
OUTPUT:

```
[[3, 0, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [1, 0, 0, 4]]
training;
0 DOWN
4 RIGHT
5 RIGHT
hit a wall
0 DOWN
4 RIGHT
5 LEFT
4 RIGHT
5 UP
1 DOWN
5 UP
1 DOWN
5 UP
1 DOWN
5 LEFT
4 RIGHT
5 DOWN
9 UP
5 LEFT
4 DOWN
8 DOWN
hit a wall

----------


0 RIGHT
1 RIGHT
2 RIGHT
3 DOWN
7 DOWN
11 DOWN
Target reached

Q Values
0 [2.3720932 1.8992106 2.2173681 2.4427826]
1 [2.4345908 2.0929148 1.9616205 2.581113 ]
```

2 [2.3791873 2.4087768 2.426102 2.7110744]
3 [2.7222464 2.7957635 2.6974301 2.5380318]
4 [1.7537462 1.479912 1.6729028 1.6943152]
5 [2.3550935 1.9232206 2.1312804 1.976614]
6 [1.6290405 1.8833666 1.9506173 1.9110621]
7 [2.5776744 2.8820634 2.5313945 2.7156956]
8 [1.6553805 1.5330678 1.514137 1.510173]
9 [2.063649 1.5916361 1.7150327 1.8191462]
10 [2.5243807 2.0043151 2.2995152 2.1293592]
11 [2.6825967 3.0155954 2.9744763 2.3577335]
12 [1.6130059 1.7181463 1.8973836 1.5587605]
13 [1.8714818 1.5990793 1.6129892 1.9221941]
14 [1.5759425 1.491981 1.6160327 1.899595]
15 [1.6895798 2.1073017 2.1452239 1.6371424]

testing





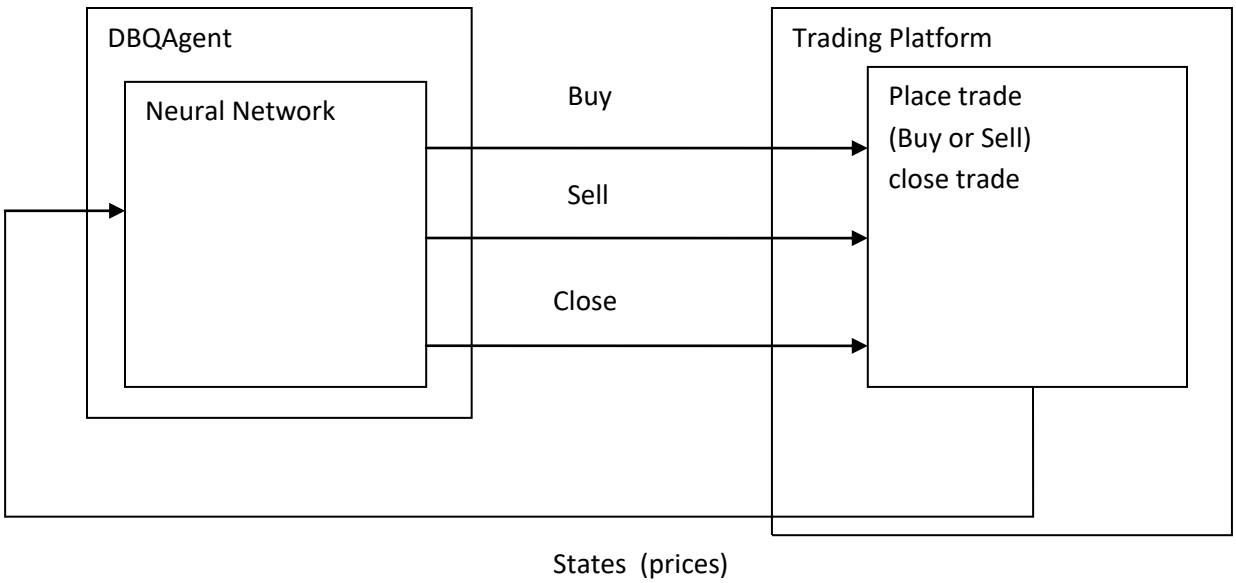
DEEP LEARNING REINFORCEMENT HOMEWORK 3

Apply Deep Reinforcement learning for the stock prediction program of previous lessons. You may need 2 neural networks, above for buying and cashing out and another one for selling and cashing out.

The actions would be holding buy/sell and cash out.

The states would be the market dynamics. The prices are increase deceasing or flat. Buying is when the prices are changing direction. The environment is buying and selling and cashing in stocks.The system is rewarded when it is making money.

Call your py file deep_reinforcement_homework3.py



You should get something like this

END