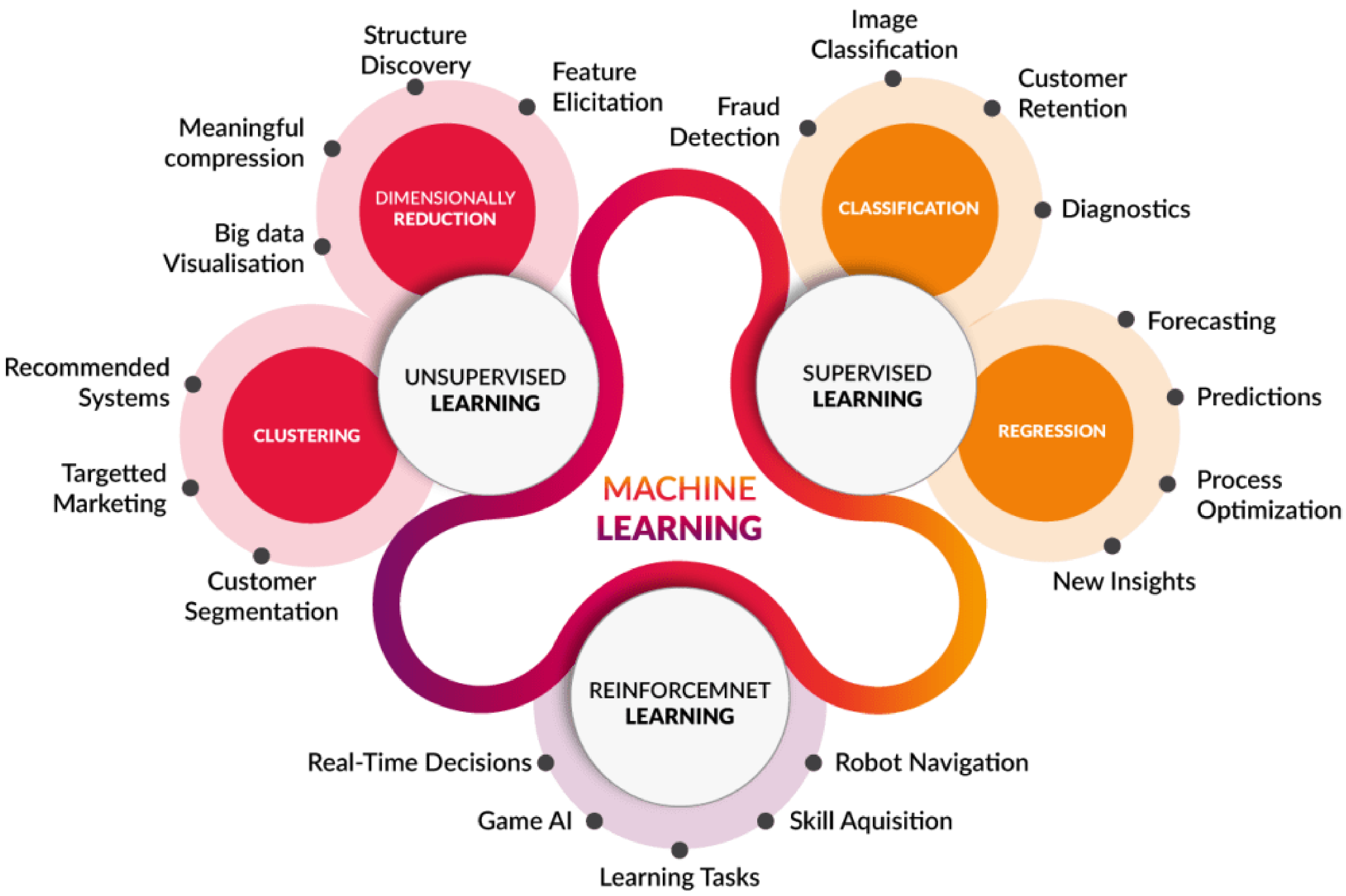
GROUP 20 – REINFORCEMENT LEARNING

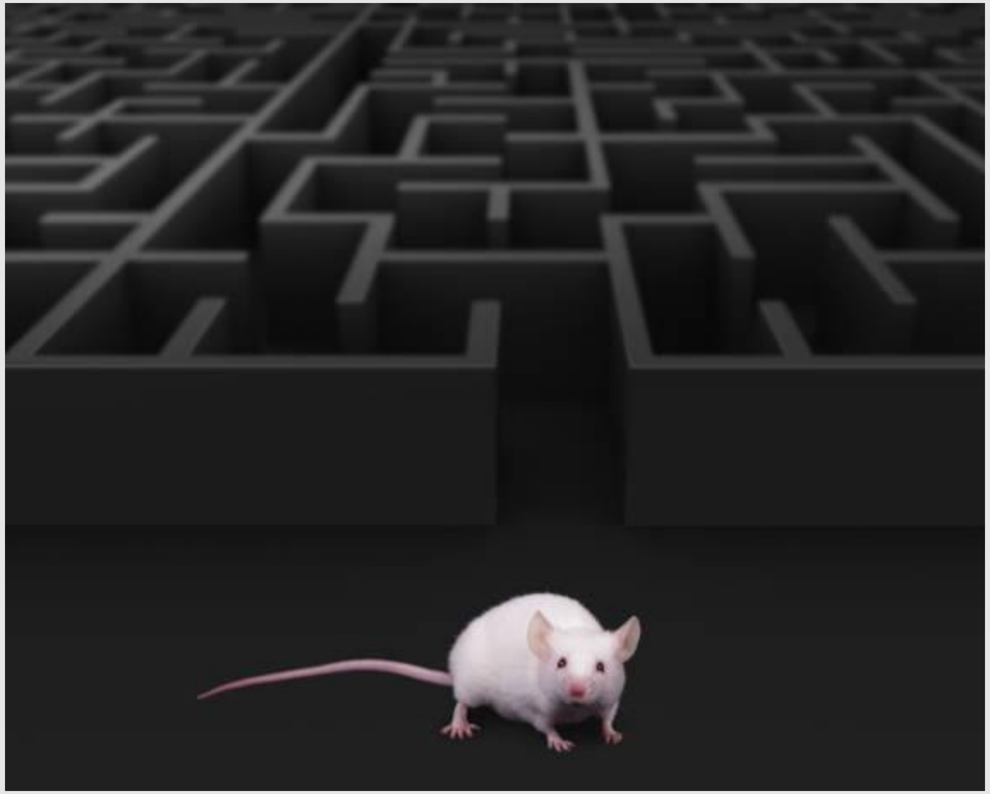


Reinforcement Learning

- Supervised Learning; requires labelled data, can classify or regress
- Unsupervised Learning; unlabeled data, clustering and dimensionality reduction
- 3rd branch of machine learning; Reinforcement Learning

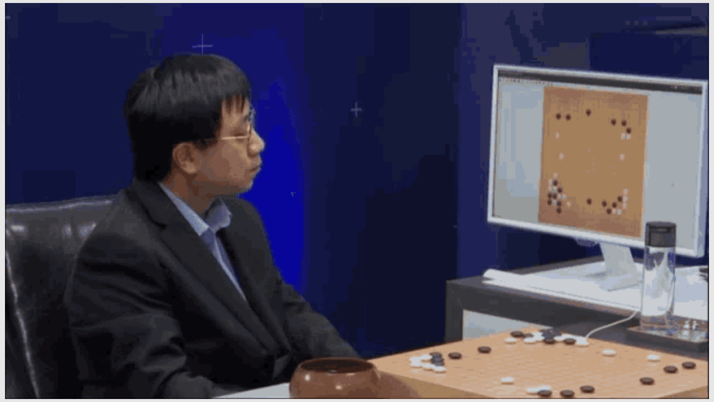
Reinforcement Learning – Introduction

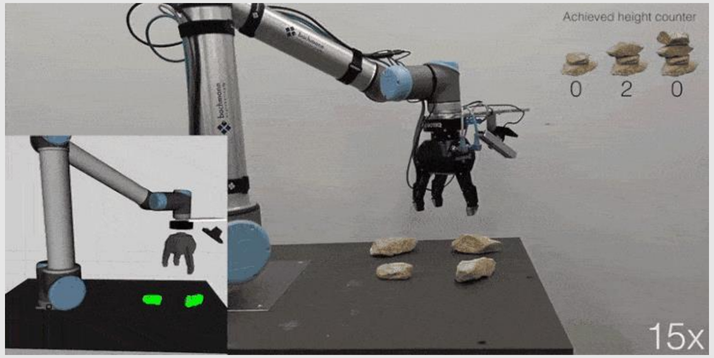
- Reinforcement Learning (RL):
  - Agent conducts action, is rewarded based on action
  - Learning via Trial and Error
  - Markov Decision Process (MDP) the bedrock of RL
- Important characteristics of RL
  - No supervisor, only reward signal
  - Process involves sequential decision making
  - Agents actions determine subsequent data
  - The agent must complete an objective but the actions to complete this are unknown
- Goal of RL
  - Take actions that maximize cumulative reward

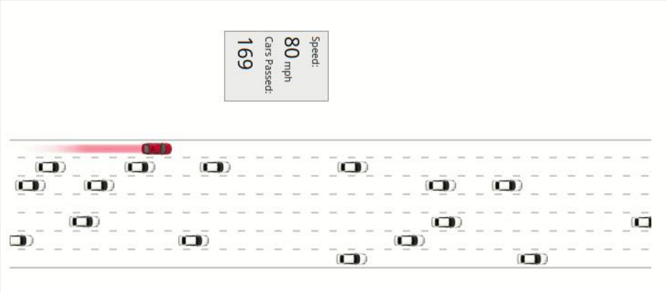


Reinforcement Learning – Applications and Success Stories

- AlphaGo; Google DeepMind AI defeated South Korean professional player Lee Sedol 4 matches to 1 in traditional "Go" game
- Robotics; find best combination of signals to complete a defined set of motions
- Autonomous driving; voyage deep drive, AWS deep racer, deep traffic

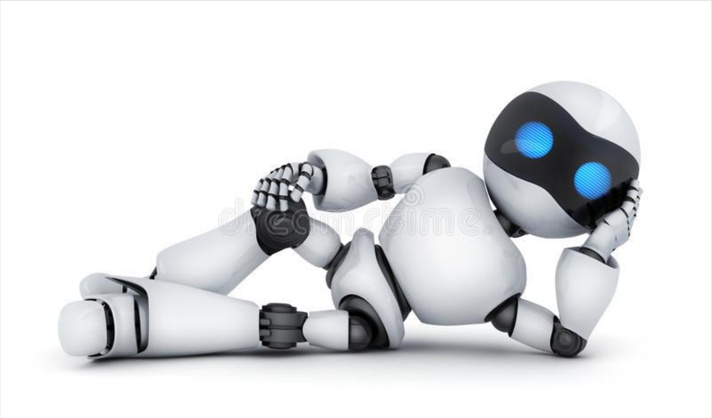






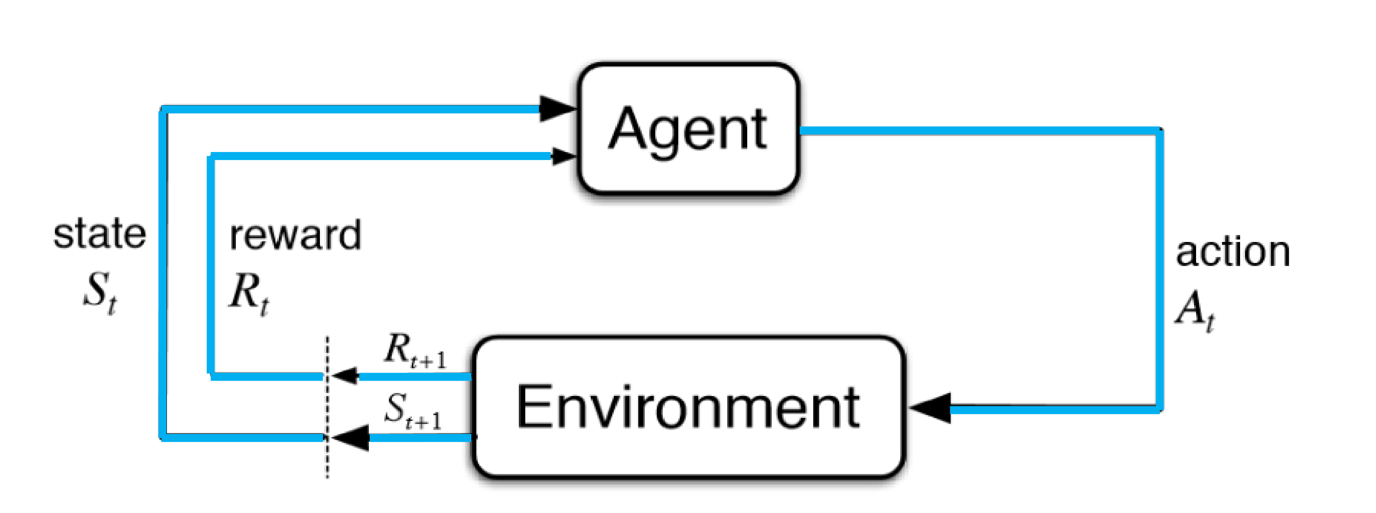
Reinforcement Learning – Important Terms

- **Agent:** an entity which performs actions in an environment to gain some reward
- **Environment:** a scenario that an agent must face
- **Reward:** a return given to an agent when a specific task is performed
- **State:** the current condition returned by the environment
- **Policy:** strategy applied by agent to decide the next action based on current state
- **Value:** expected long term reward (i.e. collection of rewards)as compared to the short-term reward



Markov Decision Process

- MDPs have a finite set of states (S), a set of actions (A), and a set of Rewards (R)



- The process of receiving a reward is an arbitrary function that maps state action pairs to rewards

$$f(S_t, A_t) = R_{t+1}$$

- The set of sequential processes that represent the MDP can be represented as $S_t, A_t, R_{t+1}, \ldots S_n, A_n, R_{n+1}$
- Since our goal is to maximize the reward, we need to find a sequence of state, action pairs that achieve maximum <u>cumulative R</u>

Solving MDPs with Q Learning – Cumulative Reward

- The goal of an agent is to maximize cumulative rewards; need to aggregate discrete R terms
- Can do this with the concept of expected return:

$$G_t = R_{t+1} + R_{t+2} + \cdots R_T$$

- Where T is the final time step, in an episodic task
- For continuous tasks $\mathrm{T} = \infty$, need to consider discounting
- Discount rate γ is a number between 0 and 1, used to determine the present value of future rewards

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$
$$= \sum \gamma^k R_{t+k+1}$$

- The discounted return (G_T) makes it so that immediate rewards are worth more than future rewards, because future rewards are more heavily discounted

Solving MDPs with Q Learning – Policies and Value Functions

Policies

- What is the probability that an agent will select a specific action from a specific state?

Value Functions

- How beneficial is being in a specific state or taking a specific action for the agent?

Solving MDPs with Q Learning – Policies and Value Functions

Policies

- If an agent follows a given policy (π) at time t then $\pi(a|s)$ is the probability that the agent will take action a in state s (or $A_t = a \,\&\, S_t = s$)
- At time t, following policy π, the probability of an agent taking an action a in state s is $\pi(a|s)$

Value Functions

- Q learning uses action-value function (q_π); gives the value of an action under policy π
- The value of action a, in state s, under policy π is the expected return from starting from state s at time t, taking action a, and following policy π
- Mathematically

$$\underbrace{\boxed{q_\pi(s,a)}}_{\text{Q-Function}} = \underbrace{\boxed{E[G_t|S_t = s, A_t = a]}}_{\text{Q-Value}}$$

Solving MDPs with Q Learning – Optimal Action Value Function

- Optimal Q function (q_*) gives the largest expected return achievable when following π for all possible state action pairs and can be defined as

$$q_*(s, a) = \max_{\pi} q_\pi(s, a)$$

- For Q Learning, q_* must satisfy the Bellman Optimality equation

$$q_*(s, a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s', a')\right]$$

- The Q learning algorithm iteratively updates the Q values for a state-action pair using the Bellman equation until the Q function converges to q_*; called value iteration
- Q values are stored in Q table, agent will take action that leads to highest Q value in a given state

	Action 1	Action 2	Action 3	Action 4
State 1	Q(1, 1)	Q(1, 2)	Q(1, 3)	Q(1, 4)
State 2	Q(2, 1)	Q(2, 2)	Q(2, 3)	Q(2, 4)
State 3	Q(3, 1)	Q(3, 2)	Q(3, 3)	Q(3, 4)

Solving MDPs with Q Learning – Exploitation and Exploration

	Action 1	Action 2	Action 3	Action 4
State 1	0	0	0	0
State 2	0	0	0	0
State 3	0	0	0	0

?

- The agent must weigh exploitation/exploration tradeoff
  - Exploit what it already knows vs. explore rewards from untested actions
  - Can use the following to address exploitation exploration tradeoff
    - Random exploration
    - Greedy exploitation
    - ϵ - Greedy strategy

Solving MDPs with Q Learning – Developing a Full Q Table

- What happens if agent is unable to find optimal Q values for each state action pair on first try?
- Do not want agent to forget everything from previous iteration if new iteration is started
- Objective is to develop a Q table with optimal Q values for each state action pair based on many iterations of agent in state action pairs (called episodes)
- Can be achieved with the learning rate ($\alpha, \in [0, 1]$)

$$\underbrace{q_{new}(s,a)}_{\text{New table value}} = \underbrace{(1-\alpha)q(s,a)}_{\text{Old table value}} + \alpha\underbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}_{\text{New calculated value } (q_*)}$$

Episode 1

	Action 1
State 1	0

Episode 2

	Action 1
State 1	-0.5

Episode 3

	Action 1
State 1	-0.25

The Q Learning Algorithm

Initialize Q table

Select Initial State

Perform an Action

Evaluate Q value and Update Q Table

Return to 2 if goal is reached, else return to 3

Q Learning Example – The Frozen Lake

- Open AI gym – Frozen Lake
- Frozen Lake game; Hiker crossing frozen lake
- Actions that lead to holes in the ice receive no reward
- Actions that lead to the party receive positive reward
- Over many iterations the actions that receive negative rewards will be filtered out while actions that receive positive rewards will remain

Q Learning Example – The Frozen Lake

- Agent can move left, right, up, or down (4 actions)
- Agent can walk on 11 frozen lake tiles, 4 hole tiles, or 1 party tile (16 states)
- Walking to the party tile yields a reward of 1 while all other tiles yield reward of 0

Q Learning Example – Define Helper Functions

```
In [3]: import gym
        import numpy as np
        import matplotlib.pyplot as plt
        import time, pickle, os
        from statistics import mean
        %matplotlib inline
```

Library imports

```
def choose_action(state):
    action=0
    if np.random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q[state, :])
    return action

def learn(state, state2, reward, action):
    predict = Q[state, action]
    target = reward + gamma * np.max(Q[state2, :])
    Q[state, action] = Q[state, action] + lr_rate * (target - predict)
```

Function which chooses an action based on a state

Function which Updates Q table

Q Learning Example – Initialize Enviornment

```
In [5]: env = gym.make('FrozenLake-v0')

        epsilon = 0.9
        total_episodes = 2000
        max_steps = 100

        lr_rate = 0.81
        gamma = 0.96

        Q = np.zeros((env.observation_space.n, env.action_space.n))
        scores = []
```

Initializes game environment

Defines q learning paramaters

Initializes Q table

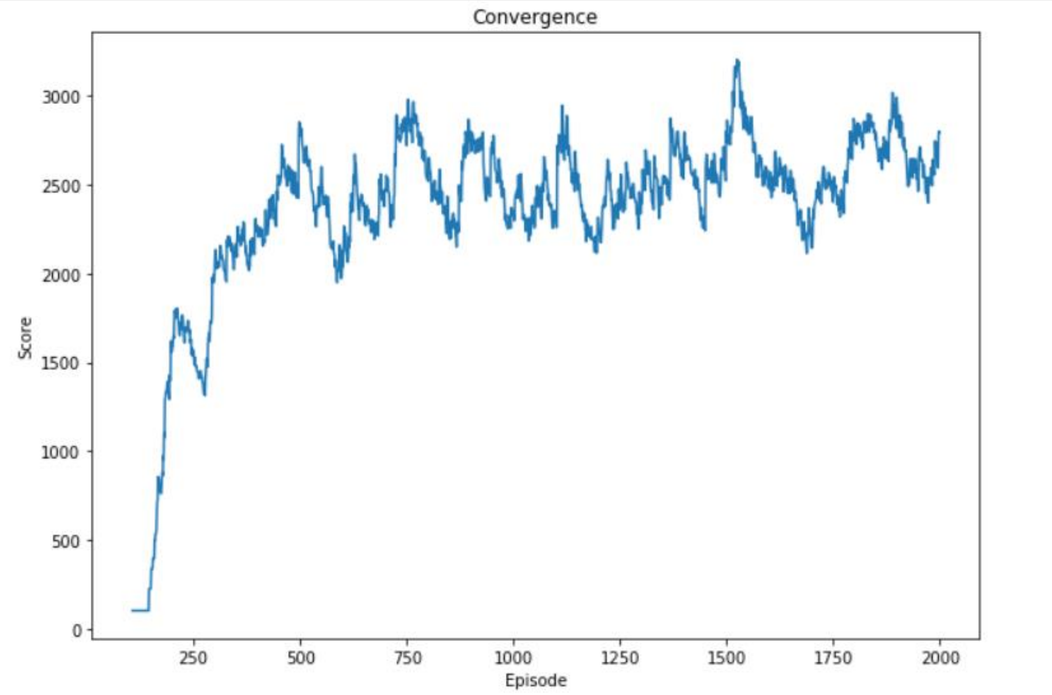
Q Learning Example – Running the Episodes

```
In [6]: for episode in range(total_episodes):
            state = env.reset()
            t = 0

            while t < max_steps:
                env.render()

                action = choose_action(state)

                state2, reward, done, info = env.step(action)

                learn(state, state2, reward, action)

                state = state2

                t += 1

                print(episode)

                if done:
                    break

                time.sleep(0.1)

            score = np.round(np.sum(Q/np.max(Q)*100))
            scores.append(score)

        print(Q)
```

Iterating through episodes, our agent takes actions, receives rewards, calculates q values and updates the q table based on the actions it has taken

Q Learning Example – Convergence

Q Score

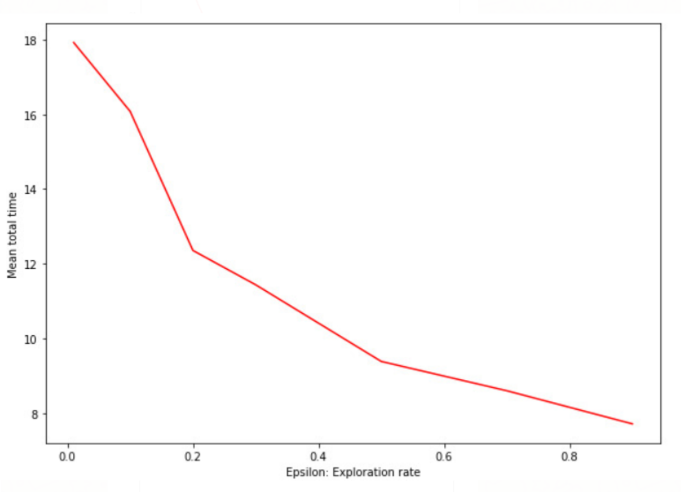


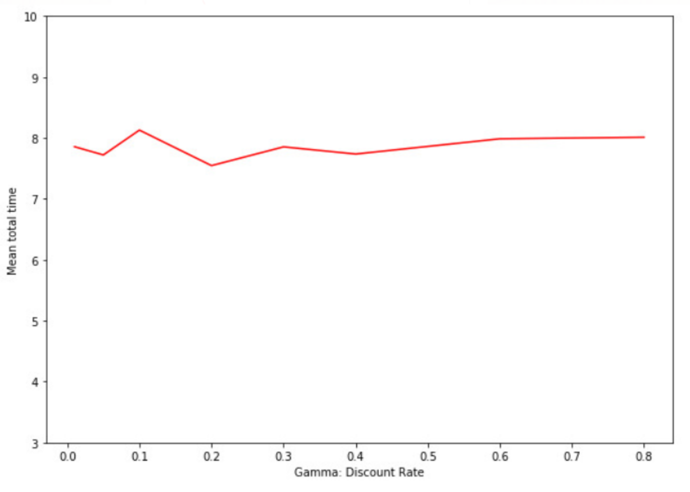
Final Q matrix

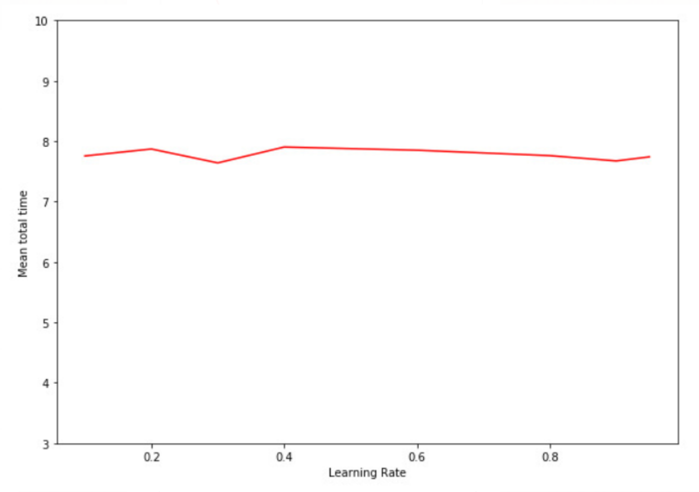
```
[[6.53118287e-01 6.76787165e-01 6.51955832e-01 6.62853501e-01]
 [6.56569162e-01 5.35738000e-01 5.72516132e-01 6.60733902e-01]
 [6.85009109e-01 6.86108158e-01 6.66498436e-01 6.80596059e-01]
 [5.57194560e-01 6.56040256e-01 6.47382293e-01 6.82087373e-01]
 [6.69494180e-01 6.34166430e-01 1.40381227e-01 6.53596992e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [8.00347795e-01 3.09376784e-05 7.51117493e-01 1.30250050e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [2.13311240e-02 7.83891691e-01 5.98714178e-01 7.12689471e-01]
 [7.14559204e-01 8.31028819e-01 8.86971198e-01 1.45849926e-01]
 [8.18272147e-01 9.26745089e-01 7.83663317e-01 1.52822043e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [4.87152449e-03 9.13074747e-01 9.34373235e-01 8.23645602e-01]
 [9.18358844e-01 9.07034013e-01 8.56813866e-01 9.95949472e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

Q Learning Example – Hyperparameter Tuning

- Tuned hyper parameter to get agent to complete maze in fewest steps
- Hyper Parameter Tuning yielded an optimal $\epsilon = 0.9$, $\alpha = 0.3$, and $\gamma = 0.2$







Thanks!