


 Edward-Kuhn / ml_Assaginment Public


generated from [Thayer-ENG5108/Assignment_5_Fall2022](#)

<> Code

 Issues


 Pull requests

 Actions

 Projects

 Wiki


 Security


 Insig


 main ▾



ml_Assaginment / Assignment_5_Fall2022.Dianhao_Liu.ipynb

 Edward-Kuhn Add files via upload 🕒 History

 1 contributor

1.74 MB 

ENGS 108 Fall 2022 Assignment 5

Due October ??, 2022 at 11:59PM on Canvas

Instructors: George Cybenko

TAs: Clement Nyanhongo and Chase Yakaboski

Rules and Requirements

1. You are only allowed to use Python packages that are explicitly imported in the assignment notebook or are standard (builtin) python libraries like random, os, sys, etc, (Standard Builtin Python libraries will have a Python.org documentation). For this assignment you may use:

- [numpy](#)
- [pandas](#)
- [scikit-learn](#)
- [matplotlib](#)

2. All code must be fit into the designated code or text blocks in the assignment notebook. They are identified by a **TODO** qualifier.

3. For analytical questions that don't require code, type your answer cleanly in Markdown. For help, see the [Google Colab Markdown Guide](#).

```
In [ ]: ''' Import Statements '''
import numpy as np
import pandas as pd
import sklearn
import sys
import matplotlib.pyplot as plt
import os
from PIL import Image
import seaborn as sns
from copy import deepcopy
from random import *
from numpy.random import choice
```

Data Loading

Upload the dataset for this assignment

```
In [ ]: #TODO: Set your base datasets path. This is my base path, you will need to chan
```

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

```
In [ ]: base_dir = '/content/drive/MyDrive/ml_Assignment_5/'
```

```
image_path = base_dir + 'images/'
```

```
# Load images
images = os.listdir(image_path)
print(images)
```

```
['image_3.jpg', 'image_1.jpg', 'image_2.jpg']
```

Problem 1: Revisit: K -Means Clustering (Color Compression)

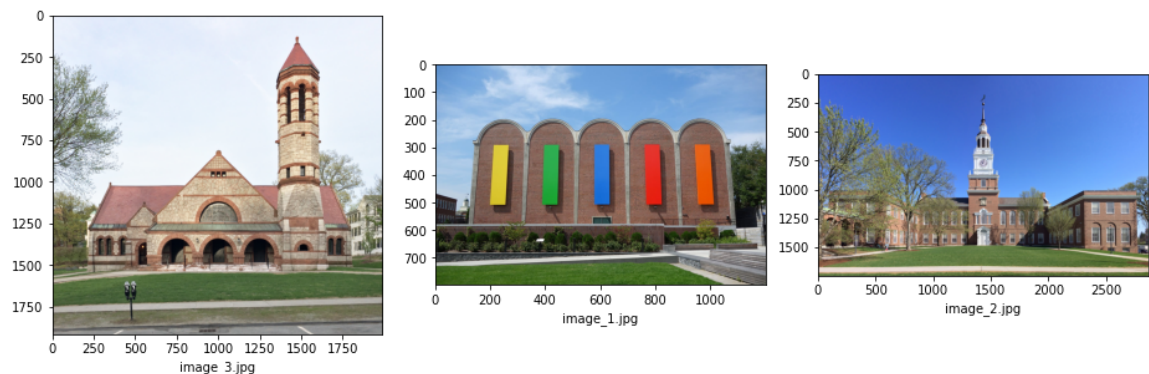
In this problem, we revisit the K-means clustering algorithm to compress the number of pixels in an image. We will also implement the algorithm from scratch.

Display set of images

```
In [ ]: #display all images in a 1*3 subplot
fig, axes = plt.subplots(1, 3, figsize = (13, 10))

for i, d in enumerate(images):
    img = plt.imread(image_path + d)
    axes[i].imshow(img)
    axes[i].set_xlabel(d)

fig.tight_layout(h_pad=1, w_pad=1)
```



```
In [ ]: # The following function resizes the image to return a size*size rgb vector rep
def image_resize (img_path, size):
    # resize image
    img = Image.open(img_path)
    img = img.resize((size, size), Image.ANTIALIAS)
    return img
```

a) Resize and display all the given images to a 64 * 64 image in RGB format

```
In [ ]: def resize_all (images):

    new_images = []    # array to store resized images

    for i in images:
        new_images.append(image_resize(image_path + i, 64))

    return new_images
```

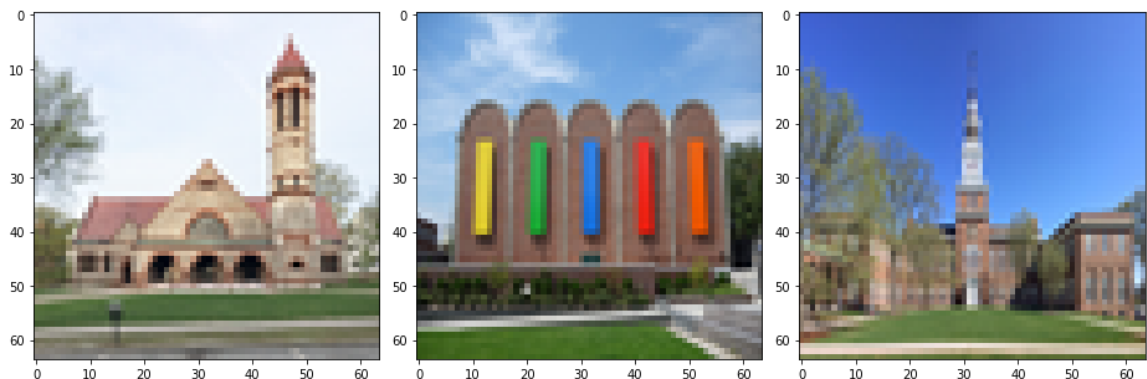
```
In [ ]: # display the resized images below
new_images = resize_all(images)

def plot_images (new_images):
    fig, axes = plt.subplots(1, 3, figsize = (13, 10))

    for i, d in enumerate(new_images):
        axes[i].imshow(d)

    fig.tight_layout(h_pad=1, w_pad=1)

plot_images(new_images)
```



b) Details of the naive Kmeans clustering algorithm are provided in https://en.wikipedia.org/wiki/K-means_clustering

Based on the naive Kmeans, we will implement our algorithm as follows:

1. Initialize a set of k pixel locations as your clusters (at time 0 this is done randomly, afterwards they are computed in step 4).

For each image pixel:

2. Find the distance of each pixel to each cluster (will use Euclidean distance)

3. Assign pixel-color to the color of its closest cluster
4. Recompute the cluster centers based on color assignments and repeat procedure

The procedure is done for n-steps.

This procedure is described in the function below:

```
In [ ]: def my_Kmeans (k, image, n_runs):
        """
        k - number of clusters
        image - image array
        n_runs - number of k-means iterations
        """

        # 1. initialize cluster centers (at t = 0)
        K_clusters = initialize_clusters(image, k)

        for iteration in range(n_runs):

            # pixel_location is a disctionary (key = index of k-cluster, value - list
            # of the image pixels belonging to the kth cluster
            pixel_locations = {i:[] for i in range(k)}

            # 2. compute distance from each image pixel to the clusters ((i, j) - list
            for i in range(len(image)):
                for j in range(len(image[0])):
                    dist_to_clust = distance_to_cluster (i, j, K_clusters, image)

                    # 3. change pixel color to that of its closest cluster
                    change_pixel_color(image, i, j, K_clusters, dist_to_clust, pixel_locations)

            # 4. reassign clusters by computing new pixel centers
            K_clusters = re_assign_clusters (pixel_locations, K_clusters)

        # return image after the kmeans algorithm compresison
        return image
```

Complete the following functions to perform the Kmeans algorithm

- c). TODO: Write a function to initialize a set of pixel clusters given an individual image

```
In [ ]: # is necessary, use the numpy random library to select k - (x, y) coordinates f
        # of these k-locations are the initial cluster centers

        def initialize_clusters(image, k):

            K_clusters = [] # cluster centers

            # TODO: Find random clusters here
```

```

for l in range(k):

    i = randint(0, len(np.array(image)[0])-1)
    j = randint(0, len(np.array(image)[1])-1)
    K_clusters.append([i, j, image[i][j]])

# Your output should be a list of k random clusters each in the form [x, y,
return K_clusters

```

```

In [ ]: # To test your initialize_cluster functions, we will plot an image as well as i

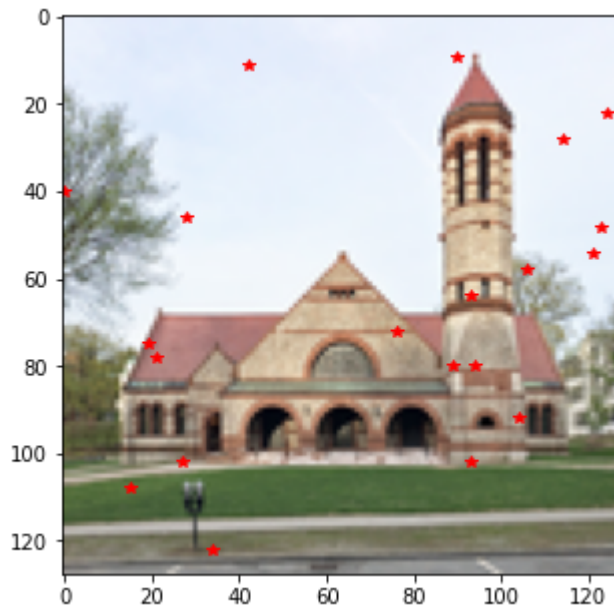
img_path = base_dir + 'images/image_3.jpg'

img = np.asarray(image_resize (img_path, 128))
clusters = initialize_clusters(img, 20)

# plot image
fig, axes = plt.subplots(figsize = (5, 20))
axes.imshow(img)

clusters = initialize_clusters(img, 20)
# plot initialized cluster centers in color = blue
for k in clusters:
    axes.plot(k[0], k[1], '*', color = 'red')

```



c): TODO: Write a function to find the distance from each image pixel to all the clusters centres

```

In [ ]: # function to compute the pixel distance from pixel (i, j) to each of the K clu
# pixel distance = Euclidean distance of an (i, j) pixel to each a pixel in k

# After recomputing new clusters, some
def distance_to_cluster (i, j, K_clusters, image):

    distances = [sys.maxsize for i in range(len(K_clusters))] # initialize dis

```

```
# TODO: Write function to modify distances with actual distances
for k in range(len(K_clusters)):
    a = image[i][j]
    b = K_clusters[k][2]
    distances[k] = np.linalg.norm(a - b)

return distances    # vector of Euclidian distances from each image's pizel
```

```
In [ ]: """[cluster[0] for cluster in K_clusters] #list comprehension"""
```

```
Out[ ]: '[cluster[0] for cluster in K_clusters] #list comprehension'
```

d) TODO: Change pixel color to that of its closest cluster (in terms of distance)

```
In [ ]: # pixel_locations is a dictionary with key - index to a cluster, value - list o
# dist_to_clusters - distance from pixel at (i, j) to the K pixel clusters

def change_pixel_color(image, i, j, K_clusters, dist_to_clusters, pixel_locatio

    """
    (i, j) : Coordinates of pixel
    pixel_locations - is a dictionary with key > index to a cluster, value > li
    dist_to_clusters - distance from pixel at (i, j) to the K pixel clusters
    """

    # TODO: change pixel color to that of its closes cluster based on dist_to_c

    min = np.argmin(dist_to_clusters)
    pixel_locations[min].append([i,j,image[i][j]])
    image[i][j] = K_clusters[min][2]

    return;
```

e). TODO: Reassign the new clusters

```
In [ ]: def re_assign_clusters (pixel_locations, K_clusters):

    new_clusters = [[] for i in range(len(K_clusters))]    # This will have our

    #TODO:
    # Iterate the pixel locations dictionary (represents pixels of the same co
    # The pixel of this x, y coordinate is the new cluster center (round of va
    # new_clusters should be a list with entries of the form [x, y, RGB_pixel],
    #     locations

    # Complete here!

    for i in range(len(pixel_locations)):

        x_total=0
        y_total=0
```

```

    for j in range(len(pixel_locations[i])):
        x_total+=pixel_locations[i][j][0]
        y_total+=pixel_locations[i][j][1]
        x_mean=x_total/len(pixel_locations[i])
        y_mean=y_total/len(pixel_locations[i])

    new_clusters[i]=[round(x_mean),round(y_mean),K_clusters[i][2]]

# Note: Ensure that new_clusters has a size of K. If new cluster has no ima

    return new_clusters # should be in the format of K_clusters

```

Function performs Kmeans and returns the original and the modified images

```

In [ ]: def perform_Kmeans (image_set, k, n_runs):

    images = [np.asarray(img) for img in image_set]

    store = []
    # perform K_means to all images
    for i, image in enumerate(images):
        print("processing image = ", i)
        processed_image = my_Kmeans(k, deepcopy(image), n_runs)
        store.append(processed_image)

    # plot images before Kmeans
    plot_images (images)

    # plot images after Kmeans
    plot_images(store)

    return images, store

```

f) Run a k means clustering algorithm to compress an image based on k and n_runs

```

In [ ]: def number_of_unique_pixels (images):
    # compute the average number of unique pixels in the set of original image

    pixel_set = []
    for img in images:
        arr = []
        for i in range(len(img)):
            for j in range(len(img[0])):
                pixel = tuple(img[i, j])
                arr.append(pixel)
            pixel_set.append(len(arr))

    return sum(pixel_set)/len(images)

```

```

In [ ]: # The average number of pixels in the compressed image = 30
k, n_runs = 30, 2

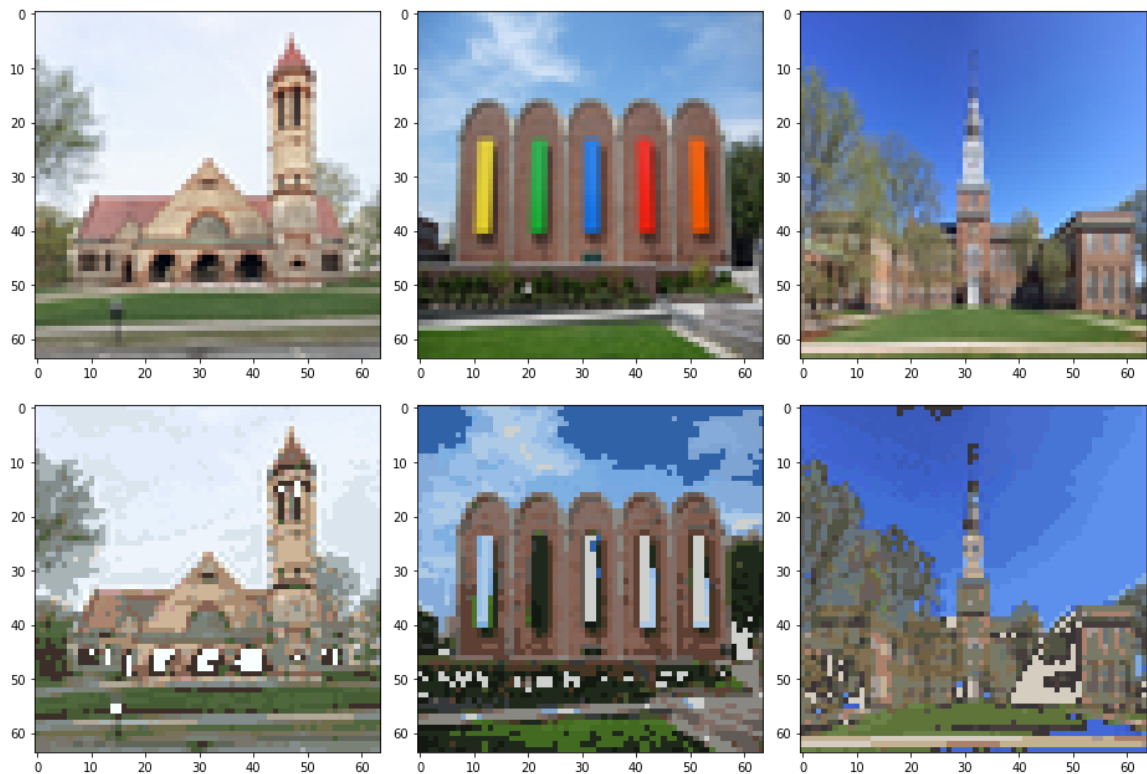
original, compressed = perform_Kmeans(new_images, k, n_runs)

```



```
original, compressed = perform_Kmeans(new_images, k, n_runs)
average_original_pixels = number_of_unique_pixels(original)
print('compression factor = {}'.format(k/average_original_pixels*100))
```

```
processing image = 0
processing image = 1
processing image = 2
compression factor = 0.732421875%
```



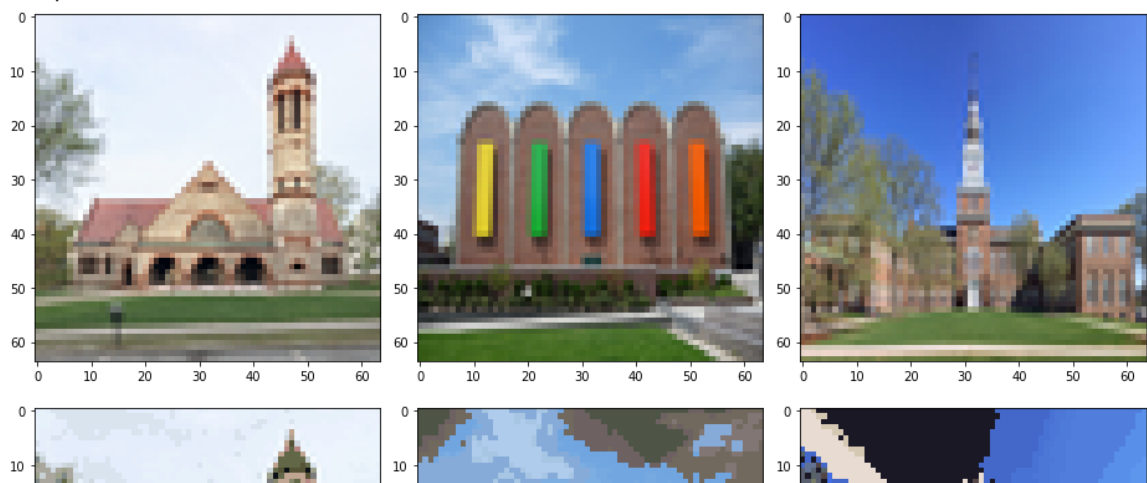
In []:

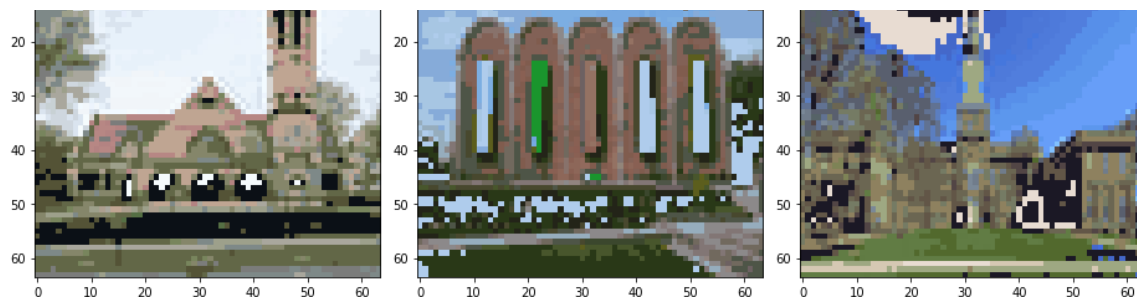
```
# The average number of pixels in the compressed image = 300
```

```
k, n_runs = 30, 20
```

```
original, compressed = perform_Kmeans(new_images, k, n_runs)
average_original_pixels = number_of_unique_pixels(original)
print('compression factor = {}'.format(k/average_original_pixels*100))
```

```
processing image = 0
processing image = 1
processing image = 2
compression factor = 0.732421875%
```





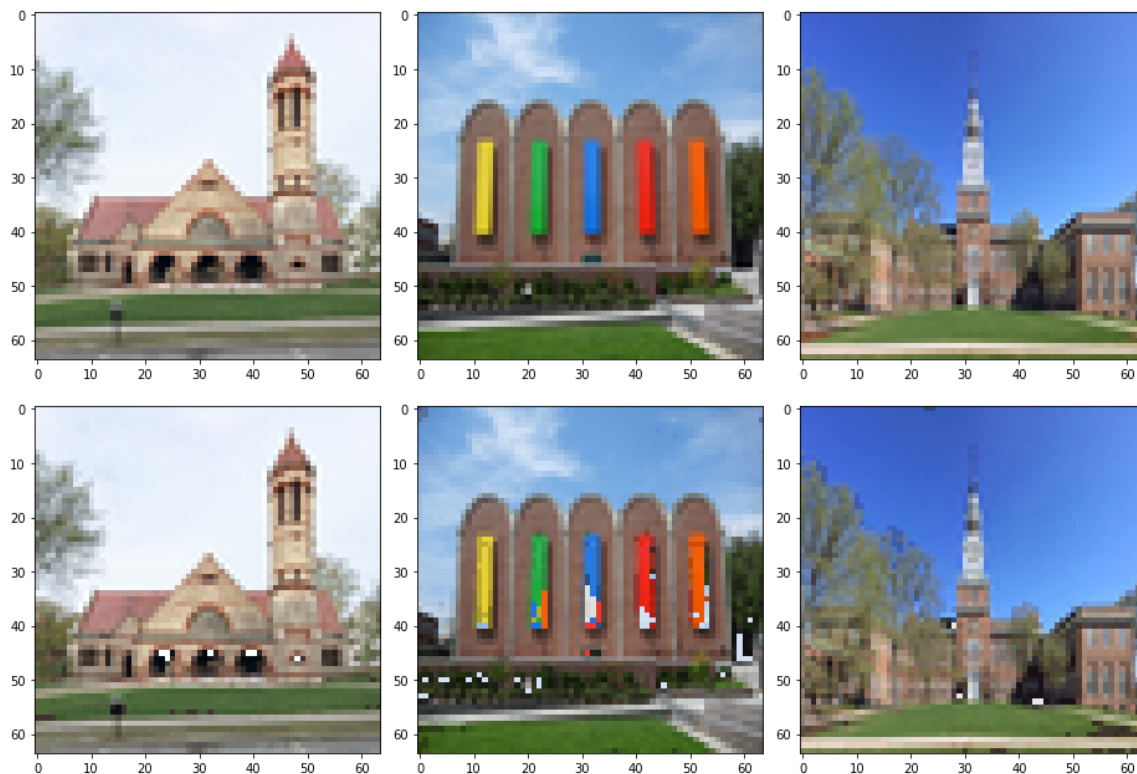
In []:

```
# The average number of pixels in the compressed image = 300
```

```
k, n_runs = 1000, 20
```

```
original, compressed = perform_Kmeans(new_images, k, n_runs)
average_original_pixels = number_of_unique_pixels(original)
print('compression factor = {}'.format(k/average_original_pixels*100))
```

```
processing image = 0
processing image = 1
processing image = 2
compression factor = 24.4140625%
```



g) Whats the effect of increasing the number of iterations, n_runs ? Whats the effect of increasing k ?

In []:

```
#TODO
```

```
...
```

```
Increasing  $n\_runs$  makes the image smoother
```

```
Increasing  $k$  will make the image contain more possible colors
```

```
...
```

h) For our stopping criterion, we used the number of iterations (n_{runs}) to finish a run. What alternative method (or

metric) could you use to automate the stopping criterion?

In []:

```
#TODO
'''
There is an optimal ratio of  $n_{\text{runs}}$  to  $k$  such that the image does not underfit
'''
```

Problem 2: Reinforcement Learning (RL): GRID-World

In this problem, the goal is to train an AI that traverses an $n \times n$ grid to find paths that lead to the bottom right corner $(n - 1, n - 1)$, from the top left corner, $(0, 0)$. Assume $n = 15$, and the terminal (end) states of the grid include the desired state $(14, 14)$ as well as areas with quicksand (death traps!). An agent gets a reward of 0 when it moves to a state that is not terminal, 1 when it moves to the desired state $(14, 14)$, and -1 when it moves into a quicksand. The game ends whenever the agent reaches a terminal state.

Use the code below to generate a heatmap representing the gridworld.

In []:

```
fig, axes = plt.subplots(figsize = (10, 6))

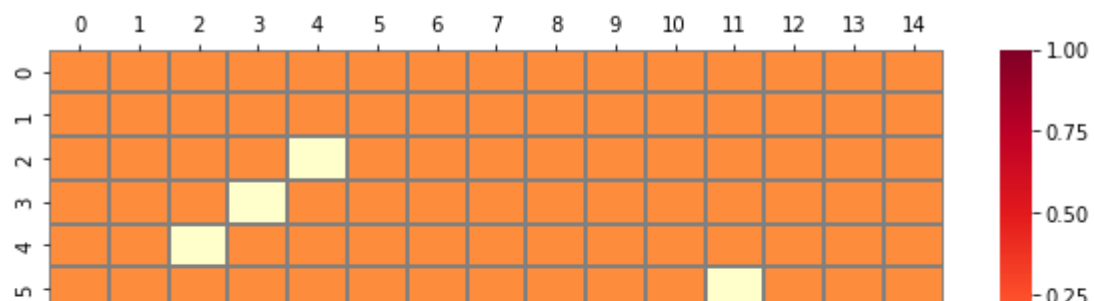
# Quick sand locations that we want to avoid
quicksand = [(2, 4), (3, 3), (4, 2), (12, 3), (13, 2), (11, 4), (11, 9), (6, 10)

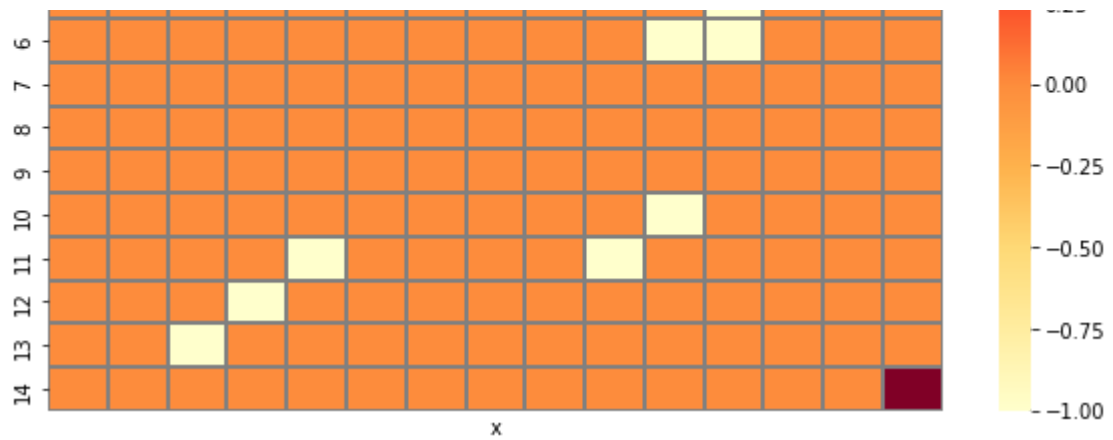
grid = np.zeros((15, 15))
for danger in quicksand: # These are signified by a reward of -1
    x, y = danger
    grid[x, y] = -1

# The location we want to traverse to! (reward = 1)
grid[14, 14] = 1

df = pd.DataFrame(grid)
sns.heatmap(df, cmap='YlOrRd', linewidths=1, linecolor='grey')
axes.xaxis.set_ticks_position("top")
axes.set_xlabel('x')

rewards = grid
```





To run any RL, we need to specify a Markov Decision Process which is a tuple comprising of S, A, P, R, γ . In this problem:

S - denotes states, which are the grid coordinates, (x, y) .

A - denotes agent actions: will consider the following 1-step movement actions (0 - *right*, 1 - *down*, 2 - *left*, 3 - *up*)

P - denotes the transition matrix $P : S \times A \times S$ which is the probability of going to state s' , from state s , via action a . We will consider a deterministic environment where $p(s, a, s') = 1$ if it's possible to take action a from state s leading to state s' , otherwise its 0

R - the reward for the goal state,

$(x = \text{gridsize} - 1, y = \text{gridsize} - 1) = 1$. The reward for quicksand locations = -1, the reward for any other state = 0.

γ - discount factor. Variable caters for preference of immediate rewards over long-term rewards. We will consider $\gamma = 0.8$

Several Reinforcement Learning algorithms exist in literature. In this problem we will examine:

1. Q-Learning
2. Deep Q-Learning

a) Q-Learning. The algorithm for Q-learning is shown below

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in S^+, a \in A(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

 until S is terminal

Source:

<https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPLBook2ndEd.pdf>

Another look at the algorithm is shown below:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{current value}} \right)}_{\text{temporal difference}}$$

new value (temporal difference target)

where r_t is the reward received when moving from the state s_t to the state s_{t+1} , and α is the **learning rate** ($0 < \alpha \leq 1$).

Note that $Q^{new}(s_t, a_t)$ is the sum of three factors:

- $(1 - \alpha)Q(s_t, a_t)$: the current value
- αr_t : the reward $r_t = r(s_t, a_t)$ to obtain if action a_t is taken when in state s_t (weighted by learning rate)
- $\alpha \gamma \max_a Q(s_{t+1}, a)$: the maximum reward that can be obtained from state s_{t+1} (weighted by learning rate and discount factor)

Source: <https://en.wikipedia.org/wiki/Q-learning>.

In short, we are updating the current q-values by adding proportions of the previous q-value, the reward for the current transition (s, a, s'); and some portion of the maximum discounted future q-values.

(a1) TODO: Initialize a q-table $Q(s, a)$ which comprises of each state s , and the possible movement actions from each states.

Dimensions should be $Q : S \times A$, where S - state space, A - action space.

Note: Table should be initialized to zeros! -Represent the q-table as a dictionary with states representing the grid locations, and a list representing the possible 4 actions (integers) from each state.

```
In [ ]: # TODO: Initialize Q-table

#Q = {i:[0 for i in range(4)] for i in range(15*2)}

# complete initialization of Q!

Q = np.zeros((15*15, 4))
F = 14
```

(a2) TODO: Write a function to determine whether a state (x, y) is terminal (end state). Terminal states are quicksand locations with a reward = -1 and the goal state with a reward of 1.

```
In [ ]: #TODO: is terminal state?
```

```

def is_terminal (x, y, rewards):
    # (x, y) - input state
    # rewards - the grid's reward function

    if rewards[x][y] == 1 or rewards[x][y] == -1:
        # Return boolean indicating is state is terminal
        return True
    return False

```

(a3) TODO: Given a current state, s , write a function to compute the set of all successive states (in 1-timestep) that an agent can traverse to as well as their corresponding actions. For example, if an agent is in state $(0, 0)$, it can go to state $(0, 1)$ by taking the right action = 0; it can also go to state $(1, 0)$ by taking the down action = 1. (NOTE: In some states, all 4 actions cannot be performed. For example, in state $(0, 0)$ an agent can't go up or left - hence your function should not consider these cases).

```

In [ ]: # TODO: Get successive states
def get_next_states (x, y):
    # (x, y) - current state

    next_states = [] # List of successive state s' from state s
    next_actions = [] # List of actions taken to move to state s' from state s

    # Complete below

    if x<14:
        next_states.append([x+1,y])
        next_actions.append(0)

    if y<14:
        next_states.append([x,y+1])
        next_actions.append(1)

    if x>0:
        next_states.append([x-1,y])
        next_actions.append(2)

    if y>0:
        next_states.append([x,y-1])
        next_actions.append(3)

    return next_states, next_actions

```

(a4) TODO: From part a, you can see the variable labeled as the **estimate of the optimal future value, maxQ**.

Using your q-table, and your next states list, compute the maxQ value as well as the action needed to get this value:

```
In [ ]: # TODO: compute the maxQ value
def get_maxQ (x, y, Q, next_states, next_actions):
    # (x, y) - current state
    # Q - table of q-values
    # next_states - list of the states that you can visit from state (x, y)
    # next_actions - list of the actions needed to move from state (x, y) to th

    #TODO: Compute maxQ:
    #If multiple actions can yield the same maxQ, select the return action

    Q_value = []

    for i in range(len(next_states)):
        Q_value.append(Q[y*F+x][i])

    maxQ=np.max(Q_value)
    Q_index = []

    for i in range(len(Q_value)):
        if Q_value[i]==maxQ:
            Q_index.append(i)

    indx_q=choice(Q_index)
    max_action=next_actions[indx_q]

    # return maximum Q_value as well as the action leading to it (from next_act
    return maxQ, max_action
```

(b1) TODO: To move within the grid, and agent can be greedy and choose actions that takes it to states with the maximum q-values.

However, this behavior has a tradeoff of not navigating other potentially beneficial states the agent might have missed in initial learning phases, hence several techniques are devised to determine which action an agent should choose. One of the most popular techniques is the e-greedy method where an agent is greedy $(1 - \alpha)$ times, but also explores other states at a rate of α .

Implement the e-greedy method below:

```
In [ ]: def e_greedy (alpha, x, y, max_action, next_actions):
    # (x, y) - current state
    # max_action - greedy action
    # next_actions - set of all actions the agent could potentially take
    # alpha - probability of taking a non-greedy action

    if random()>alpha:
        action = max_action

    else:
        action = next_actions[randint(1,len(next_actions)) - 1]

    # Complete here!
```

```
return action # action after e-greedy method
```

(b2) TODO: Now lets compile all the components we need to write the q-learning algorithm.

```
In [ ]: def QLearning (rewards, Q, gridsize = 15, alpha = 0.01, discount = 0.8, learning_rate = 0.1):
    # (x, y) - initial state for the agent
    # alpha - exploration probability during action selection
    # Q - q-table
    sumQ = [] # List of the sums of q-values for each epoch
    Trajectories = [] # List of trajectories collected for each individual

    for epoch in range(epochs):
        x, y = 0, 0 # start from initial state

        new_trajs = [(x, y)] # store trajectory (paths) sequences

        while not is_terminal(x, y, rewards): # run until we reach a terminal

            # get the potential successive states and actions here
            next_states, next_actions = get_next_states(x, y)

            # compute maxQ
            maxQ, max_action = get_maxQ (x, y, Q, next_states, next_actions)

            # e-greedy method to choose either the max_action, or a random action
            action = e_greedy(alpha, x, y, max_action, next_actions)

            # compute new state based on the previous action
            index = next_actions.index(action)
            x_next, y_next = next_states[index]

            # TODO: UPDATE Q-VALUES HERE:
            #if len(Trajectories)==0:
            # index_old=0

            P = Q[y*F+x][index]
            Q[y*F+x][index] = P + learning_rate*(rewards[x_next][y_next] + discount*P)

            # UPDATE Q-value - should be a one liner!
            # update states

            x, y = x_next, y_next

            #index_old=index

            # add new state to trajectories
            new_trajs.append((x, y))

        sumQ.append(sum([sum(Q[k]) for k in range(len(Q))]))
        #print(f"sumQ=", sumQ)
        Trajectories.append(new_trajs)

    return sumQ, Trajectories
```

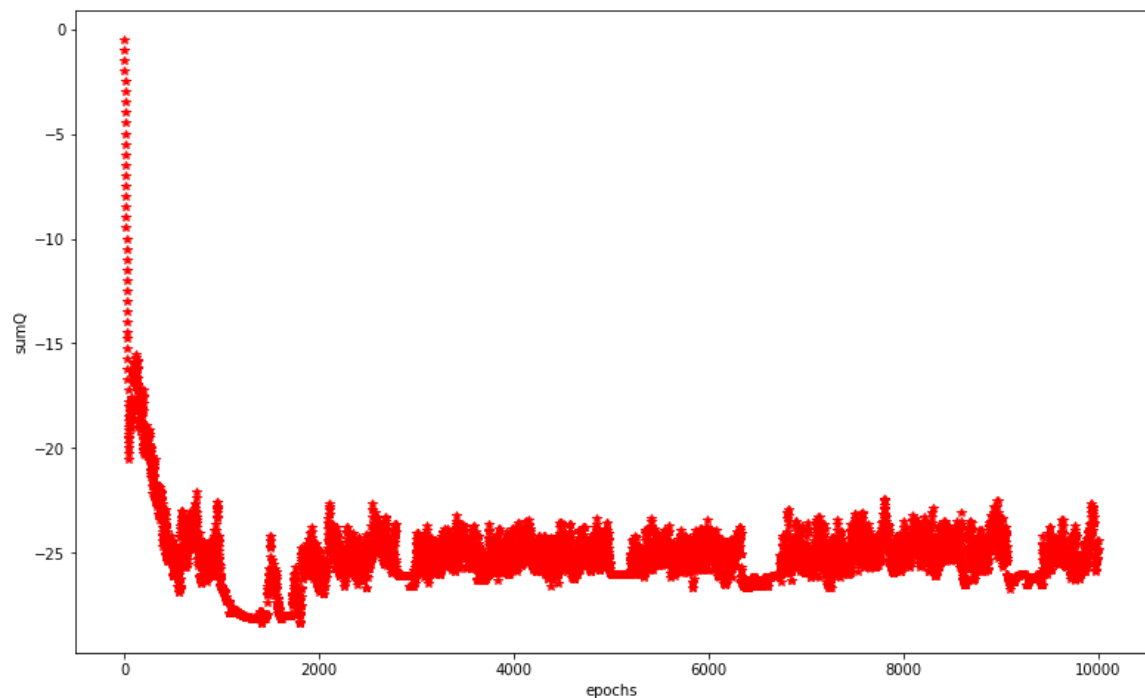

(b3) Now perform Q-learning below:

```
In [ ]: Qvals = deepcopy(Q)
        sumQ, Trajectories = QLearning(rewards, Qvals)
```

(c) Plot a scatter plot of the sum of q-values (sumQ). How many trajectories are needed to reach convergence?

```
In [ ]: fig, axes = plt.subplots(figsize = (13, 8))
        axes.plot(sumQ, '*', color = 'red')
        axes.set_xlabel('epochs')
        axes.set_ylabel('sumQ')
```

```
Out[ ]: Text(0, 0.5, 'sumQ')
```



(d1) TODO: The list *Trajectories* records each trajectory obtained while performing Q-learning. Using the first 10 trajectories plot a heatmap showing whether a state has been visited or not?

```
In [ ]: # TODO: Complete the visitations array for the first 10 trajectories
        # indicate a state that has been visited as a one, and not visited as a 0

        is_visited = np.zeros((15, 15))

        # fill up visitations here
        for Traj in Trajectories[:10]:
            for state in Traj:
                x, y = state
                is_visited[x, y] = 1
```

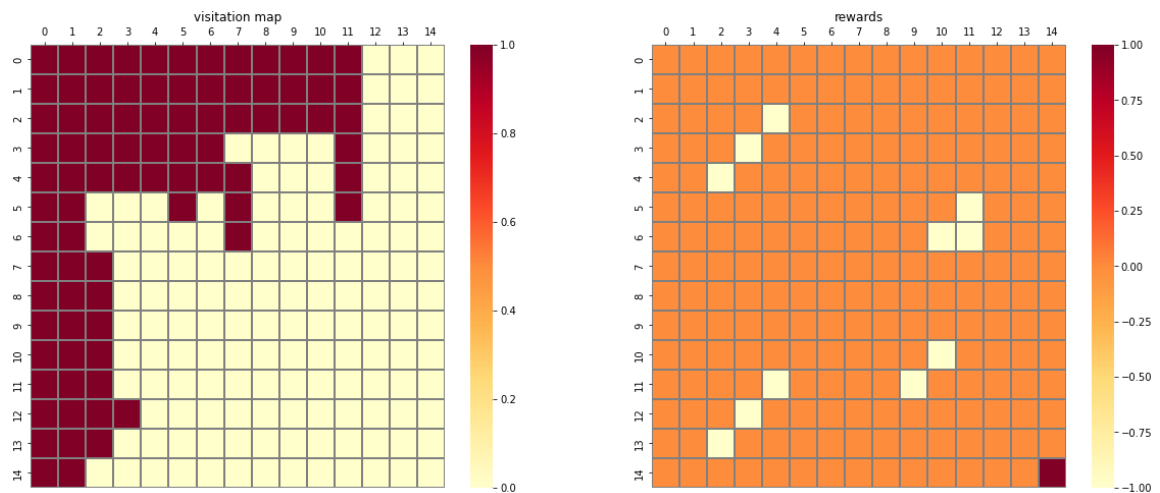
```
visitations = is_visited
```

```
In [ ]: fig, axes = plt.subplots (1, 2, figsize = (20, 8))

df = pd.DataFrame(visitations)
sns.heatmap(df, cmap='YlOrRd', linewidths=1, linecolor='grey', ax = axes[0])
axes[0].xaxis.set_ticks_position("top")
axes[0].set_title('visitation map')

df1 = pd.DataFrame(rewards)
sns.heatmap(df1, cmap='YlOrRd', linewidths=1, linecolor='grey', ax = axes[1])
axes[1].xaxis.set_ticks_position("top")
axes[1].set_title('rewards')
```

```
Out[ ]: Text(0.5, 1.0, 'rewards')
```



(d2) Plot a similar plot for the last 10 trajectories

```
In [ ]: visitations = np.zeros((15, 15))

print(len(Trajectories[len(Trajectories) - 5: ]))

# fill up visitations here
for Traj in Trajectories[len(Trajectories) - 5: ]:
    for state in Traj:
        x, y = state
        visitations[x, y] = 1

print(visitations)
```

```
5
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1.]
```

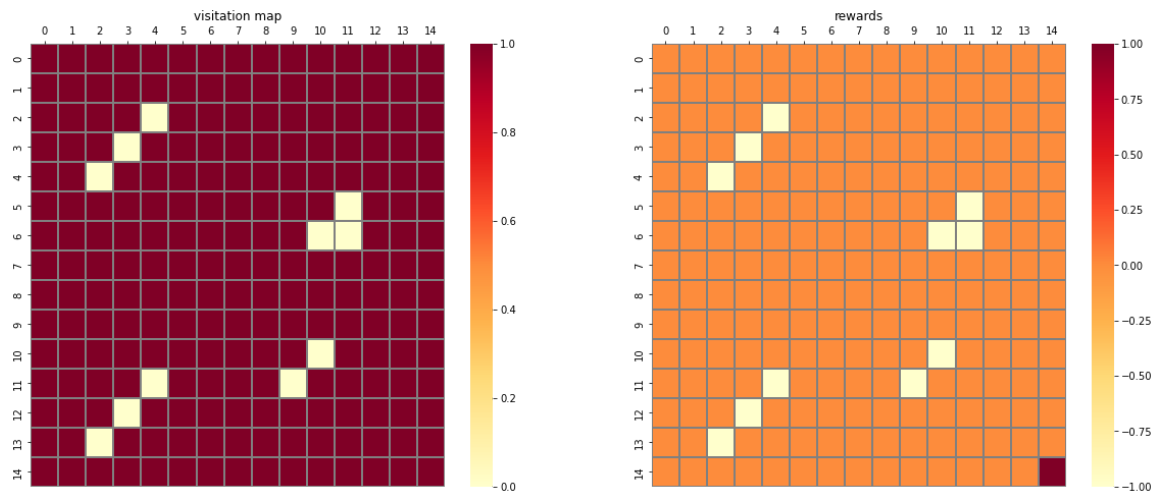
```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1.]
[1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1.]
[1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

```
In [ ]: fig, axes = plt.subplots (1, 2, figsize = (20, 8))

df = pd.DataFrame(visitations)
sns.heatmap(df, cmap='YlOrRd', linewidths=1, linecolor='grey', ax = axes[0])
axes[0].xaxis.set_ticks_position("top")
axes[0].set_title('visitation map')

df1 = pd.DataFrame(rewards)
sns.heatmap(df1, cmap='YlOrRd', linewidths=1, linecolor='grey', ax = axes[1])
axes[1].xaxis.set_ticks_position("top")
axes[1].set_title('rewards')
```

```
Out[ ]: Text(0.5, 1.0, 'rewards')
```

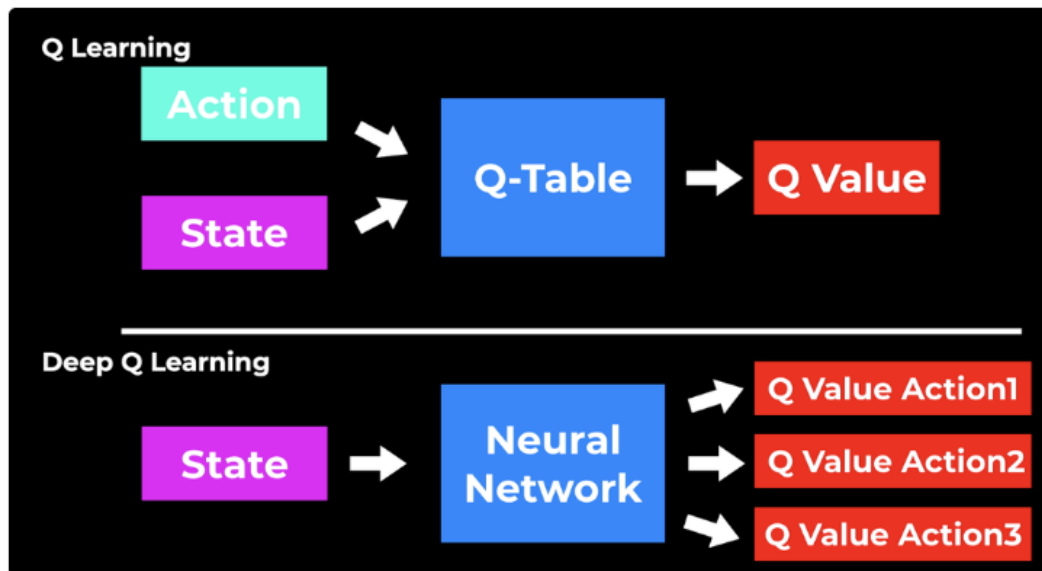


(d3) Comment on differences between observations in d1 vs d2? Does Q-learning find a policy that avoids obstacles?

```
In [ ]: ...
The accuracy of the first 10 trajectories is low, and the accuracy of the last
Yes, it is.
...
```

BONUS: DEEP Q-LEARNING In Q-learning, we create a Q-table that has a size of $S * A$ where S is the state-space, A is the action-space. When we have a large state space, updating the q-values becomes computationally expensive hence deep learning techniques have been incorporated to scale training. One such application is to have a Q-network that takes a state as input, and predicts the q_values of the state under different

actions. In this case, we don't have to store all the states in a table, and we can use the network to predict q-values.



Source: <https://www.assemblyai.com/blog/reinforcement-learning-with-deep-q-learning-explained/>

(e) Write some detailed psuedocode (and maybe implement if you can!) on how you could perform deep Q-learning using the architecture above. You can follow the structure of algorithm in (a)

```
In [ ]: '''It's bouns'''
```

