AI & Automation Exam 1

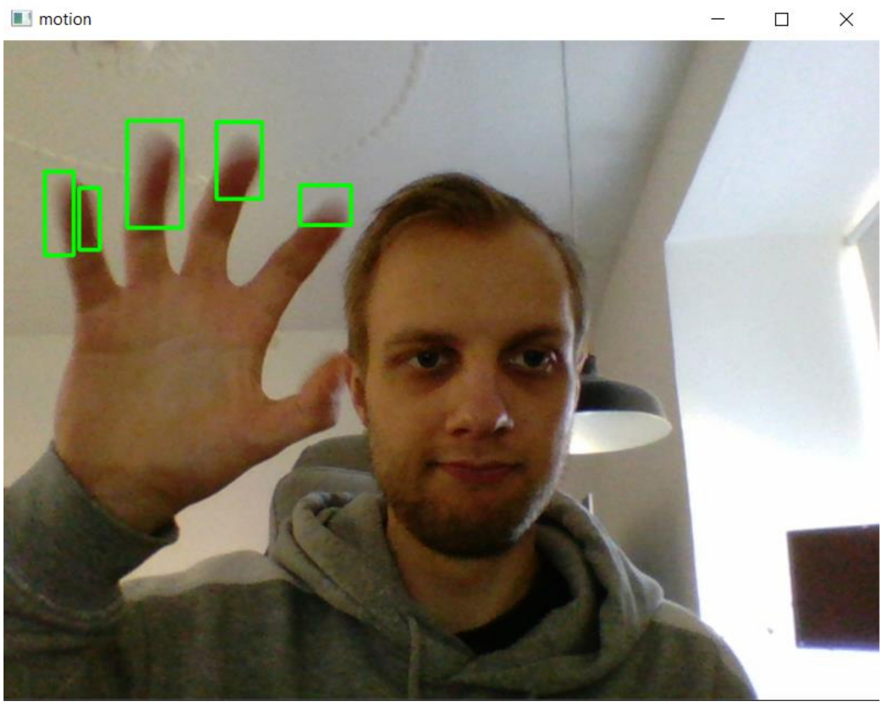
04/04/2022

Mads Hegewald (EAAMHEG)

Exercise 1

When exercise1 is run it will automatically start with motion detection. To change to full body detection press 'p', for face recognition press 'f' and to go back to motion detection press 'm'. The detection mode can also be seen in the window name at the top left.

For the motion detection we take 2 frames and compare them and find the difference between them. After this we convert it to grayscale to make it less complex. This is because RGB colors have three dimensions while grayscale images have one dimension. Then we smoothen the image with Gaussian blur to remove some of the noise (outlier pixels) to easier. Thresholding will convert the image to binary and help separate the objects (foreground) from its background. Finally, we look for contours in the image, which is basically just identifying the objects shape, and then making a rectangle around it.



For the full body detection, we use a HOG (Histogram of Oriented Gradients) feature to extract the shapes (objects) in the image and look for a full body. We can adjust the sensitivity of what we would classify as a person with the weights attribute in the code, where the lower the value the more likely it will define the object as a full body. To find the color we would have to define lower- and upper limits for each color with its BGR values. With this we can look for certain colors (within the BGR values) in the image. My thought process was making a new frame from the coordinates detected by HOG:

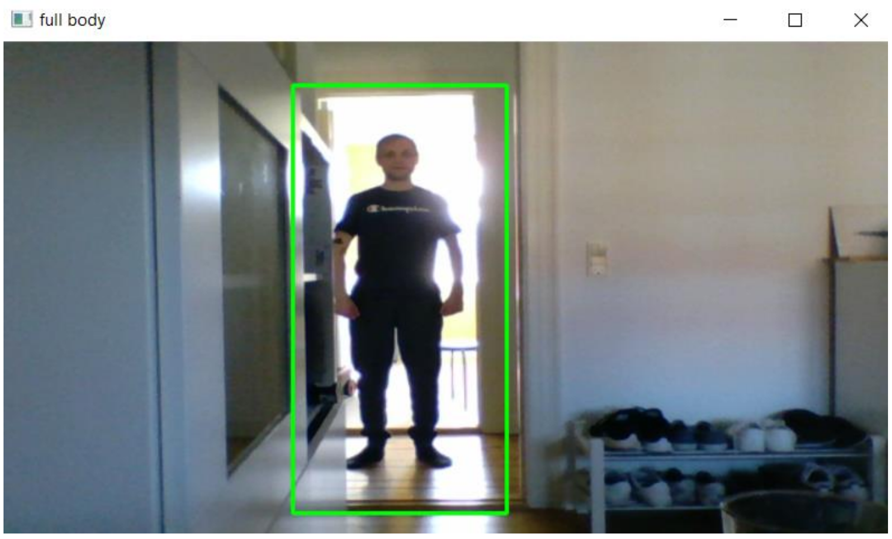
```
frame2 = frame[y:y+w, x:x+h]
```

And then using the mask to look for contours in the new frame. This however resulted in it detecting most of the colors at the same time in the image, this could probably be solved with finding the most dominant color. Another approach I looked at was just getting the color values from the picture itself:

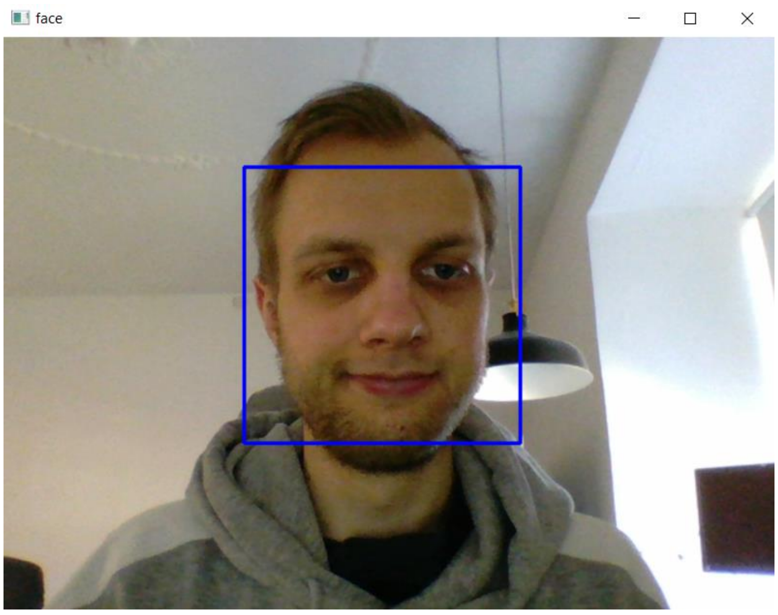
```
color = frame[x, y]
print(color)
```

```
[30 30 43]
[33 32 36]
[32 31 40]
[39 41 46]
```

This only worked some of the time and was not a solid solution to the problem.



The face detection uses a Haar cascade, where the Haar features help identify edges or lines and make it easier detecting a face.



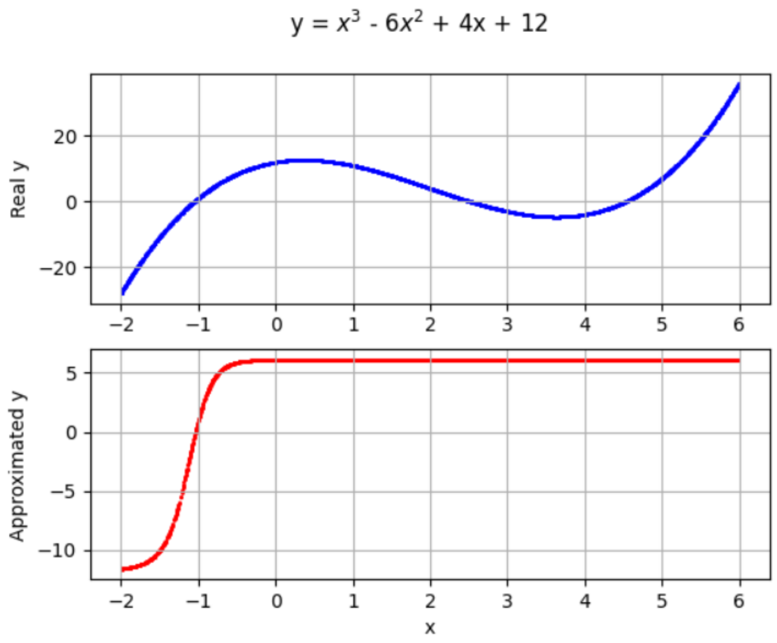
Exercise 2

A

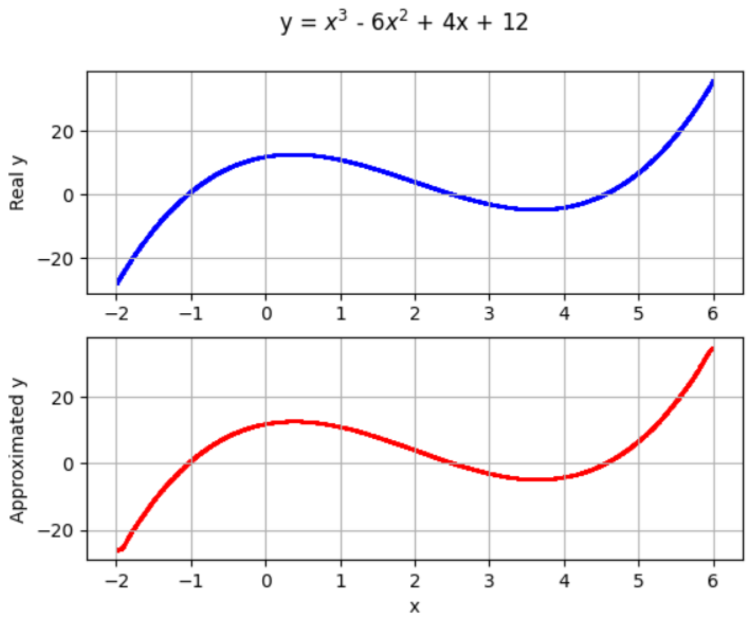
To get the different points for our x is have used the following line:

```
x = np.random.uniform(low=-2, high=6, size=(5000, 1))
```

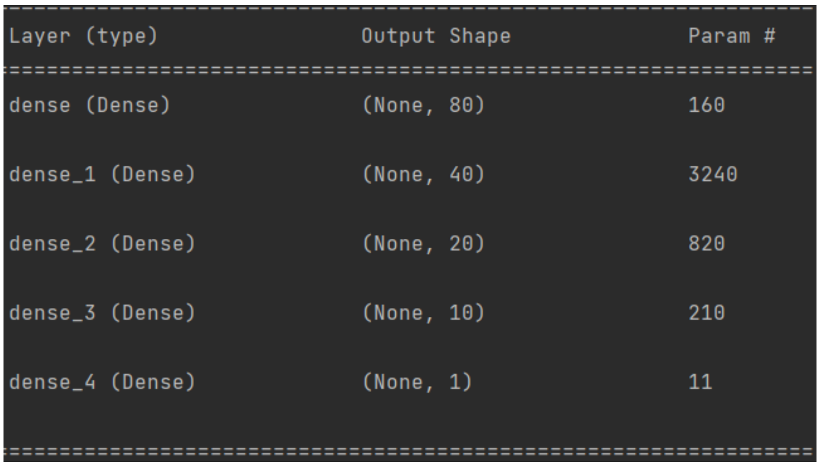
Gives 5000 random points between -2 and 6.



The above experiment was run with the settings of 2 hidden layers with 20 and 10 neurons. The number of points used was 2500 and was run for 50 epochs. I used sigmoid as activation function in the hidden layers because it resembles the mathematical function which it is trying to learn. The neural net did not learn function, most likely because the network isn't big enough because of insufficient epochs and points.



For the experiment above I changed the neural net to the following:

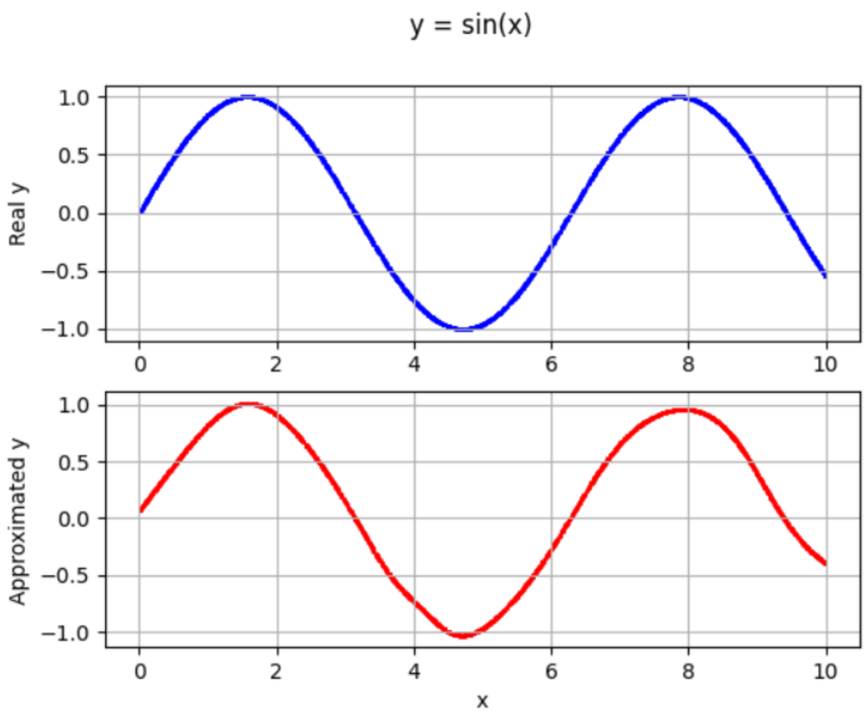


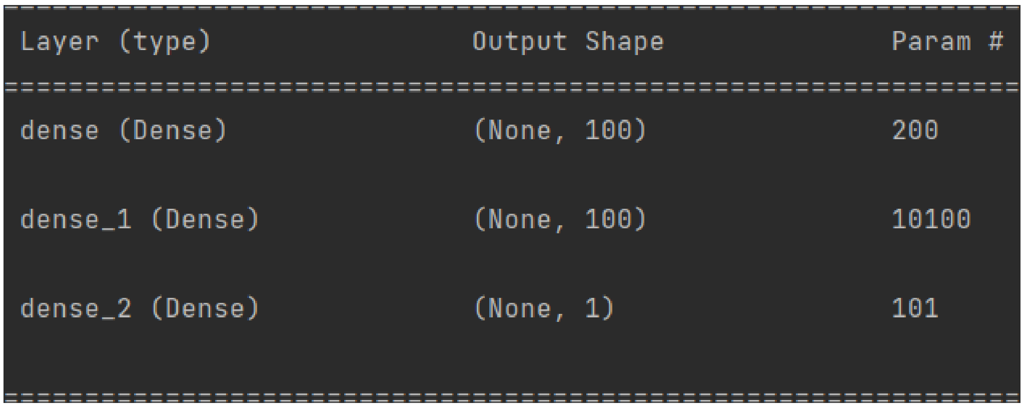
```
=================================================================
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 80)                160

dense_1 (Dense)              (None, 40)                3240

dense_2 (Dense)              (None, 20)                820

dense_3 (Dense)              (None, 10)                210

dense_4 (Dense)              (None, 1)                 11

=================================================================
```

I also decided to increase the number of points to 5000 and make it run for 150 epochs.

Sometimes it would learn the function in less than 100 epochs and other times it wouldn't learn the function at all. But I have found most success with above given parameters.

The neural network to predict the sin(x) function consist of 2 hidden layers with 100 neurons each. For the number of points used was 2500, which I felt was standard for this type of problem. It goes through 150 epochs and gets better at predicting once it nears the end. The result is not perfect, but I would say that it is not far away.





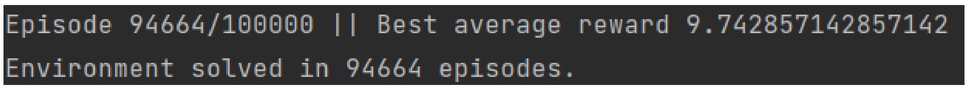
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               200

dense_1 (Dense)              (None, 100)               10100

dense_2 (Dense)              (None, 1)                 101

=================================================================
```

(activation=sigmoid, epochs=150, data points=2500)

Exercise 3.a

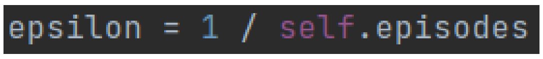
For my experiments I made changes to Taxi_2.py where we look to receive an average reward (from last 100 episodes) greater than 9.7 to win the game.

In an experiment with gamma of 0.6, learning rate at 0.1 and epsilon at 0.0001 the agent solved the environment close to the end.



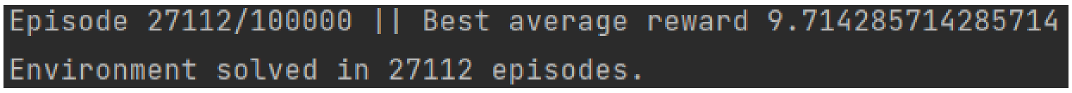
```
Episode 94664/100000 || Best average reward 9.742857142857142
Environment solved in 94664 episodes.
```

In another experiment I used another epsilon in the given code:



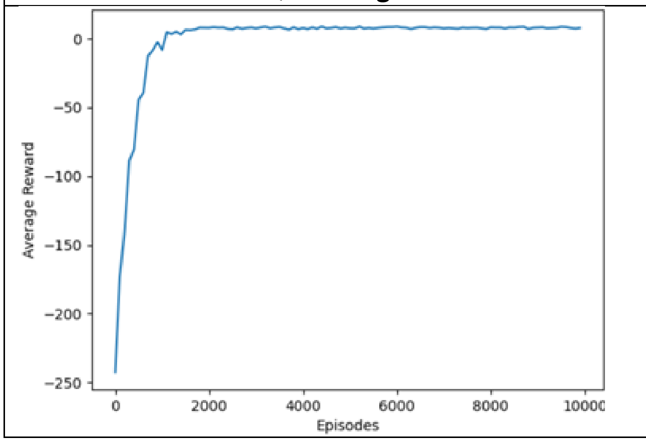
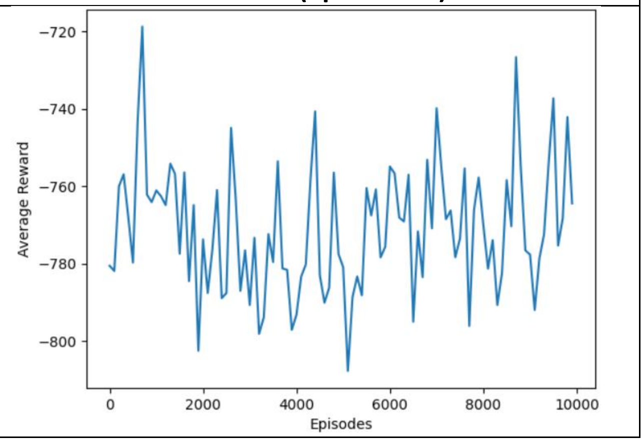
```
epsilon = 1 / self.episodes
```

The result for this was better than before, where the epsilon is larger and gradually decreases. So, for the epsilon it is better to start with a higher value to explore and develop a strategy. After exploring we want to focus more on exploitation to get the average reward we are looking for. Here we manage to reach the goal of an average reward larger than 9.7 in fewer episodes.



```
Episode 27112/100000 || Best average reward 9.714285714285714
Environment solved in 27112 episodes.
```

For an agent that only takes random actions we must set the exploration rate to 1. Comparing the average reward per 100 episodes with the agent using q-learning we get the following results:

Q-learning	Random (epsilon = 1)
	

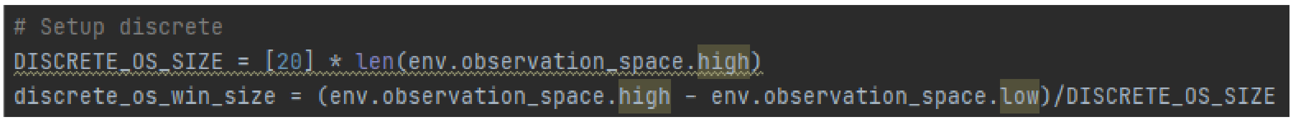
Here we see that the agent does not actually receive any consistent or significant average reward when only taking random actions. We can see that q-learning does work compared to always just taking random actions.

Exercise 3.b

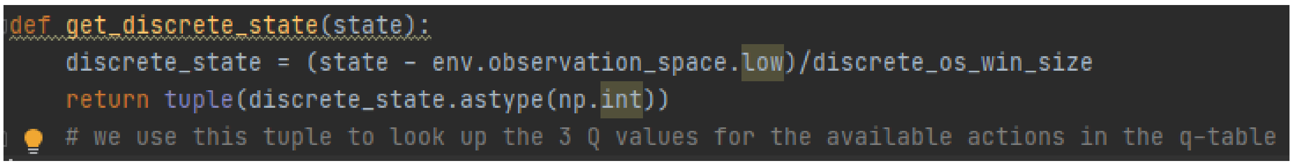
A

The "infinite number of states" problem is caused by the number of possible positions existing between -1.2 and 0.06 and the same with velocity our between -0.07 and 0.07. When the state space is infinite we can't use q-learning to solve the problem. To make it so that the state space is not infinite we can discretize the state space. This means we are basically making categories for the space, where we can choose the best action to take based on the category. When making these categories we don't want too many or too few.

Basically, the following lines:



```
# Setup discrete
DISCRETE_OS_SIZE = [20] * len(env.observation_space.high)
discrete_os_win_size = (env.observation_space.high - env.observation_space.low)/DISCRETE_OS_SIZE
```



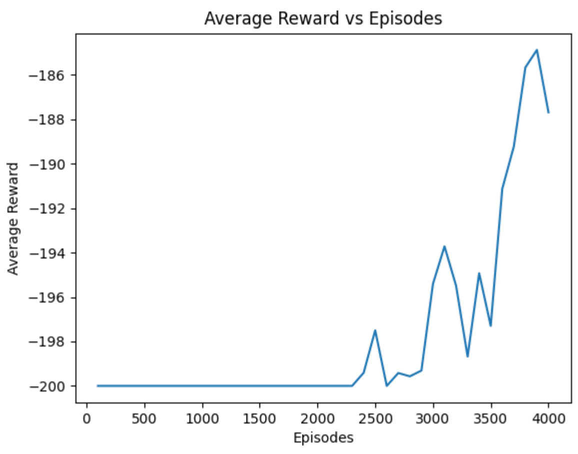
```
def get_discrete_state(state):
    discrete_state = (state - env.observation_space.low)/discrete_os_win_size
    return tuple(discrete_state.astype(np.int))
    # we use this tuple to look up the 3 Q values for the available actions in the q-table
```

It returns a tuple with the category for the position and velocity, which solves the infinite states problem, and we can determine the next action to take based on this.

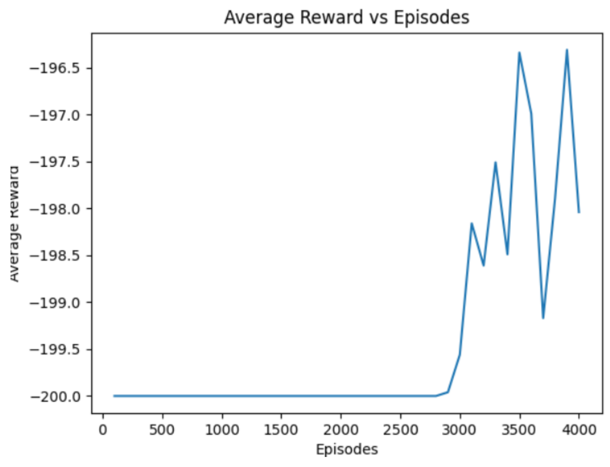
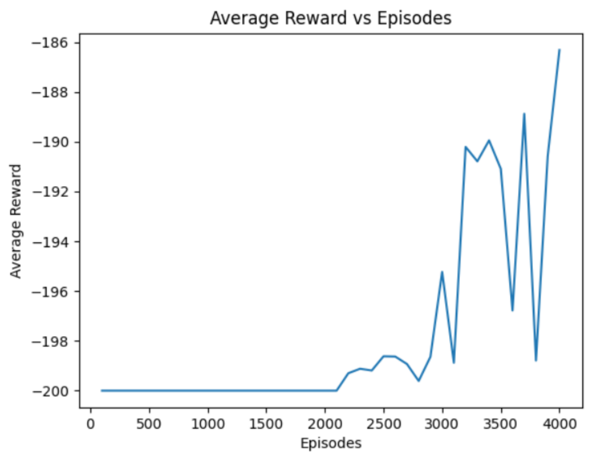
Another way to solve this could be by using a DQN (Deep Q Learning), where we can use a neural network to approximate the q-function.

B

Just running the code handed out for the mountain car problem we get a decent average reward when we get past the 3500 episodes mark. This was with a learning rate of 0.2, discount rate at 0.9, epsilon at 0.8 and 4000 episodes.

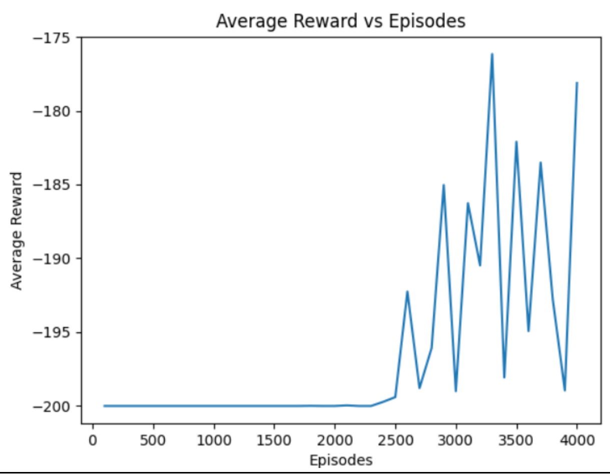
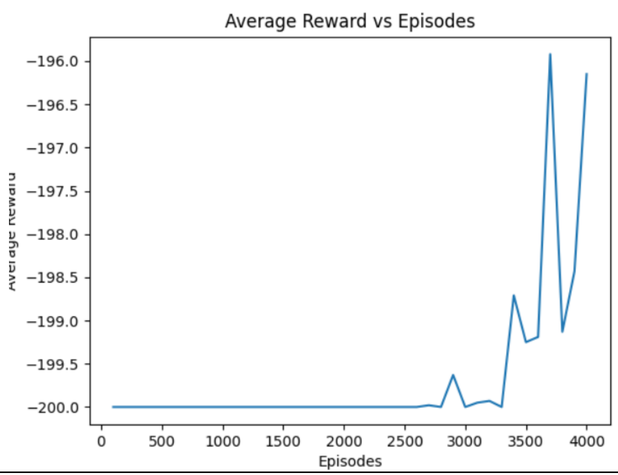


To make it learn faster, one possibility could be changing the number of states used, which is [20*20] in the given code.

[10*10]	**[30*30]**
	

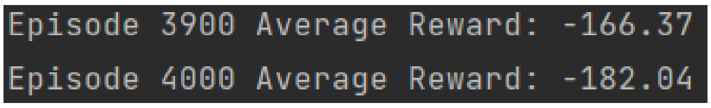
Too few states will cause the agent to not make the correct adjustments based on the current state and it takes relatively longer before it gets a reward. When the states are increased it seems to start learning the problem at around 2100 episodes, which is better than the one with less states, but too manty states can also be bad for the agent. I tried another experiment with [50*50] and 4000 episodes where it did not yield any reward.

For learning rate, I tried increasing it too 0.4, 0.8 and a decrease to 0.05.

0.05	0.4
	

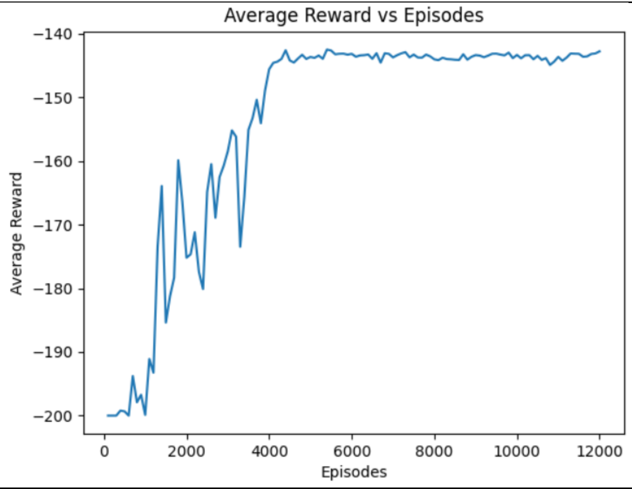
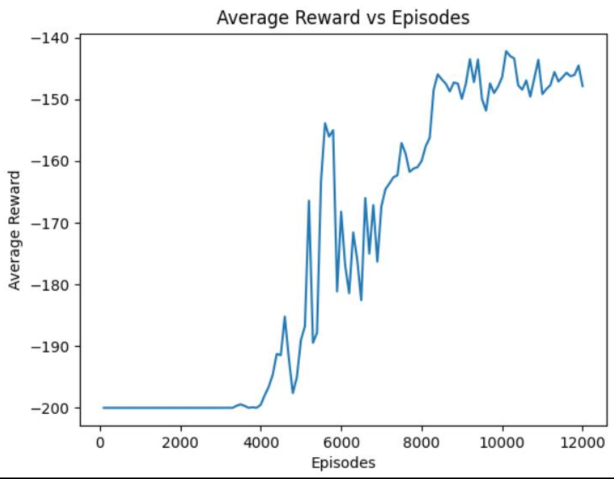
In the experiment with learning rate of 0.05 it starts receiving a reward at the same number of episodes, but the average reward is very unstable. Increasing the learning rate does not make the agent learn faster either.

Another way to possibly make it learn faster is changing the discount rate, which in the given code is at 0.9. The experiments where I lowered the discount rate didn't give a better result and I didn't make the agent learn it quicker either. Only when I increased the discount rate to 0.99 did I get a different result which is noteworthy, where I reached the best average reward gotten so far:



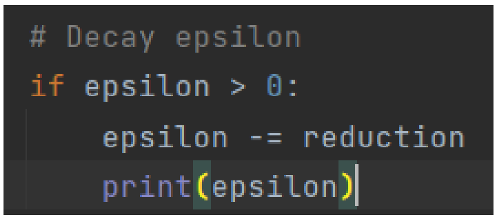
```
Episode 3900 Average Reward: -166.37
Episode 4000 Average Reward: -182.04
```

Lastly, I tried changing the epsilon to see what would happen if the agent focused on exploration, and the same for exploitation. I decided to increase the episodes to 12000 to get some more data, since in one of the experiments the actions taken is random (epsilon is at 1).

0.2	1
	

It is obvious that the experiment where it takes greedy actions learns faster and yields a better result than what I have previously seen.

One possibility to make it learn more effectively could be implementing the epsilon dynamically, where at the beginning is a high value making it explore and learn the different actions. Once it reaches higher episodes it can begin lowering the epsilon and taking more greedy actions and focus on exploitation. The decay is already implemented in the code here:



```
# Decay epsilon
if epsilon > 0:
    epsilon -= reduction
    print(epsilon)
```

The epsilon is reduced by the reduction attribute for every episode, making it take more greedy actions the further we get.

Exercise 4

a)

In q-learning the agent traverses the board based on which action is going to give the highest reward, and occasionally the agent takes a random action depending on the epsilon. This results in the agent learning the most optimal route from start to goal after exploring the board. However, if we reduce the learning rounds from 500 to 30 or 50 it will cause the agent to most likely not find an optimal route, or not finding a route at all. Or in other words the q-table is not sufficiently updated, and the agent doesn't find the optimal route.

b)

Increasing the exploration rate will cause the agent to take more random actions, this would be good to explore more of the board, but not so much when traversing the optimal route, which is along the cliff. The agent will be more likely to step onto the cliff and lose the game the higher the exploration rate. Lowering the exploration rate causes the agent to always(almost) reach the goal once a walkable route has been explored, but this will also cause the agent to not learn anything new.

In my experiments with changing the learning rate they all still manage to find the same route from start to goal. My guess as to why, is because of there only being rewards -1 and -100, where the agent can't randomly gain a large reward from a random action.

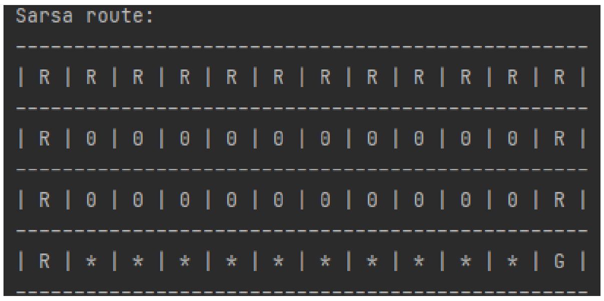
c)

We set the exploration rate to 0 to always take actions where the agent gets the highest reward. After the first agent has updated the q-table with the values for the optimal route, which should be along the cliff after 500 rounds. If we set the exploration rate to 0.2 the new agent will sometimes take random actions when it already knows the most optimal route. This would not be useful unless the board has changed, and the agent is required to explore for a new route.

d)

The reason why SARSA finds another route compared to q-learning is because of it looking at the current state-action combo as well as the next state-action combo. This means SARSA will look at future possible penalties/rewards, which makes the method more conservative. Because of this it will look for a safer route compared to q-learning, which will focus on taking the optimal route along the cliff.

To support the statement of SARSA being more conservative I guessed that an increase in exploration rate will most likely cause the agent to take an even safer path. In an experiment I increased exploration rate to 0.5 and got the following route:



```
Sarsa route:
-------------------------------------------------
| R | R | R | R | R | R | R | R | R | R | R | R |
-------------------------------------------------
| R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | R |
-------------------------------------------------
| R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | R |
-------------------------------------------------
| R | * | * | * | * | * | * | * | * | * | * | G |
-------------------------------------------------
```

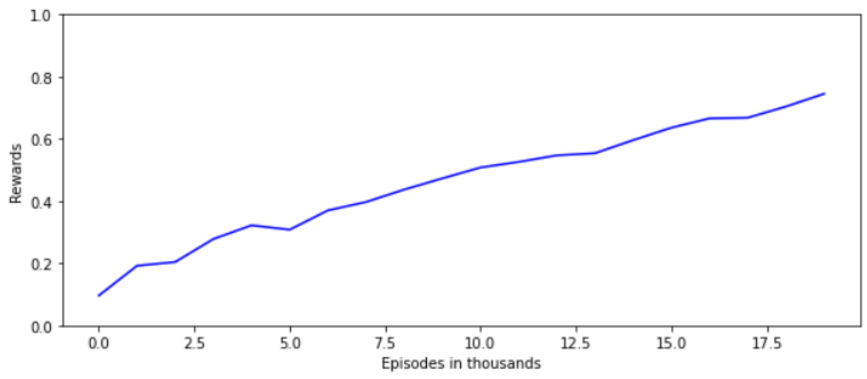
If we wanted to have the SARSA agent walk along the same route as the q-learning agent, it would only be possible once it has learned sufficiently about the board and can take greedier actions without risking the penalty.

Exercise 5

a)

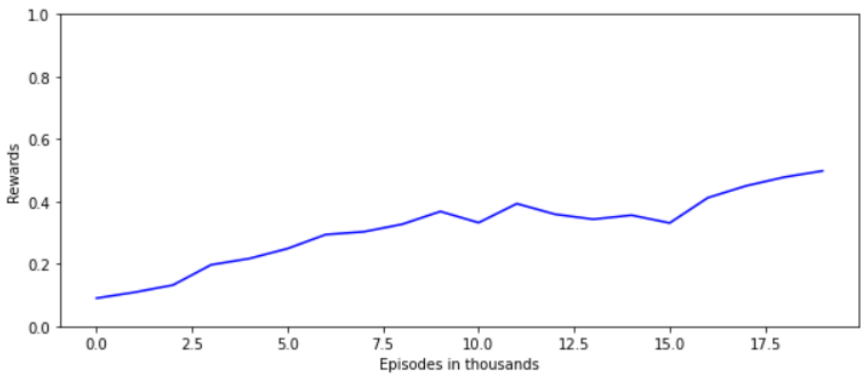
Simply changing the learning rate (alpha) did not give a better result than the one handed out, where we got a result of around 7.15 as the average reward at 20,000 episodes. For one of my experiments, I tried changing the learning rate to 0.4 and got an average reward of 0.685. In the next attempt I raised the learning rate to 0.8 and got a worse average reward (0.617). Only increasing the learning rate might make the agent learn a walkable path relatively fast but it might not be the most optical path. For the last experiment I lowered the learning rate to 0.01 and got the best result so far: (average reward = 0.745)

```
num_of_episodes = 20000
max_steps_per_episode = 200

learning_rate = 0.01
discount_rate = 0.99

exploration_rate = 1
```



Next, I did some experiments with changing the discount factor (gamma), where the default is 0.99 and gives us the reward of around 7.15. The intention of the discount factor is to secure future reward instead of immediate reward. The closer the value to 0 the more focus on immediate reward, which is not what we want in this problem since we want to reach the final goal state. Below is the experiment with a discount rate at 0.8.

```
num_of_episodes = 20000
max_steps_per_episode = 200

learning_rate = 0.01
discount_rate = 0.8

exploration_rate = 1
```



The epsilon is used for sometimes taking a random action to learn possible new paths in the q-table. Here it is set to 1 from the beginning and gradually decreases, which is seen in the following line:

```
exploration_rate = 1 - np.log(episode + 1) / np.log(num_of_episodes + 1)
```

As mentioned before, this causes the agent to start with random actions and as we go along it starts to focus on exploiting what it has learned through the random actions. The possible changes to the epsilon would be to keep it the same throughout the entire run. This would however cause the agent to sometimes take random actions when it has learned enough and should be focusing on greedy actions.

b)

The difference between SARSA and q-learning is primarily seen in the way Q is updated. As mentioned previously q-learning will take the action with the most reward (greedy policy), while SARSA will look at the states and actions for the next action.

- **SARSA:**
  $$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$
- **Q-Learning:**
  $$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$
- **Expected SARSA:**
  $$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \sum_a \pi(a|s_{t+1}) Q(s_{t+1}, a) - Q(s_t, a_t))$$

To change the code to SARSA we must change the update method, which is as following:

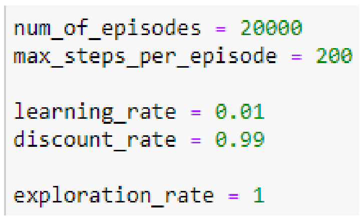
```
# q-learning update
Q_table[state, action] = Q_table[state, action] * (1 - learning_rate) \
+ learning_rate * (reward + discount_rate * np.max(Q_table[new_state,:]))
```

I defined a method to select the next action based on the given state, so it would be possible to get the action for the next state. To update the q-table with SARSA implementation I changed the update code to the following:

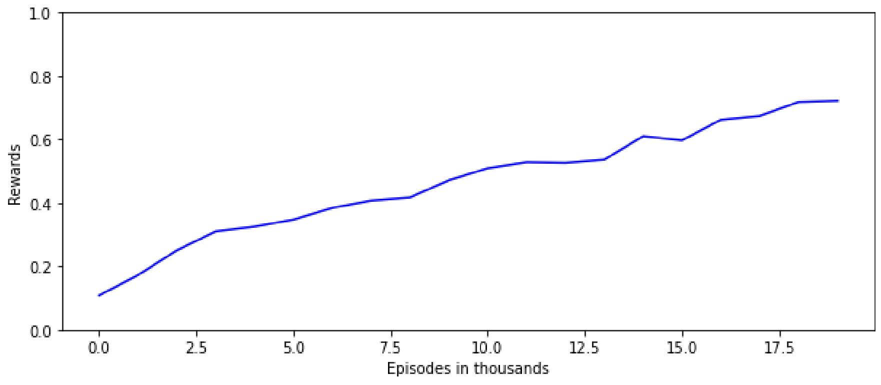
```
# Sarsa update
Q_table[state, action] = Q_table[state, action] + learning_rate * \
((reward + discount_rate * Q_table[state2, action2]) - Q_table[state, action])
```

The code runs and we get the following results:

It is possible to solve the problem with SARSA and we receive a satisfactory result.



```
num_of_episodes = 20000
max_steps_per_episode = 200

learning_rate = 0.01
discount_rate = 0.99

exploration_rate = 1
```


Rewards
Episodes in thousands