SIT215 PROJECT

IFTEKHAR QURESHI [218676618]

LINK TO GITHUB REPOSITORY FOR CODE:
https://github.com/yourboyifte/Project-AI.git

OCTOBER 10, 2020

Table of Contents

Taxi problem

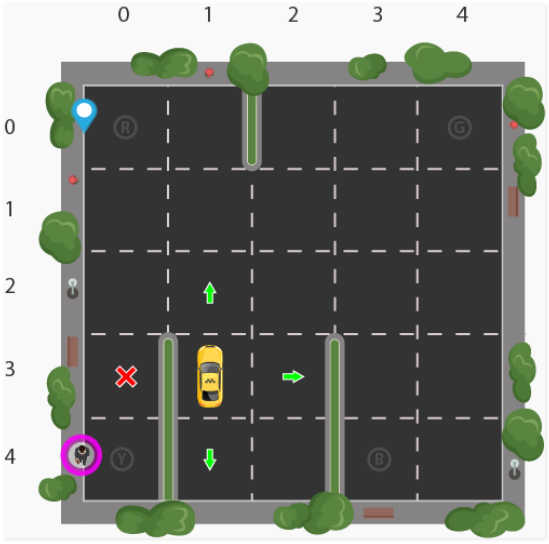
The taxi problem uses Reinforcement learning, an area of machine learning that is concerned with agents taking the right actions in a particular environment. The actions taken by the agents improve with time. This is the goal of reinforcement learning; making the agents learn the optimal way of behaving in an environment by themselves.

For solving the taxi problem using Reinforcement learning, we need to teach a taxi to pick up and drop off passengers at the right location, effectively building it into a self-driving cab. The **agent** in this scenario will be the driver of the taxi and the **environment** is the roadmap that will be created for the taxi. The roadmap will contain all the possible locations the taxi can travel including the pickup location of the passenger and the drop-off location. In short, we want our smart cab to have these three important criteria-

- Pick-up from the passenger from the correct location and drop off to the correct location.
- Try to reach the destination at the least possible time by taking the best route.
- Ensure traffic rules are followed and passengers are safe in the trip.

The Reinforcement learning model of this problem will be created based on these important features-

- The agent will receive **rewards** for every possible action it takes. There will be high positive reward for a successful drop-off and a penalty for an unsuccessful one. For every minute it takes to reach the destination, the agent will receive a slight negative reward. This will ensure that in time, the agent selects the paths that will result in the least negative cumulative sum of rewards.
- As I mentioned before, we will have a roadmap of the taxi. This roadmap is the **state space** of the taxi containing all the possible states the taxi could be in. Based on the state, the taxi will take the appropriate **action**.



Roadmap

For purpose of training our smart taxi, we will now follow this certain roadmap. This is a 5x5 grid, containing 25 possible locations and forming one part of our state space. The current location of the taxi is at coordinate (3,1). There are 4 possible pickup and drop-off location: [R, Y, G, B]. Our passenger is in Y and wish to reach location R. First our cab has to reach location Y and then venture its journey to R. In total we have (5*5)*5*4 = 500 states- 25 for possible taxi locations, (4+1) 5 for possible passenger locations(with additional 1 accounting for the state of the passenger being in the car or not) and 4 for destination locations.

Now for reaching these destinations we will have to take appropriate actions as mentioned before. There can be six possible actions in this state space at a certain state that will form our **action space-**

1. South
2. North
3. East
4. West
5. Pickup
6. Drop-off

For our illustration above, it can be seen that the cab can move North, South and East as depicted by the green arrows, but it is not possible to go west because of the wall. Obstructions are common in real world and to prevent it in our model, we will assign a negative penalty whenever the taxi hits a wall and this will encourage the taxi to take a different action.

Let's look at how we can use **OpenAI Gym** to use the taxi environment that is already present there.

```
In [17]: import sys
         sys.path.append("c:/users/iftek/appdata/local/conda/conda/envs/gym/lib/site-packages")

         import gym


         env = gym.make("Taxi-v3").env

         state = env.encode(3, 1, 2, 0) # (taxi row, taxi column, passenger index, destination index)
         print("State:", state)

         env.s = state
         #env.reset() # reset environment to a new, random state
         env.render()


         print("Action Space {}".format(env.action_space))
         print("State Space {}".format(env.observation_space))

State: 328
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| |: | : |
|Y| : |B: |
+---------+

Action Space Discrete(6)
State Space Discrete(500)
```

We load the environment using "gym.make("Taxi-v2").env". Our taxi is in state 328, out of 500 states. This code will create the same environment that we looked in our illustration above because we are encoding its state using env.encode(3,1,2,0). The code prints action

space as size of 6 and State space as 500 which aligns with our illustration. We have the walls (“|”) and all the possible pickup and drop off destinations (R, G, Y, B). Letter in blue is the current location and in purple the destination.

The environment creates a default reward table “P” which can be thought as a matrix containing the different states in rows and actions in the columns.

```
In [18]: env.P[328]

Out[18]: {0: [(1.0, 428, -1, False)],
          1: [(1.0, 228, -1, False)],
          2: [(1.0, 348, -1, False)],
          3: [(1.0, 328, -1, False)],
          4: [(1.0, 328, -10, False)],
          5: [(1.0, 328, -10, False)]}
```

env.P[328] prints out the 6 states and the reward values assigned to them. The information is in a dictionary structure- {action: [(probability, nextstate, reward, done)]}. Probability of falling into the states is same for all. The nextstate is the state we would be in if we take the action at this index of the dictionary. The movements have -1 reward whereas, the pickup and drop-off have -10. Done will be true when we successfully drop off a passenger.

Instead of using Reinforcement learning, we can select a random action from all possible actions and see how our agent behaves.

```
In [21]: env.s = 328  # set environment to illustration's state

         epochs = 0
         penalties, reward = 0, 0

         frames = [] # for animation

         done = False

         while not done:
             action = env.action_space.sample()
             state, reward, done, info = env.step(action)

             if reward == -10:
                 penalties += 1

             # Put each rendered frame into dict for animation
             frames.append({
                 'frame': env.render(mode='ansi'),
                 'state': state,
                 'action': action,
                 'reward': reward
                 }
             )

             epochs += 1


         print("Timesteps taken: {}".format(epochs))
         print("Penalties incurred: {}".format(penalties))

         Timesteps taken: 5327
         Penalties incurred: 1771
```

This code prints out the time taken or **timesteps** taken and penalties incurred throughout an entire **episode** (journey of a single passenger). env.action_space.sample() is the random action selector. Timesteps taken is huge as well as the penalties incurred making it an inefficient solution.

We need to optimise the naive algorithm and use a RL algorithm- Q-learning, which will give the agent the ability to keep the best action for each state in its memory. For this, the Taxi environment will have a **Q-table** to store the **Q-values**. Q-values, and they map to a (state, action) combination. Q-values depict the quality of action taken from a state. Better Q-values imply better chances of getting rewards.

Initially, the Q-values are zero or any random value. As the taxi will travel and gain rewards, the Q-values will change. They change according to this equation-

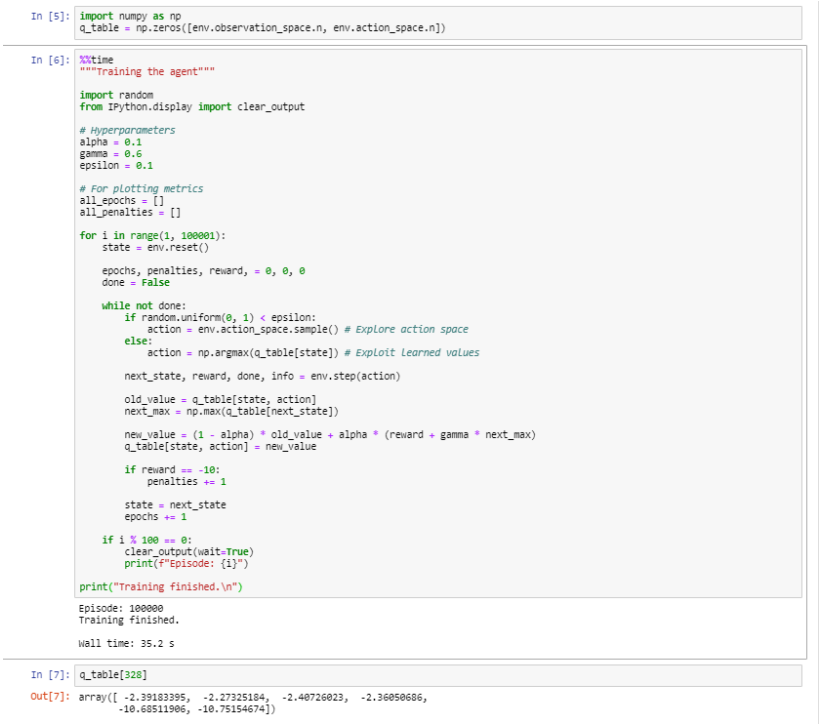
$$Q(\text{state},\text{action}) \leftarrow (1-\alpha)Q(\text{state},\text{action}) + \alpha(\text{reward} + \gamma \max Q(\text{next state},\text{all actions}))$$

- α is the rate at which our Q-values are changing ($0<\alpha<1$).
- γ is the **discount factor.** $0<\gamma<1$ because we have a γ value that exponentially decreases. This is because we would prefer to get the rewards over a long period of time. For example- if reward is in sequence (3,2,1)- 1*3+0.5*2+0.25*1 instead of (1,2,3)- 1*1 + 0.5*2+ 0.25*3, first sequence will give us the highest cumulative reward.
- $Q(\text{state}, \text{action}) \leftarrow (1-\alpha)Q(\text{state}, \text{action})$ is telling us that we are updating the Q function to the left by adding the old Q function value with the rest of the equation- $\alpha(\text{reward} + \gamma \max Q(\text{next state}, \text{all actions}))$. This is equivalent to A = A + B in maths. We do $1-\alpha$ which is the weight of the old Q-value.
- The rest of the equation is $\alpha(\text{reward} + \gamma \max Q(\text{next state}, \text{all actions}))$ is the learned value. “reward” is earned transitioning from the present state to the next state, and $\gamma \max Q(\text{next state}, \text{all actions})$ is the value of the action that is estimated to return the maximum total future reward based on all the possible actions that can be made in the next state.

In short, the taxi is learning to take proper action at the current state by looking at the reward at the current state and maximum reward we can earn in the next states.

Training our taxi Agent

We need to have a training algorithm to fill up our Q-table.



```
In [5]: import numpy as np
        q_table = np.zeros([env.observation_space.n, env.action_space.n])

In [6]: %%time
        """Training the agent"""

        import random
        from IPython.display import clear_output

        # Hyperparameters
        alpha = 0.1
        gamma = 0.6
        epsilon = 0.1

        # For plotting metrics
        all_epochs = []
        all_penalties = []

        for i in range(1, 100001):
            state = env.reset()

            epochs, penalties, reward, = 0, 0, 0
            done = False

            while not done:
                if random.uniform(0, 1) < epsilon:
                    action = env.action_space.sample() # Explore action space
                else:
                    action = np.argmax(q_table[state]) # Exploit learned values

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

                old_value = q_table[state, action]
                next_max = np.max(q_table[next_state])

                new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
                q_table[state, action] = new_value

                if reward == -10:
                    penalties += 1

                state = next_state
                epochs += 1

            if i % 100 == 0:
                clear_output(wait=True)
                print(f"Episode: {i}")

        print("Training finished.\n")

        Episode: 100000
        Training finished.

        Wall time: 35.2 s

In [7]: q_table[328]

Out[7]: array([ -2.39183395,  -2.27325184,  -2.40726023,  -2.36050686,
               -10.68511906, -10.75154674])
```

- In[5] we initialise a 500 x 6 matrix containing zeros only.

- In[6] is our actual algorithm. We declare our hyperparameters alpha, gamma and an additional one called epsilon. We use epsilon to prevent taking the best Q-value all the time and prefer exploring the action space further.
- We are training over 100,000 episodes.
- Rest of the code is implementing the equation that we discussed above.
- Output of q_table[328] shows us the best actions to take at the state of 328 which is our illustration state. -2.27325184 is the max Q-value which is north and our agent was successful in figuring that out.

We can now test our agent's performance by comparing results without and with Q-learning.

In [9]:
```
"""Evaluate agent's performance without Q-learning"""

total_epochs, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = False

    while not done:
        action = env.action_space.sample()
        state, reward, done, info = env.step(action)

        if reward == -10:
            penalties += 1

        epochs += 1

    total_penalties += penalties
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

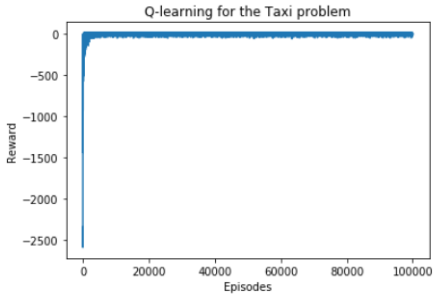
```
Results after 100 episodes:
Average timesteps per episode: 2478.97
Average penalties per episode: 807.21
```

In [8]:
```
"""Evaluate agent's performance after Q-learning"""

total_epochs, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = False

    while not done:
        action = np.argmax(q_table[state])
        state, reward, done, info = env.step(action)

        if reward == -10:
            penalties += 1

        epochs += 1

    total_penalties += penalties
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

```
Results after 100 episodes:
Average timesteps per episode: 12.8
Average penalties per episode: 0.0
```

Average timesteps decreased drastically from 2478.97 to 12.8 and 0 penalties incurred. Use of Q-table has been effective and training was successful. Optimal policy is way better than the random policy.

In [163]:
```
plt.plot(all_rewards)
plt.xlabel("Episodes")
plt.ylabel("Reward")

plt.title("Q-learning for the Taxi problem")
plt.show()
```



Graph shows how Q-learning changed the negative rewards into positive ones over the period of 100,000 episodes.

Cartpole problem

Cartpole can be thought as an inverted pendulum. The goal of the cartpole problem is to balance the inverted pendulum on a cart (which is on a frictionless track) and keep the pole upright for as long as possible. Equal forces, such as +1 and -1 can be applied to the cart from left and right to keep the pendulum upright.

Similarly to the taxi problem, the cartpole problem will have an action and reward mechanism. A positive reward will be provided for every time step the pole remains upright. An episode ends when the pole falls below a threshold height of 15 degrees from the upright position or the cart moves more than 2.4 units from the center.

```
In [12]: env = gym.make("CartPole-v0")
         env.reset()
Out[12]: array([-0.04517563, -0.00695125,  0.0202226 , -0.03206729])
```

```
In [17]: print("Action Space {}".format(env.action_space.n))
         print("State Space {}".format(env.observation_space.shape[0]))

         Action Space 2
         State Space 4
```

- The state spaces contain the 4 possible observations from the cartpole- cart position, cart velocity, pole angle and pole velocity at the tip.
- The action has only two actions- push cart to the right and push cart to the left.

There is a simple method using weight vector to solve this problem. We take a random vector of size 4 which is equal to the state space dimension. A dot product is taken between the weight vector and the states.

```
In [19]: parameters = np.random.rand(4) * 2 - 1
```

```
In [20]: def run_episode(env, parameters):
             observation = env.reset()
             totalreward = 0
             for _ in range(200):
                 env.render()
                 action = 0 if np.matmul(parameters,observation) < 0 else 1
                 observation, reward, done, info = env.step(action)
                 totalreward += reward
                 if done:
                     break
             return totalreward
```

- In In[19] the weights are being generated randomly between [-1,1].
- We create a function in In[20] to run one episode and see how much reward we can get.
- "action = 0 if np.matmul(parameters,observation) < 0 else 1" here we multiply our weight by the observations as discussed before and if we get less than 0 the cart moves left otherwise right.
- We output the total reward we get from assigning the weights.

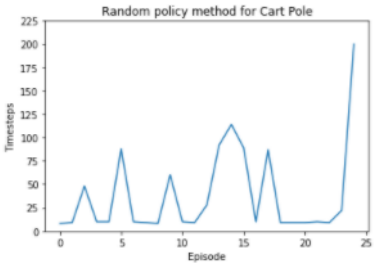
Now we can test this function out by a random search method:

In [15]:
```
%%time

results = []
timeSteps = []
bestparams = None
bestreward = 0
for _ in range(1000):
    parameters = np.random.rand(4) * 2 - 1
    reward = run_episode(env,parameters)
    if reward > bestreward:
        bestreward = reward
        bestparams = parameters
        # considered solved if the agent lasts 200 timesteps
        if reward == 200:
            timeSteps.append(200)
            break
    timeSteps.append(reward)
env.close()
```

```
Wall time: 16.3 s
```

In [16]:
```
plt.plot(timeSteps)
plt.title("Random policy method for Cart Pole")
plt.xlabel("Episode")
plt.ylim(0, 225)
plt.ylabel("Timesteps")
```

Out[16]:
```
Text(0, 0.5, 'Timesteps')
```



In this part, I am using the random search method which is trying random weights and picks the one that performs the best. For 1000 iterations, we see if we can reach our goal of 200 timesteps or 200 rewards. The graph shows that we were able to reach 200 timesteps but in a span of 25 episodes or after failing for almost 25 times. This is unpredictable because at times it would take as less as 3 episodes to reach 200 timesteps.

We can use Q-learning instead just like we did with the taxi problem. But there is a difference between the two solutions. In taxi problem, our states where discrete but here the states (velocity, position, etc.) are continuous variables. To be able to solve this problem we first discretize these states. Discretizing here means we would group several values of each of the variables into a 'discrete bucket' and treat them as similar states. If any time, the state does not end up in these buckets, a failure is considered which is the end of an episode.

In [9]:
```
#Convert Cartpoles continues state space into discrete one
from typing import Tuple

n_bins = ( 6 , 12 )
lower_bounds = [ env.observation_space.low[2], -math.radians(50) ]
upper_bounds = [ env.observation_space.high[2], math.radians(50) ]

def discretizer( _ , __ , angle, pole_velocity ) -> Tuple[int,...]:
    """Convert continues state intro a discrete state"""
    est = KBinsDiscretizer(n_bins=n_bins, encode='ordinal', strategy='uniform')
    est.fit([lower_bounds, upper_bounds ])
    return tuple(map(int,est.transform([[angle, pole_velocity]])[0]))
```

In [10]:
```
#Initialise the Q table
Q_table = np.zeros(n_bins + (env.action_space.n,))
Q_table.shape
```

Out[10]:
```
(6, 12, 2)
```

For discretizing, I am using KBinsDiscretizer from sklearn. We are ignoring cart position and cart velocity which is why first arguments are blank in the discretizer function. For these two variables, we are dividing them into 6 and 12 buckets/bins respectively. Then, we initialise our Q-table with the buckets and action space.

In [11]:
```
def policy( state : tuple ):
    """Choosing action based on epsilon-greedy policy"""
    return np.argmax(Q_table[state])
```

In [12]:
```
def new_Q_value( reward : float ,  new_state : tuple , discount_factor=1 ) -> float:
    """Temperal diffrence for updating Q-value of state-action pair"""
    future_optimal_value = np.max(Q_table[new_state])
    learned_value = reward + discount_factor * future_optimal_value
    return learned_value
```

In [13]:
```
# Adaptive Learning of Learning Rate
def learning_rate(n : int , min_rate=0.01 ) -> float  :
    """Decaying learning rate"""
    return max(min_rate, min(1.0, 1.0 - math.log10((n + 1) / 25)))
```

In [14]:
```
def exploration_rate(n : int, min_rate= 0.1 ) -> float :
    """Decaying exploration rate"""
    return max(min_rate, min(1, 1.0 - math.log10((n  + 1) / 25)))
```

These are the functions that will be implementing the same mathematical equation that we saw for Q-learning of the taxi model. Policy function is a new thing here which gives a policy for picking the highest value in the Q-table.

$$Q(\text{state,action}) \leftarrow (1-\alpha)Q(\text{state,action}) + \underline{\alpha(\text{reward} + \gamma \max Q(\text{next state,all actions}))}$$

new_Q_value function returns this part of the equation. Discount factor is our γ.

“learning_rate” function is our function to calculate α because we do not want to learn too much from one individual action. At the start of the simulation, the learning rate will be high but it will decay in time slowing towards the end.

Similarly, we explore the state space as much as we can at the first but the rate of exploration should decay with time which is done using the **exploration_rate** function. This is the same as the epsilon hyperparameter of the taxi problem.

Training our cartpole agent

```
In [15]: #Training our agent
n_episodes = 1000
for e in range(n_episodes):

    # Siscretize state into buckets
    current_state, done = discretizer(*env.reset()), False

    while done==False:

        # policy action
        action = policy(current_state) # exploit

        # insert random action
        if np.random.random() < exploration_rate(e) :
            action = env.action_space.sample() # explore

        # increment enviroment
        obs, reward, done, _ = env.step(action)
        new_state = discretizer(*obs)

        # Update Q-Table
        lr = learning_rate(e)
        learnt_value = new_Q_value(reward , new_state )
        old_value = Q_table[current_state][action]
        Q_table[current_state][action] = (1-lr)*old_value + lr*learnt_value

        current_state = new_state
 # print the episode number every 100 episodes for the user to keep track of progress of training
    if e % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {e}")

print("Training finished.\n")
```

```
Episode: 900
Training finished.
```

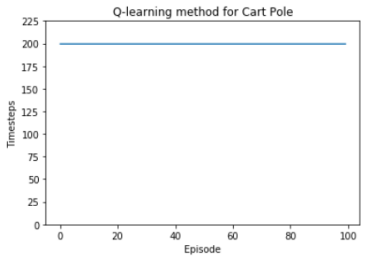
```
In [19]: """Evaluate agent's performance after Q-learning"""
total_epochs = 0
episodes = 100
timesteps = []

for i in range(episodes):
    # discretise the state
    state = discretizer(*env.reset())
    epochs, reward = 0, 0
    done = False

    while not done:
        action = policy(state)
        state, reward, done, info = env.step(action)
        state = discretizer(*state)

        epochs += 1

    total_epochs += epochs
    timesteps.append(epochs)

env.close()

clear_output(wait=True)

# print results to screen
print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
```

```
Results after 100 episodes:
Average timesteps per episode: 200.0
```

Finally, we complete our training for 900 episodes. Evaluating agent's performance shows that results are perfect with average timesteps per episode 200. Agent was able to receive maximum reward each episode. This is better than the random weighting method which was giving us unpredictable results. Q-learning proved to be effective yet again.

```
In [94]: plt.plot(timesteps)
plt.title("Q-learning method for Cart Pole")
plt.xlabel("Episode")
plt.ylim(0, 225)
plt.ylabel("Timesteps")
```

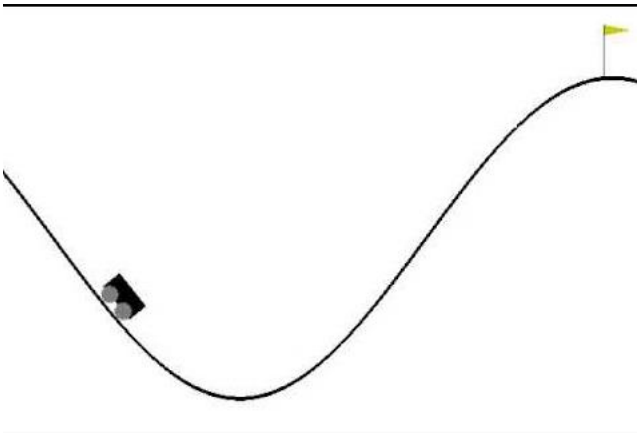
```
Out[94]: Text(0, 0.5, 'Timesteps')
```



In conclusion, we have seen that for taxi-problem, we were able to completely eliminate penalties using Q-learning. Similarly, we were able to get maximum rewards for each episode in our cartpole problem. It is possible to achieve perfection through training our agent and implementing an effective Q-learning algorithm.

Mountain car problem

Another common problem in RL is the study of the mountain car problem. The objective of solving this problem is to create an algorithm that can ride up a steep mountain slope and reach its goal (the goal is marked by a flag). The slope is extremely steep. Hence, the car needs an initial momentum to climb up the hill. This momentum is gained depended on the position of the car from the left hill.



There can be two states of the car for this scenario-

- Position of the car
- Velocity of the car

Both of which are continuous variables which means we need to discretize the same way we did for the cartpole problem.

There are three possible actions that the car can take-

- Move left
- Do nothing
- Move right

The Q-learning algorithm will use the same action and reward mechanism we have seen before. For each timestep that the car takes to reach its destination located at position 0.5, the reward will be -1. The goal is to reach in less than 200 timesteps. Reaching 200 timesteps results in 1 episode or reaching our goal completes 1 episode. The mathematical model for this problem is-

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$$

Q-Learning Update Equation

Training the mountain car agent

In [34]:
```
# discretizer method
def discretizer(state):
    result = (state - env.observation_space.low) * np.array([10, 100])
    result = np.round(result, 0).astype(int)

    return result

# method to determine the shape of the discretized state space to create the Q-table with
def getDiscreteStateSize(env):
    result = (env.observation_space.high - env.observation_space.low) * np.array([10, 100])
    result = np.round(result, 0).astype(int) + 1

    return result

# Hyperparameters
alpha = 0.2
gamma = 0.9
epsilon = 0.8

# global variables
episodes = 10000
last_ten_episodes = episodes - 10

# get shape of discretized state space
num_states = getDiscreteStateSize(env)

# Initialize Q table
Q = np.zeros((num_states[0], num_states[1], env.action_space.n))

# Initialize variables to track rewards
rewards = []
average_rewards = []

# Calculate reduction in the epsilon value per episode
reduction = epsilon / episodes
```

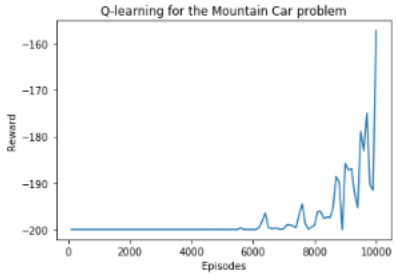
This part of the code we discretize and using that discretized state space, we create our corresponding Q-table. Then we create our hyperparameters same as before, announce the number of episodes over which we want to train our algorithm.

In [35]:
```
%%time

# Run Q learning algorithm
for i in range(episodes):
    # Initialize local variables
    done = False
    total_reward, reward = 0,0
    state = env.reset()

    # Discretize current state
    discretized_state = discretizer(state)

    while done != True:

        # Determine next action based of an epsilon greedy strategy
        if np.random.random() < 1 - epsilon:
            # exploit known states
            action = np.argmax(Q[discretized_state[0], discretized_state[1]])
        else:
            #the state space
            action = np.random.randint(0, env.action_space.n)

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

        next_discretized_state = discretizer(next_state)

        # update the state corresponding to the end point for the car
        if done and next_state[0] >= 0.5:
            Q[discretized_state[0], discretized_state[1], action] = reward

        # Adjust Q value for current state
        else:
            q_value = alpha*(reward +
                             gamma*np.max(Q[next_discretized_state[0],
                                            next_discretized_state[1]]) -
                             Q[discretized_state[0], discretized_state[1], action])

            Q[discretized_state[0], discretized_state[1], action] += q_value

        # Update total reward to include results from this episode and move the current state to the next state
        total_reward += reward
        discretized_state = next_discretized_state

    # decrease epsilon value with every episode untill it reaches 0
    if epsilon > 0:
        epsilon -= reduction

    # add the running total to the rewards array
    rewards.append(total_reward)

    # on every 100th episode
    if (i+1) % 100 == 0:
        # get the average of the last 100 episode and store this average in another list
        last_average_reward = np.mean(rewards)
        average_rewards.append(last_average_reward)

        # reset the rewards list for the next 100 episodes
        rewards = []

        # clear screen and output average to user
        clear_output(wait=True)
        print(f"Episode {i + 1} out of {episodes} episodes")
        print(f"Average for episode: {last_average_reward}")

env.close()
```

```
Episode 10000 out of 10000 episodes
Average for episode: -157.2
Wall time: 53.5 s
```

After training our algorithm using Q-learning, we can see that the average reward is -157.2. The less negative the reward, the least time it took for the car to reach its goal. -157.2 is not really a satisfactory value. Plotting a graph of average rewards against the 10000 episodes would show improvement but this is not drastic:

In [36]:
```
# Plot Rewards to show results

# plot each average value 100 data points away from each other
plt.plot(100*(np.arange(len(average_rewards)) + 1), average_rewards)
plt.xlabel("Episodes")
plt.ylabel("Reward")
plt.title("Q-learning for the Mountain Car problem")
plt.show()
```



In [ ]:

It is a tricky environment for an agent to perform compared to the other ones we discussed before even after implementing a Q-learning algorithm. Here, we are sort of getting punished (getting negative reward) for not being able to reach our destination early. Tuning of hyperparameters correctly such as our discount factor becomes extremely important on this occasion.