COSC 2753 | Machine Learning

Week 8+1 Lab Exercises: **Reinforcement learning**

```
!pip install gym[box2d]==0.17.* pyvirtualdisplay==0.2.* PyOpenGL==3.1.* PyOpenGL-accelerate==3.1.*
```

Introduction

In this lab you will be:

1. Learning how to use OpenAI gym.
2. Implement Q-learning to solve a well-known toy reinforcement learning problem called MountainCar problem or `Cartpole` problem (https://gym.openai.com/envs/CartPole-v1/).

** Please run this lab on the anaconda environment on your PC.**

Mountain Car with RL

Mountain Car is a classic control Reinforcement Learning problem that was first introduced by A. Moore in 1991 [1]. A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. It can be tricky to find this optimal solution due to the sparsity of the reward.

Mountain Car Problem definition:

> Objective: Get the car to the top of the right hand side mountain.

> State: Car's horizontal position and velocity (can be negative).

> Action: Direction of push (left, nothing or right).

> Reward: -1 for every time step until success, which incentivises quick solutions.

More information about Mountain Car can be found: https://en.wikipedia.org/wiki/Mountain_car_problem or https://www.toptal.com/machine-learning/deep-dive-into-reinforcement-learning

OpenAI Gym

OpenAI Gym is a Python package comprising a selection of RL environments, ranging from simple "toy" environments to more challenging environments, including simulated robotics environments and Atari video game environments. It was developed with the aim of becoming a standardized environment and benchmark for RL research. In this Lab, we will use the OpenAI Gym Cartpole environment to demonstrate how to get started in using this exciting tool and show how Q-learning can be used to solve this problem.

Setting up the environment

Lets first import the libraries required for the implementation.

```python
import matplotlib.pyplot as plt
%matplotlib inline
from IPython import display
!pip install gym
import numpy as np
import gym
```

```
Requirement already satisfied: gym in /Users/thienbao/opt/anaconda3/lib/python3.9/site-packages (0.17.3)
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /Users/thienbao/opt/anaconda3/lib/python3.9/site-packages (from gym) (1.5.0)
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /Users/thienbao/opt/anaconda3/lib/python3.9/site-packages (from gym) (1.6.0)
Requirement already satisfied: scipy in /Users/thienbao/opt/anaconda3/lib/python3.9/site-packages (from gym) (1.7.1)
Requirement already satisfied: numpy>=1.10.4 in /Users/thienbao/opt/anaconda3/lib/python3.9/site-packages (from gym) (1.20.3)
Requirement already satisfied: future in /Users/thienbao/opt/anaconda3/lib/python3.9/site-packages (from pyglet<=1.5.0,>=1.4.0->gym) (0.18.2)
```

To start using the mountain car environment initialize it as follows:

```python
import gym
print(gym.__version__)# for me: 0.15.4
env = gym.make("MountainCar-v0")
obs = env.reset()
for i in range(1000):# it's changable
    env.step(env.action_space.sample())
    env.render()# won't work in Google Colab
env.close()
```

```
0.17.3
```

The `env.reset()` command resets the environemnt and return the initial state

Lets explore the state space and the action space og the Cartpole environment

```python
print(obs)
```

```
[-0.4160749  0.       ]
```

```python
print('State space: ', env.observation_space)
print('Action space: ', env.action_space)
```

```
State space:  Box(-1.2000000476837158, 0.6000000238418579, (2,), float32)
Action space:  Discrete(3)
```

1. `print('State space: ', env.observation_space)` : This line is printing the state space of the environment. The state space, represented by `env.observation_space`, is the set of all possible states that the agent can be in. In a game, for example, this could include the positions of all the objects on the screen. The structure of the state space depends on the specific environment. It could be a multi-dimensional array for complex environments or a simple numeric value for simpler ones.
2. `print('Action space: ', env.action_space)` : This line is printing the action space of the environment. The action space, represented by `env.action_space`, is the set of all possible actions that the agent can take. In a game, this could include moving left, moving right, jumping, etc. Similar to the state space, the structure of the action space depends on the specific environment. It could be a simple discrete value for environments with a fixed number of actions or a continuous value for environments with a range of possible actions.

The `print` function is a built-in Python function that writes the specified message to the screen, or other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen. In this case, it's being used to display the state and action spaces of the environment.

This tells us that the state space is a 2-dimensional space, so each state observation is a vector of 2 (float) values, and that the action space comprises three discrete actions (left, nothing or right). By default, the three actions are represented by the integers 0, 1 and 2. How about the state space? What are the limits of the state space?

```python
print('State space Low: ', env.observation_space.low)
print('State space High: ', env.observation_space.high)
```

```
State space Low:  [-1.2  -0.07]
State space High:  [0.6  0.07]
```

State: Two-dimensional continuous state space.

> Velocity=(-0.07,0.07)

> Position=(-1.2,0.6)

Actions: One-dimensional discrete action space.

> action=(left,neutral,right)

Reward: For every time step:

> reward=-1

Update function: For every time step:

> Action=[-1,0,1]

> Velocity=Velocity+(Action)*0.001+cos((3*Position)*(-0.0025))

> Position=Position+Velocity

Starting condition: Optionally, many implementations include randomness in both parameters to show better generalized learning.

> Position=-0.5

> Velocity=0.0

Termination condition: End the simulation when:

> Position >= 0.6

This shows that the first state variable (horizontal position) has a range [-1.2, 0.6] and the second state variable(speed) has a range [-0.07, 0.07]. The state space of the environment is a continuous state space, which means that there are infinitely many state-action pairs, making it impossible to build a Q table. As a solution to this problem we can discretize the state space. One simple discretization is to cover the state space to a grid with spacing of 0.1 along first element and 0.01 along second element in the state space. The states can be given integer indexes multiply the first element by 10 and the second by 100. lets see the size of discretized state space.

```
num_states = (env.observation_space.high - env.observation_space.low)*np.array([10, 100])
num_states = np.round(num_states, 0).astype(int) + 1
print(num_states)
```

```
[19 15]
```

We can also write a function that will convert a continuous state vector to a discrete one.

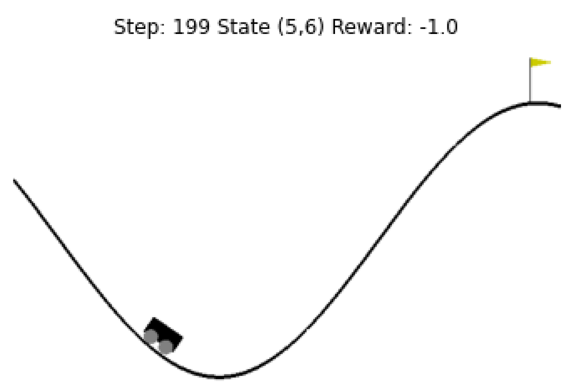
```
# Discretize state
def discretize_state(state, env_low):
  state_adj = (state - env_low)*np.array([10, 100])
  state_adj = np.round(state_adj, 0).astype(int)
  return state_adj
```

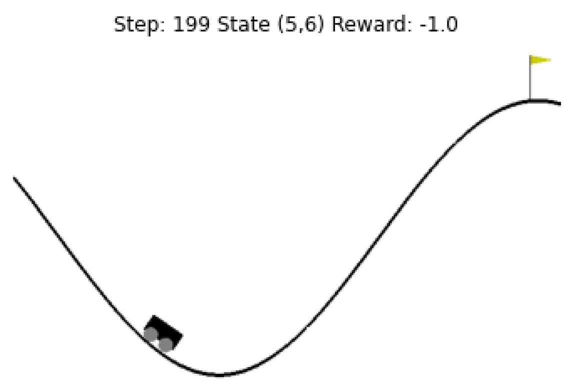
Let's now make some random actions in the environment and see what the output will be. For this we need a function to plot the output of the environment.

```
def show_state(env, step=0, info=""):
  plt.figure(1)
  plt.clf()
  plt.imshow(env.render(mode='rgb_array'))
  plt.title("Step: %d %s" % (step, info))
  plt.axis('off')
  display.clear_output(wait=True)
  display.display(plt.gcf())
```

Now we take some random actions:

```
env.reset()
done = False
step_index = 0
while done != True:
  action = env.action_space.sample() # get a random action from the set of actions
  state, reward, done, info = env.step(action) # perform the action and receive new state and reward
  d_state = discretize_state(state, env.observation_space.low)
  show_state(env, step=step_index, info='State ({},{}) Reward: {}'.format(d_state[0], d_state[1], reward))
  step_index = step_index + 1
```





The provided Python code is a simple loop that runs a reinforcement learning episode in an environment represented by the `env` object. Here's a step-by-step explanation of what the code does:

1. `env.reset()` : This line resets the environment to its initial state. This is typically done at the beginning of each episode to ensure that the environment is in a known state.
2. `done = False` : This line initializes the `done` variable to `False` . This variable is used to indicate whether the episode has ended. The loop continues to run as long as `done` is `False` .
3. `while done != True:` : This line starts a while loop that continues to run until `done` is `True` , which indicates that the episode has ended.
4. `action = env.action_space.sample()` : This line selects a random action from the action space of the environment. The `env.action_space.sample()` function returns a random action.
5. `state, reward, done, info = env.step(action)` : This line performs the selected action in the environment, which results in a new state and a reward. The `env.step(action)` function takes an action as input and returns the new state, the reward for performing the action, a boolean indicating whether the episode has ended (`done`), and an info dictionary containing extra information.
6. `d_state = discretize_state(state, env.observation_space.low)` : This line discretizes the continuous state space into a discrete state space. This is typically done when you want to apply a discrete-space algorithm, like Q-learning, to a continuous-space problem.
7. `show_state(env, step=step_index, info='State ({},{}) Reward: {}'.format(d_state[0], d_state[1], reward))` : This line visualizes the current state of the environment. The `show_state` function is not defined in the provided code, but it likely takes the environment, the current step index, and some info text as input and displays the state of the environment in some way.
8. `step_index = step_index + 1` : This line increments the step index by one. The step index is used to keep track of the number of steps that have been taken in the current episode.

In summary, this code runs a single episode of a reinforcement learning experiment in a given environment. It selects actions randomly, performs them in the environment, and visualizes the resulting states.

Run the following block if the visualization of the environment gives an error. On mac you need to install pyglet version 1.5.11 to get the gym environment to render. The installation will give an error, but it will work.

```
#!pip install pyglet==1.5.11
```

Now let's develop a function for Q learning. The function prototype is given below and the algorithm for Q learning is given in Algorithm 1. Assume that Q is a numpy matrix with dimensions (number of elements for state 1, number of elements for state 2, number of actions).

```
# Define Q-learning function
def QLearning(env, Q, learning, discount, epsilon, episodes):
    # Env: The OpenAI gym environment
    # Q: Initial Q table
    # learning: Learning Rate of Q learing
    # discount: discount factor (gamma)
    # epsilon: epsilon for exploration vs exploitation
    # episodes: number of episodes to run when learing the Q table

    # Initialize variables to hold rewards
    reward_list = []

    # Calculate reduction in epsilon per episode
    epsilon_d = (epsilon)/episodes

    for i in range(episodes):
```

```
    for i in range(episodes):
        done = False
        tot_reward, reward = 0,0
        state = env.reset()

        state_adj = discretize_state(state, env.observation_space.low)

        while done != True:

            # Determine next action - epsilon greedy strategy for explore vs exploitation
            if np.random.random() < 1 - epsilon:
                # select the best action according to Qtable (exploitation)
                # TODO
            else:
                # select a random action (exploration)
                # TODO

            # Step and Get the next state and reward
            # TODO

            # Allow for terminal states
            if done and state2[0] >= 0.5:
                Q[state_adj[0], state_adj[1], action] = reward

            # Update the Q table
            else:
                # TODO

            # Update variables
            tot_reward += reward
            state_adj = state2_adj

        # Update epsilon
        if epsilon > 0:
            epsilon -= epsilon_d

        # Track rewards
        reward_list.append(tot_reward)

        if (i+1) % 100 == 0:
            ave_reward = np.mean(reward_list)
            reward_list = []

        # Average reward is the average number of steps that the agent spent to win
        if (i+1) % 100 == 0:
            print('Episode {} Average Reward: {} Epsilon {}'.format(i+1, ave_reward, np.round(ep

    env.close()

    return Q
```

```
# Define Q-learning function
def QLearning(env, Q, learning, discount, epsilon, episodes):
    # Env: The OpenAI gym environment
    # Q: Initial Q table
    # learning: Learning Rate of Q learing
    # discount: discount factor (gamma)
    # epsilon: epsilon for exploration vs exploitation
    # episodes: number of episodes to run when learing the Q table

    # Initialize variables to hold rewards
    reward_list = []
    ave_reward_list = []

    # Calculate reduction in epsilon per episode
    epsilon_d = (epsilon)/episodes

    for i in range(episodes):
        done = False
        tot_reward, reward = 0,0
        state = env.reset()

        state_adj = discretize_state(state, env.observation_space.low)

        while done != True:

            # Determine next action - epsilon greedy strategy for explore vs exploitation
            if np.random.random() < 1 - epsilon:
                # select the best action according to Qtable (exploitation)
                # TODO
                action = np.argmax(Q[state_adj[0], state_adj[1]])
            else:
                # select a random action (exploration)
                # TODO
                action = np.random.randint(0, env.action_space.n)

            # Step and Get the next state and reward
            # TODO
            state2, reward, done, info = env.step(action)

            # Discretize state2
            state2_adj = (state2 - env.observation_space.low)*np.array([10, 100])
            state2_adj = np.round(state2_adj, 0).astype(int)

            # Allow for terminal states
            if done and state2[0] >= 0.5:
                Q[state_adj[0], state_adj[1], action] = reward

            # Update the Q table
            else:
                # TODO
                delta = learning*(reward +
                                 discount*np.max(Q[state2_adj[0],
                                                   state2_adj[1]]) -
                                 Q[state_adj[0], state_adj[1],action])
                Q[state_adj[0], state_adj[1],action] += delta

            # Update variables
            tot_reward += reward
            state_adj = state2_adj

        # Update epsilon
        if epsilon > 0:
            epsilon -= epsilon_d

        # Track rewards
        reward_list.append(tot_reward)

        if (i+1) % 100 == 0:
            ave_reward = np.mean(reward_list)
            ave_reward_list.append(ave_reward)
            reward_list = []

        # Average reward is the average number of steps that the agent spent to win
        if (i+1) % 100 == 0:
            print('Episode {} Average Reward: {} Epsilon {}'.format(i+1, ave_reward, np.round(epsilon,2)))

    env.close()

    return Q, ave_reward_list
```

The provided Python code defines a function `QLearning` that implements the Q-learning algorithm, a type of reinforcement learning algorithm. The function takes as input an OpenAI Gym environment (`env`), an initial Q-table (`Q`), a learning rate (`learning`), a discount factor (`discount`), an epsilon value for the epsilon-greedy policy (`epsilon`), and the number of episodes to run (`episodes`).

The function begins by initializing two lists to hold the rewards for each episode (`reward_list`) and the average rewards over every 100 episodes (`ave_reward_list`). It also calculates the amount by which epsilon should be decreased after each episode (`epsilon_d`) to gradually shift from exploration to exploitation.

The function then enters a loop that runs for the specified number of episodes. For each episode, it initializes some variables, resets the environment to its initial state, and discretizes this state. It then enters another loop that continues until the episode ends.

In this inner loop, the function first decides whether to take the best action according to the current Q-table (exploitation) or a random action (exploration), based on a random number and the current epsilon value. It then performs the chosen action in the environment using the `env.step(action)` function, which returns the new state, the reward for the action, a boolean indicating whether the episode has ended, and an info dictionary.

The function then discretizes the new state and updates the Q-table. If the episode has ended and the agent has reached the goal state, it sets the Q-value for the current state-action pair to the reward. Otherwise, it calculates the difference between the current Q-value and the expected Q-value based on the reward and the maximum Q-value for the new state, scales this difference by the learning rate, and adds the result to the current Q-value.

After updating the Q-table, the function updates the total reward for the episode and the current state, and exits the inner loop. It then decreases epsilon by `epsilon_d`, adds the total reward for the episode to `reward_list`, and calculates and prints the average reward every 100 episodes.

Finally, after all episodes have been run, the function closes the environment and returns the final Q-table and the list of average rewards.

Sample Solutions

If you are struggling with the above function, a sample solution has been provided. Only use this if you have made your absolute best attempts at implementing the function yourself. The purpose of this lab is to understand common aspects of RL algorithm, though the Q-learning algorithm. You will gain significantly less out of this lab if you don't try to solve the problems yourself.

Learning & testing the model

Now we have all the elements required. Let's learn the model.

```
# Initialize Q table randomly
Q = np.random.uniform(low = -1, high = 1, size = (num_states[0], num_states[1], env.action_space.n))
# Run Q-learning algorithm
Q, rewards = QLearning(env, Q, 0.2, 0.9, 0.8, 2000)
```

```
Episode 100 Average Reward: -200.0 Epsilon 0.76
Episode 200 Average Reward: -200.0 Epsilon 0.72
Episode 300 Average Reward: -200.0 Epsilon 0.68
Episode 400 Average Reward: -200.0 Epsilon 0.64
Episode 500 Average Reward: -200.0 Epsilon 0.6
Episode 600 Average Reward: -200.0 Epsilon 0.56
Episode 700 Average Reward: -200.0 Epsilon 0.52
Episode 800 Average Reward: -200.0 Epsilon 0.48
Episode 900 Average Reward: -200.0 Epsilon 0.44
Episode 1000 Average Reward: -200.0 Epsilon 0.4
Episode 1100 Average Reward: -200.0 Epsilon 0.36
Episode 1200 Average Reward: -199.89 Epsilon 0.32
Episode 1300 Average Reward: -200.0 Epsilon 0.28
Episode 1400 Average Reward: -199.47 Epsilon 0.24
Episode 1500 Average Reward: -199.75 Epsilon 0.2
Episode 1600 Average Reward: -199.14 Epsilon 0.16
Episode 1700 Average Reward: -197.57 Epsilon 0.12
Episode 1800 Average Reward: -200.0 Epsilon 0.08
Episode 1900 Average Reward: -199.81 Epsilon 0.04
Episode 2000 Average Reward: -179.46 Epsilon 0.0
```

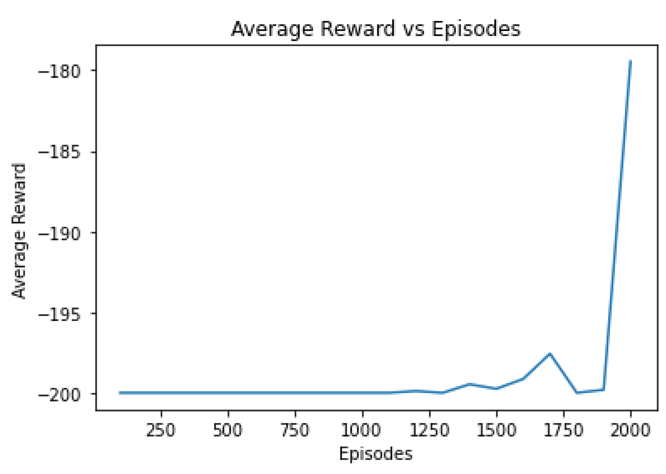
Now let's see how we can perform the task with the learned model

```
state = env.reset()
state_adj = discretize_state(state, env.observation_space.low)
done = False
step_index = 0
while done != True:
   action = np.argmax(Q[state_adj[0], state_adj[1]]) # Best action using the learned Q table
   state, reward, done, info = env.step(action)
   state_adj = discretize_state(state, env.observation_space.low)
   show_state(env, step=step_index, info='State ({},{}) Reward: {}'.format(d_state[0], d_state[1], reward))
   step_index = step_index + 1
```

Now you can change the parameters of Q-learning function and can see how the performance varies.

You can plot the rewards to see

```
# Plot Rewards
plt.plot(100*(np.arange(len(rewards)) + 1), rewards)
plt.xlabel('Episodes')
plt.ylabel('Average Reward')
plt.title('Average Reward vs Episodes')
#plt.savefig('rewards.jpg')
plt.show()
```



Cartpole with RL

`Cartpole` is a classic control Reinforcement Learning problem that was first introduced by by Barto, Sutton, and Anderson [Barto83]. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

Cartpole Problem definition:

> Objective: Prevent the pole (pendulum) from falling over.

> State: {Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity}

> Action: {Push cart to the left, Push cart to the right}.

> Reward: +1 for every timestep that the pole remains upright (we will change this slightly in our implementation)

```
import matplotlib.pyplot as plt
%matplotlib inline
from IPython import display
#!pip install gym
import numpy as np
import gym
```

Only uncomment the following block if the visualization of the environment gives an error. On mac you need to install pyglet version 1.5.11 to get the gym environment to render. The installation will give an error, but it will work.

```
#!pip install pyglet==1.5.11
```

```
Requirement already satisfied: pyglet==1.5.11 in /Users/thienbao/opt/anaconda3/lib/python3.9/site-packages (1.5.11)
```

```
#!pip install pygame
```

```
Collecting pygame
  Using cached pygame-2.1.2-cp39-cp39-macosx_10_9_x86_64.whl (8.9 MB)
Installing collected packages: pygame
Successfully installed pygame-2.1.2
```

To begin with this environment, import and initialize it as follows:

```
env = gym.make('CartPole-v0')
state = env.reset()
print(state)
```

```
[ 0.0486408  -0.02611125 -0.0323816  -0.04627936]
```

```
print('State space Low: ', env.observation_space.low)
print('State space High: ', env.observation_space.high)
```

```
State space Low:  [-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
State space High:  [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
```

This shows that the first state variable (Cart Position) has a range [-4.8, 4.8] and the second state variable (Cart Velocity) has a range [-∞, ∞].... The state space of the environment is a continuous state space, which means that there are infinitely many state-action pairs, making it impossible to build a Q table. As a solution to this problem we can descritize the state space. One simple discritization is to conver the stat espace to a grid where there are 20 grid positions in along each dimention. Note that we have truncated the two dimetions with infinite limits.

```
numBins = 20
bins = [np.linspace(-4.8, 4.8, numBins),
        np.linspace(-4, 4, numBins),
        np.linspace(-.418, .418, numBins),
        np.linspace(-4, 4, numBins)]
obsSpaceSize = len(env.observation_space.high)
```

We can also write a function that will convert a continuous state vector to a descrete one.

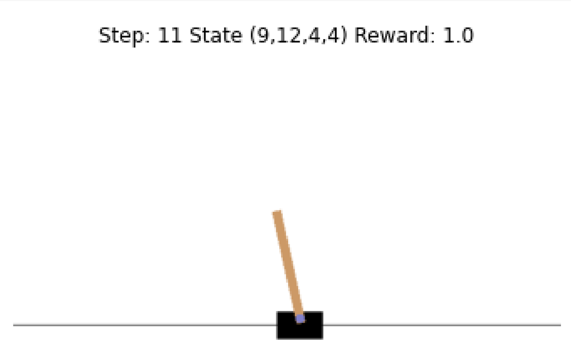
```
def discretize_state(state, bins, obsSpaceSize):
    stateIndex = []
    for i in range(obsSpaceSize):
        stateIndex.append(np.digitize(state[i], bins[i]) - 1) # -1 will turn bin into index
    return tuple(stateIndex)
```

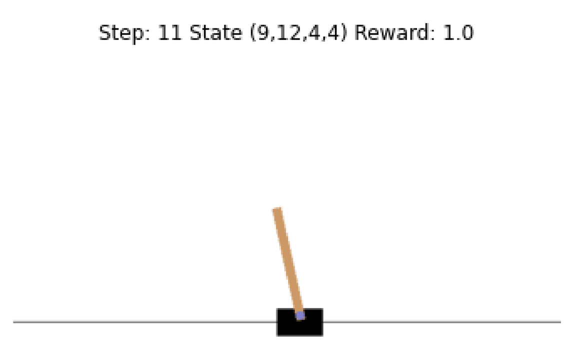
Lets now make some random actions in the environment and see what the output will be. For this we need a function to plot the output of the environment.

```
def show_state(env, step=0, info=""):
    plt.figure(1)
    plt.clf()
    plt.imshow(env.render(mode='rgb_array'))
    plt.title("Step: %d %s" % (step, info))
    plt.axis('off')

    display.clear_output(wait=True)#clear_output(wait=True)
    display.display(plt.gcf())
    #display.display(plt.gcf())
```

```
env.reset()
done = False
step_index = 0
while done != True:
    action = env.action_space.sample()    # get a random action from the set of actions
    state, reward, done, info = env.step(action) # perform the action and receive new state and reward
    d_state = discretize_state(state, bins, obsSpaceSize)
    show_state(env, step=step_index, info='State ({},{},{},{}) Reward: {}'.format(d_state[0], d_state[1],d_state[2], d_state[3], reward))
    step_index = step_index + 1
```





Did the pole remain upright for a long time?

Learning

We are going to use Q-learning for this task. Lets first define some hyper parameters. You may change them to get better performance later.

```
LEARNING_RATE = 0.1
DISCOUNT = 0.95
EPISODES = 50000

# parameters for epsilon decay policy
EPSILON = 1 # not a constant, going to be decayed
START_EPSILON_DECAYING = 1
END_EPSILON_DECAYING = EPISODES // 2
epsilon_decay_value = EPSILON / (END_EPSILON_DECAYING - START_EPSILON_DECAYING)

#for testing
N_TEST_RUNS = 100
TEST_INTERVAL = 5000
```

Write a function to test a given model. The function outputs two performace values. Was the run successful (pole upright for 200 steps), and the number of steps it ran.

> ☞ **Task: Identify if there are better performace measures to be used for this task and discuss with tutor.**

```
def test_model(Qtable):
    state = env.reset()
    dstate = discretize_state(state, bins, obsSpaceSize)
    done = False
    steps = 0
    while not done:
        action = np.argmax(Qtable[dstate])
        state, reward, done, _ = env.step(action)
        dstate = discretize_state(state, bins, obsSpaceSize)
        steps = steps + 1

    success_run = 0
    if steps > 199:
        success_run = 1

    return success_run, steps
```

Now lets develop a function for Q learning. The function prototype is given below. Assume that Q is a numpy matrix with dimentions (number of elemets for Cart Position, number of elements for Cart Velocity, number of elements for Pole Angle, number of elements for Pole Angular Velocity, number of actions).

� Complete the following function using the knowladge gained in the lecture.

```
def QLearning(env, QTable):
    # Env: The OpenAI gym environment
    # QTable: Initial Q table

    for episode in range(EPISODES):
        done = False

        # get the initial state
        state = env.reset()
        discretState = discretize_state(state, bins, obsSpaceSize)

        epsilon = EPSILON

        steps = 0;
        while done != True:

            # Determine next action - epsilon greedy strategy for explore vs exploitation
            if np.random.random() < 1 - epsilon:
                # select the best action according to Qtable (exploitation)
                # TODO
                action = np.argmax(Qtable[discretState])
            else:
                # select a random action (exploration)
                # TODO
                action = env.action_space.sample()

            # Step and Get the next state and reward
            # TODO
            state, reward, done, _ = env.step(action)
            discretStateNew = discretize_state(state, bins, obsSpaceSize)


            #Allow for terminal states
            if done and steps < 200:
                reward = -375    # what is happending here?

            # Update the Q table
            # TODO
            QTable[ discretStateNew[0],  discretStateNew[1],  discretStateNew[2],  discretStateN

            # Update variables
            discretState = discretStateNew
            steps = steps + 1


        # Update epsilon
        if END_EPSILON_DECAYING >= episode and episode >= START_EPSILON_DECAYING:
            epsilon -= epsilon_decay_value

        # test the model and print test results
        if episode % TEST_INTERVAL == 0:
            success_run_ = list()
            steps_ = list()
            for i in range(N_TEST_RUNS):
                success_run, steps = test_model(QTable)
                success_run_.append(success_run)
                steps_.append(steps)

            print('Testing at Episode {}:'.format(episode))
            print('\t Successful Runs: {}/{}'.format(np.sum(success_run_), N_TEST_RUNS) )
            print('\t Average Steps: {}'.format(np.mean(steps_)))

    env.close()
    return QTable
```

Sample Solutions

If you are struggling with the above function, a sample solution has been provided. Only use this if you have **made your absolute best attempts** at implementing the function yourself. The purpose of this lab is to understand common aspects of RL algorithm, though the Q-learning algorithm. You will gain significantly less out of this lab if you don't try to solve the problems yourself.

> ☞ **Task: Identify what is happenning in the epsilon decay policy. Discuss with tutor.**