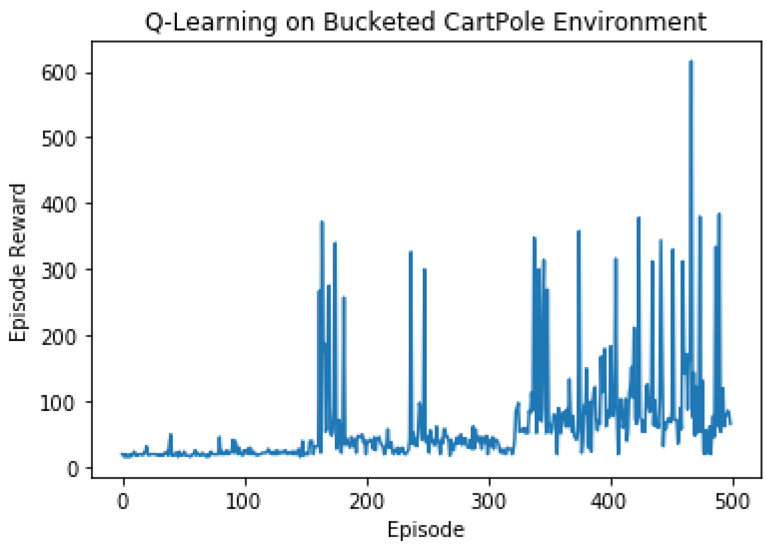
Unless cited, code is my own. I also referenced some tutorials on evolutionary neural networks and keras documentation as writing a neural network from scratch was time consuming. ChatGPT and geeksforgeeks were used in plotting and Python help.

1) Create a Q-learning agent that learns to solve the "Cart Pole" environment. The agent should balance the pole for 100 time steps.

How will you handle the continuous state space?

I discretized the state space into 3 buckets each for cart position and velocity and 6 buckets for pole angle and pole velocity. This allowed me to “fake” a discrete space for solving.



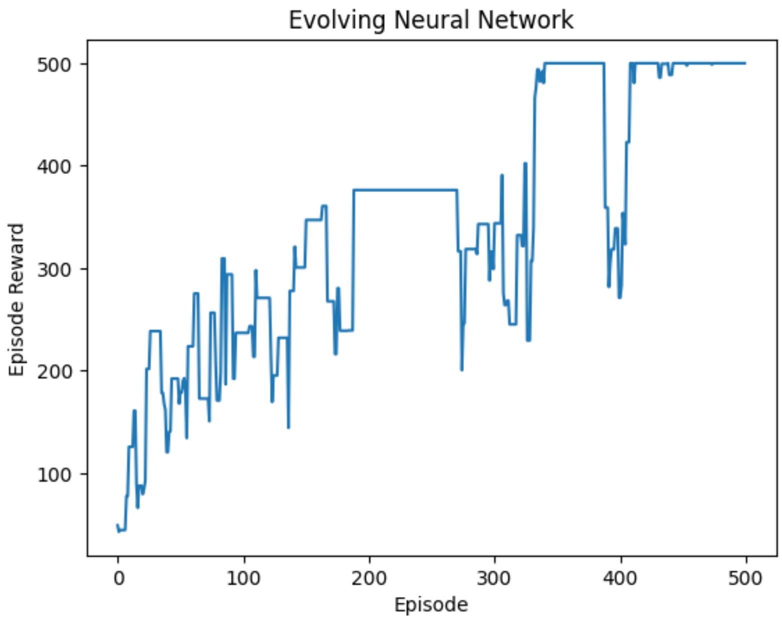
2) Evolve a neural network to solve the previous task.

What will you use for your evaluation function?

The network is evaluated by running the cartpole simulation ten times and averaging the reward across all runs.

What mapping should the network learn?

The network learns the mapping of a point in the observation space to an action. The actions are chosen based on the highest q value of the actions which incorporates the reward. Hence the network improves over time.



Algorithm Description

Q learning: I first discretized the state space so there were 3 buckets each for cart position and velocity and 6 buckets for pole angle and pole velocity. Then following the same format as standard q learning, I learned for a specified number of epochs and in each I reset the environment, and ran the following for 500 timesteps: I picked an action using either the greedy approach on the discretized state space or a random sample, then stepped in the environment, discretized the state based on the step, find the best action using the q table for that state, and then update the q table using the policy. Then update the state and repeat. It's very similar to standard q learning except the state had to be discretized.

Evolving a neural network

I first created an initial network using some data from a random agent. Then I made a copy of the network and randomly modified some of the weights. Then both of these networks were run in 10 training cycles and the rewards were averaged. The better performing network was kept, and the cycle repeated. This is a very basic form of an evolutionary network where the initial population is 1. A true evolutionary network would start with more than one initial network in the population.

Comparisons

The Q Learning algorithm performed better with more and more episodes as it better learned the best actions to take while taking very little time to do so. With the specific bucketing I used there were only a few states so this probably helped. If I had increased the number of buckets the algorithm would've been more computationally expensive but performed better. The q learner also learns with each iteration. Q learning is also a lot easier to implement than the neural network solution.

The evolved neural network took a very long time to finish but performed much better sooner and there was less variation when compared to the q learner. Here however, the method of mutation was slower and with an initial population of 1, sometimes the networks evolved in the wrong direction but they managed to keep a score above 100 after the first few episodes. If the initial population of networks contained more than 1, the reward would have consistently improved as only the higher performing networks would have been kept. I would also have only evaluated fitness on the newly created networks instead of all of them to speed up slightly, but this would still be slower than Q learning. The tradeoff is that the neural network can't change with each iteration as it can't learn online like Q learning does with each iteration.

Both perform decently in the CartPole Environment but the evolved neural network learns in fewer episodes but is computationally more expensive.

Sources:

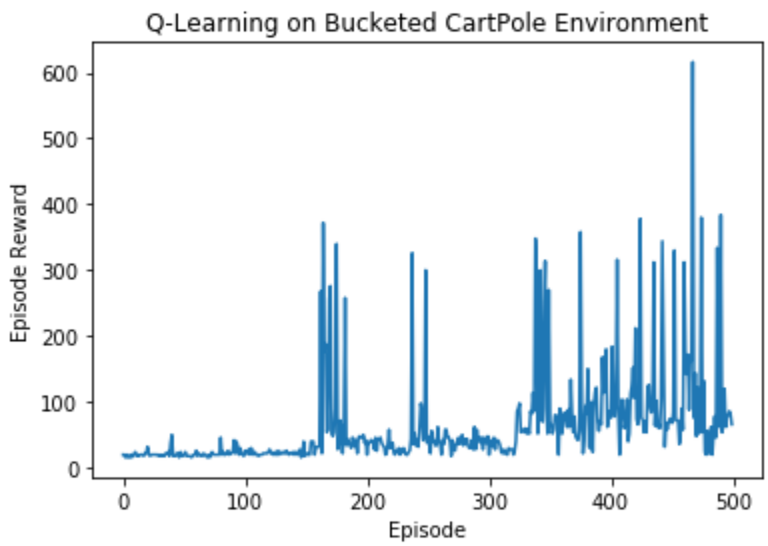
https://nandakishorej8.medium.com/part-2-evolutionary-algorithms-for-reinforcement-learning-solving-openais-cartpole-318aaef8a6eb
https://theobservator.net/neural-network-for-open-ai-cartpole-v1-challenge-with-keras/

In [141]:
```
import gymnasium as gym
import random
import numpy as np
import matplotlib.pyplot as plt
env = gym.make("CartPole-v1", render_mode="human")
```

In [143]:
```
class QLearnerSolver:
    def __init__(self,env):
        self.q_table = {}
        self.alpha = 0.1
        self.gamma = 0.95
        self.epsilon = 0.1
        self.env = env
        self.action_space = list(range(env.action_space.n))
        self.state_space_size = [3,3,6,6]
        self.q_table = np.zeros(self.state_space_size + [env.action_spa


    def convert_cont_to_discrete_space(self,state):
        if isinstance(state, tuple):
            state = state[0]
        bucket_edgesv = np.linspace(-4.8, 4.8, 3)
        bucket_edgesvdot = np.linspace(-5, 5, 3)
        bucket_edgestheta = np.linspace(-0.418, 0.418, 6)
        bucket_edgesthetadot = np.linspace(-5, 5, 6)
        bucket_indexv = np.digitize(state[0], bucket_edgesv) - 1
        bucket_indexvtheta = np.digitize(state[1], bucket_edgesvdot) -
        bucket_indexdot = np.digitize(state[2], bucket_edgestheta) - 1
        bucket_indexdottheta = np.digitize(state[3], bucket_edgesthetad

        return tuple([bucket_indexv,bucket_indexvtheta,bucket_indexdot,

    def choose_action(self, state):
        # if the random is more than epsilon
        if random.uniform(0,1) < self.epsilon:
            return self.env.action_space.sample()
        else:
            discretized_state = self.convert_cont_to_discrete_space(sta
            q_values = self.q_table[discretized_state]
            return np.argmax(q_values)


    def learn(self, num_episodes):
        total_rewards = []
        for learning_epoch in range(num_episodes):
            state = env.reset()
            total_reward = 0                 #every episode, reset the
            for time_step in range(500):
                action = self.choose_action(state) #learner chooses one
                next_state,reward,done,_,_=env.step(action)  #the actio
                discretized_state = self.convert_cont_to_discrete_space

                # Use Q-learning update rule
                max_q_next = np.max(self.q_table[self.convert_cont_to_d
                self.q_table[discretized_state + (action,)] += self.alp

                state = next_state
                total_reward += reward


                # update total reward
```

```
                total_reward = total_reward + reward
            total_rewards.append(total_reward)
        return total_rewards
```

In [144]:
```
env = gym.make('CartPole-v1')
learner1=QLearnerSolver(env)
total_rewards = learner1.learn(500)
plt.plot(range(0, 500), total_rewards)
plt.xlabel("Episode")
plt.ylabel("Episode Reward")
plt.title("Q-Learning on Bucketed CartPole Environment")
plt.show()
```



```
import gym
import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
import random
import copy

# netwrok modified from https://theobservator.net/neural-network-for-open-ai-cartp

# Define Game Commands
RIGHT_CMD = [0, 1]
LEFT_CMD = [1, 0]

# Define Reward Config
START_REWARD = 0
MIN_REWARD = 100

# Initialize Game Environment
env = gym.make('CartPole-v1')

def play_random_games(games=10):
    """
    Play Random Games to Get Some Observations
    :param games:
    :return:
    """

    # Storage for All Games Movements
    all_movements = []
    rewards = []
    for episode in range(games):
        # Reset Game Reward
        episode_reward = 0

        # Define Storage for Current Game Data
        current_game_data = []

        # Reset Game Environment
        env.reset()

        # Get First Random Movement
        action = env.action_space.sample()

        while True:

            # Play
            observation, reward, done, info = env.step(action)
```

```
            # Get Random Action (On Real, its get a "Next" movement to compensate
            action = env.action_space.sample()

            # Store Observation Data and Action Taken
            current_game_data.append(
                np.hstack((observation, LEFT_CMD if action == 0 else RIGHT_CMD))
            )

            if done:
                break

            # Compute Reward
            episode_reward += reward


        # Save All Data (Only for the Best Games)
        rewards.append(episode_reward)
        if episode_reward >= MIN_REWARD:
            print('.', end='')
            all_movements.extend(current_game_data)

    # Create DataFrame
    dataframe = pd.DataFrame(
        all_movements,
        columns=['cart_position', 'cart_velocity', 'pole_angle', 'pole_velocity_at
    )

    # Convert Action Columns to Integer
    dataframe['action_to_left'] = dataframe['action_to_left'].astype(int)
    dataframe['action_to_right'] = dataframe['action_to_right'].astype(int)

    return dataframe, rewards


def generate_ml(dataframe):
    model = Sequential()
    model.add(Dense(3, input_dim=4, activation='relu'))
    model.add(Dense(3, activation='relu'))
    model.add(Dense(2, activation='sigmoid'))

    model.compile(optimizer='adam', loss='categorical_crossentropy')
    model.fit(
        dataframe[['cart_position', 'cart_velocity', 'pole_angle', 'pole_velocity_
        dataframe[['action_to_left', 'action_to_right']],
        epochs=20
    )

    return model


def train(ml_model, games=100):
```

```
def train(ml_model, games=100):
    rewards = []
    for i_episode in range(games):
        episode_reward = 0
        observation = env.reset()
        for time_step in range(500):
            current_action_pred = ml_model.predict(observation.reshape(1, 4))
            current_action = np.argmax(current_action_pred)
            observation, reward, done, info = env.step(current_action)
            print("reward for step",reward)
            episode_reward += reward

        rewards.append(episode_reward)
    return rewards


print("[+] Gathering some data")
df,rewards = play_random_games(games=10000)
print(rewards)
```

```
/usr/local/lib/python3.10/dist-packages/tensorflow/python/framework/dtypes.py
  from tensorflow.tsl.python.lib.core import pywrap_ml_dtypes
/usr/local/lib/python3.10/dist-packages/gym/core.py:317: DeprecationWarning:
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/wrappers/step_api_compatibility.p
  deprecation(
[+] Gathering some data
.....[55.0, 15.0, 10.0, 44.0, 9.0, 18.0, 55.0, 14.0, 16.0, 56.0, 27.0, 7.0, 3
```

```
import matplotlib.pyplot as plt
print("[+] Training NN Model")
ml_model1 = generate_ml(df)
max_rewards = []

for i in range(500):
  # mutating one model
  ml_model2 = copy.deepcopy(ml_model1)
  for layer in ml_model2.layers:
    weights = layer.get_weights()[0]
    bias = layer.get_weights()[1]
    for i in range(weights.shape[0]):
      for j in range(weights.shape[1]):
          if random.random() < 0.2:
              weights[i, j] = random.uniform(0,0.5)
    layer.set_weights([weights, bias])
    break

  print("[+] Playing Games with NN")

  ml1_rewards = train(ml_model=ml_model1, games=1)
  ml2_rewards = train(ml_model=ml_model2, games=1)
```

```
    if  sum(ml1_rewards) / len(ml1_rewards) < sum(ml2_rewards) / len(ml2_rewards):
      ml_model1 = copy.deepcopy(ml_model2)
      max_rewards.append(sum(ml2_rewards) / len(ml2_rewards))
    else:
      max_rewards.append(sum(ml2_rewards) / len(ml2_rewards))

  print(max_rewards)
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: Deprecatio
  and should_run_async(code)
[+] Training NN Model
Epoch 1/20
18/18 [==============================] - 2s 2ms/step - loss: 0.6948
Epoch 2/20
18/18 [==============================] - 0s 2ms/step - loss: 0.6941
Epoch 3/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6938
Epoch 4/20
18/18 [==============================] - 0s 4ms/step - loss: 0.6935
Epoch 5/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6932
Epoch 6/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6931
Epoch 7/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6929
Epoch 8/20
18/18 [==============================] - 0s 4ms/step - loss: 0.6928
Epoch 9/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6926
Epoch 10/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6926
Epoch 11/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6925
Epoch 12/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6922
Epoch 13/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6921
Epoch 14/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6919
Epoch 15/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6918
Epoch 16/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6916
Epoch 17/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6915
Epoch 18/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6914
Epoch 19/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6914
Epoch 20/20
18/18 [==============================] - 0s 3ms/step - loss: 0.6911
[+] Playing Games with NN
1/1 [==============================] - 0s 123ms/step
reward for step 1.0
1/1 [==============================] - 0s 29ms/step
reward for step 1.0
```

```
reward for step 1.0
1/1 [==============================] - 0s 30ms/step
reward for step 1.0
1/1 [==============================] - 0s 33ms/step
reward for step 1.0
1/1 [==============================] - 0s 29ms/step
reward for step 1.0
1/1 [==============================] - 0s 31ms/step
reward for step 1.0
1/1 [==============================] - 0s 32ms/step
reward for step 1.0
```

```
plt.plot(range(0, len(max_rewards)), max_rewards)
plt.xlabel("Episode")
plt.ylabel("Episode Reward")
plt.title("Evolving Neural Network")
plt.show()
```

```
[15.0, 21.0, 16.0, 15.0, 19.0, 17.0, 9.0, 17.0, 16.0, 28.0]
```