Read Me for AI Project 1

I have used the AI-Gym environment provided by OpenAI to test 2 learning and control algorithms.

Method 1 (DQN):

The first method uses q learning to compute loss on the basis of reward gained. A neural network is used to make the prediction and it is updated using Markov decision process (MDP). The neural network improves by training on the replays of previous experiences over time.

Method 2 (ANN with GA):

The second method uses genetic algorithms to optimize the weights of a neural network over multiple generations. It starts by creating a random population. Each individual is then given a fitness score which the time for which they have survived in the game. Then the most fit individuals are chosen to create off springs for the next generation. This process is then continued over several generations until the desired score is achieved.

Method 1 Using DQN

Imports

```
import os
import random
import gym
import numpy as np
from collections import deque
from keras.models import Model, load_model
from keras.layers import Input, Dense
from keras.optimizers import Adam, RMSprop
```

Defining Parameters

```
#Training parameters
n_episodes = 300
n_win_ticks = 200

gamma = 1.0 # Discount Factor
epsilon = 1.0 # Exploration Factor
epsilon_decay = 0.99
epsilon_min = 0.01
lr = 0.01 #Learning rate
lr_decay = 0.01

batch_size = 64 # how may samples to train on from memmory
monitor = False
quiet = False


```

Setting up the Cart Pole environment

```
# Environment Parameter
memory = deque(maxlen=10000)
env = gym.make('CartPole-v0')
env.max_episode_steps = 500
input_shape = 4
action_space = 2
```

Neural Network Architechture

```
def OurModel(input_shape, action_space):
    # Input Layer of state size(4)
    X_input = Input(input_shape)
    # Hidden Layer with 512 nodes
    X = Dense(512, input_shape=input_shape, activation="relu")(X_input)
    # Hidden Layer with 256 nodes
    X = Dense(256, activation="relu")(X)
    # Hidden Layer with 64 nodes
    X = Dense(64, activation="relu")(X)
    # Output Layer with # of actions: 2 nodes (left, right)
    X = Dense(action_space, activation="linear")(X)
    model = Model(inputs = X_input, outputs = X, name='CartPole_DQN_model')
    model.compile(loss="mse", optimizer=RMSprop(lr=0.00025, rho=0.95, epsilon=0.01), metrics=["accuracy"])
    model.summary()
    return model
```

Agent

```python
class DQNAgent:
    def __init__(self):
        #Setting Up environment and initialising parameters
        self.env = gym.make('CartPole-v1')
        self.state_size = self.env.observation_space.shape[0]
        self.action_size = self.env.action_space.n
        self.EPISODES = 1000
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.001
        self.epsilon_decay = 0.999
        self.batch_size = 64
        self.train_start = 1000
        # creating main model
        self.model = OurModel(input_shape=(self.state_size,), action_space = self.action_size)
```

```python
    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))
        if len(self.memory) > self.train_start:
            if self.epsilon > self.epsilon_min:
                self.epsilon *= self.epsilon_decay

    def select_action(self, state):
        if np.random.random() <= self.epsilon:
            return random.randrange(self.action_size)
        else:
            return np.argmax(self.model.predict(state))
```

```python
    def replay(self):
        if len(self.memory) < self.train_start:
            return
        # Randomly sample minibatch from the memory and then taining neural network on the experience
        minibatch = random.sample(self.memory, min(len(self.memory), self.batch_size))
        state = np.zeros((self.batch_size, self.state_size))
        next_state = np.zeros((self.batch_size, self.state_size))
        action, reward, done = [], [], []
        for i in range(self.batch_size):
            state[i] = minibatch[i][0]
            action.append(minibatch[i][1])
            reward.append(minibatch[i][2])
            next_state[i] = minibatch[i][3]
            done.append(minibatch[i][4])
        target = self.model.predict(state)
        target_next = self.model.predict(next_state)

        for i in range(self.batch_size):
            # Updating Q value for the action
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = reward[i] + self.gamma * (np.amax(target_next[i]))

        # Train the Neural Network with batches
        self.model.fit(state, target, batch_size=self.batch_size, verbose=0)
```

```python
    def run(self):
        flag = 0
        for e in range(self.EPISODES):
            state = self.env.reset()
            state = np.reshape(state, [1, self.state_size])
            done = False
            i = 0
            while not done:
                #self.env.render()
                action = self.select_action(state)
                next_state, reward, done, _ = self.env.step(action)
                next_state = np.reshape(next_state, [1, self.state_size])
                if not done or i == self.env._max_episode_steps-1:
                    self.remember(state, action, reward, next_state, done)
                else:
                    self.remember(state, action, -100, next_state, done)

                state = next_state
                i += reward
                if done:

                    print(f"episode: {e}/{self.EPISODES}, score: {i}, e: {self.epsilon}")
                    if i >= 200:
                        print("|----------------------------------Solved----------------------------------|")
                        print(f"episode: {e}/{self.EPISODES}, score: {i}, e: {self.epsilon}")
                        flag = 1
                        break
                    if flag == 1:
                      break
                if flag == 1:
                      break
                self.replay()
            if flag == 1:
                break
```

Executing Model

```python
print("----------Method 1 Using DQN-------")
agent = DQNAgent()
agent.run()
```

```
----------Method 1 Using DQN-------
Model: "CartPole_DQN_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 4)]               0
_________________________________________________________________
dense (Dense)                (None, 512)               2560
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_2 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 130
=================================================================
Total params: 150,466
Trainable params: 150,466
Non-trainable params: 0
_________________________________________________________________
episode: 0/1000, score: 52.0, e: 1.0
episode: 1/1000, score: 28.0, e: 1.0
```

```
episode: 54/1000, score: 18.0, e: 0.8145057056571417
episode: 55/1000, score: 19.0, e: 0.7992255563671304
episode: 56/1000, score: 40.0, e: 0.767872106216294
episode: 57/1000, score: 14.0, e: 0.757191494352590
episode: 58/1000, score: 44.0, e: 0.72458144548308
episode: 59/1000, score: 32.0, e: 0.7017506636113059
episode: 60/1000, score: 56.0, e: 0.6635141250307047
episode: 61/1000, score: 20.0, e: 0.6503691570122084
episode: 62/1000, score: 84.0, e: 0.5979446000009478
episode: 63/1000, score: 81.0, e: 0.5513983909676525
episode: 64/1000, score: 72.0, e: 0.513074755348837
episode: 65/1000, score: 30.0, e: 0.4979036311114436
episode: 66/1000, score: 82.0, e: 0.4586858344239834
episode: 67/1000, score: 119.0, e: 0.4072006165777428
episode: 68/1000, score: 101.0, e: 0.36806348825922275
episode: 69/1000, score: 131.0, e: 0.32285067248442284
episode: 70/1000, score: 168.0, e: 0.27290011414765825
episode: 71/1000, score: 132.0, e: 0.23913776344553783
episode: 72/1000, score: 127.0, e: 0.21060329799922556
episode: 73/1000, score: 332.0, e: 0.15108009835289823
|---------------------------------Solved---------------------------------|
episode: 73/1000, score: 332.0, e: 0.15108009835289823
```

Method 2: Using NN with Genetic Algorithm

Imports ¶

```
import gym
import numpy as np
import math
from matplotlib import pyplot as plt
from random import randint
from statistics import median, mean
np.random.seed(seed=20)
```

Settingup Initial Parameters

```
award_set =[]
test_run = 15
best_gen =[]
n_of_generations = 1000
```

Setting Up Environment

]:

```
env = gym.make('CartPole-v1')

ind = env.observation_space.shape[0]
adim = env.action_space.n #discrete
```

Creating Neural Network

```
def softmax(x):
    x = np.exp(x)/np.sum(np.exp(x))
    return x

def lreLu(x):
    alpha=0.2
    return tf.nn.relu(x)-alpha*tf.nn.relu(-x)

def sigmoid(x):
    return 1/(1+np.exp(-x))

def reLu(x):
    return np.maximum(0,x)

def nn(obs,in_w,in_b,hid_w,out_w):

    obs = obs/max(np.max(np.linalg.norm(obs)),1)

    Ain = reLu(np.dot(obs,in_w)+in_b.T)

    Ahid = reLu(np.dot(Ain,hid_w))
    lhid = np.dot(Ahid,out_w)

    out_put = reLu(lhid)
    out_put = softmax(out_put)
    out_put = out_put.argsort().reshape(1,adim)
    act = out_put[0][0] #index of discrete action

    return act
```

Generate initial set of weights and bias

```python
def intial_gen(test_run):
    input_weight = []
    input_bias = []

    hidden_weight = []
    out_weight = []

    in_node = 4
    hid_node = 2

    for i in range(test_run):
        in_w = np.random.rand(ind,in_node)
        input_weight.append(in_w)

        in_b = np.random.rand((in_node))
        input_bias.append(in_b)

        hid_w = np.random.rand(in_node,hid_node)
        hidden_weight.append(hid_w)


        out_w = np.random.rand(hid_node, adim)
        out_weight.append(out_w)

    generation = [input_weight, input_bias, hidden_weight, out_weight]
    return generation
```

Run environment randomly

```python

def rand_run(env,test_run):
    award_set = []
    generations = intial_gen(test_run)

    for episode in range(test_run):# run env 10 time
        in_w  = generations[0][episode]
        in_b = generations[1][episode]
        hid_w =  generations[2][episode]
        out_w =  generations[3][episode]
        award = run_env(env,in_w,in_b,hid_w,out_w)
        award_set = np.append(award_set,award)
    gen_award = [generations, award_set]
    return gen_award
```

Genetic Algorithm

```
def run_env(env,in_w,in_b,hid_w,out_w):
    obs = env.reset()
    award = 0
    for t in range(300):
        #env.render() this slows the process theredore commented
        action = nn(obs,in_w,in_b,hid_w,out_w)
        obs, reward, done, info = env.step(action)
        award += reward
        if done:
            break
    return award

def mutation(new_dna):

    j = np.random.randint(0,len(new_dna))
    if ( 0 <j < 10): # controlling rate for amount of mutation
        for ix in range(j):
            n = np.random.randint(0,len(new_dna)) #random postion for mutation
            new_dna[n] = new_dna[n] + np.random.rand()

    mut_dna = new_dna

    return mut_dna

def crossover(Dna_list):
    newDNA_list = []
    newDNA_list.append(Dna_list[0])
    newDNA_list.append(Dna_list[1])

    for l in range(10):  # generation after crassover
        j = np.random.randint(0,len(Dna_list[0]))
        new_dna = np.append(Dna_list[0][:j], Dna_list[1][j:])

        mut_dna = mutation(new_dna)
        newDNA_list.append(mut_dna)

    return newDNA_list

#Generate new set of weights and bias from the best previous weights and bias
```

```python

def reproduce(award_set, generations):

    good_award_idx = award_set.argsort()[-2:][::-1] # here only best 2 are selected
    good_generation = []
    DNA_list = []

    new_input_weight = []
    new_input_bias = []

    new_hidden_weight = []

    new_output_weight =[]

    new_award_set = []


    #Extraction of all weight info into a single sequence
    for index in good_award_idx:

        w1 = generations[0][index]
        dna_in_w = w1.reshape(w1.shape[1],-1)

        b1 = generations[1][index]
        dna_b1 = np.append(dna_in_w, b1)

        w2 = generations[2][index]
        dna_whid = w2.reshape(w2.shape[1],-1)
        dna_w2 = np.append(dna_b1,dna_whid)

        wh = generations[3][index]
        dna = np.append(dna_w2, wh)


        DNA_list.append(dna) # make 2 dna for good gerneration

    newDNA_list = crossover(DNA_list)

    for newdna in newDNA_list: # collection of weights from dna info

        newdna_in_w1 = np.array(newdna[:generations[0][0].size])
        new_in_w = np.reshape(newdna_in_w1, (-1,generations[0][0].shape[1]))
        new_input_weight.append(new_in_w)

        new_in_b = np.array([newdna[newdna_in_w1.size:newdna_in_w1.size+generations[1][0].size]]).T #bias
        new_input_bias.append(new_in_b)

        sh = newdna_in_w1.size + new_in_b.size
        newdna_in_w2 = np.array([newdna[sh:sh+generations[2][0].size]])
        new_hid_w = np.reshape(newdna_in_w2, (-1,generations[2][0].shape[1]))
        new_hidden_weight.append(new_hid_w)

        sl = newdna_in_w1.size + new_in_b.size + newdna_in_w2.size
        new_out_w   = np.array([newdna[sl:]]).T
        new_out_w = np.reshape(new_out_w, (-1,generations[3][0].shape[1]))
        new_output_weight.append(new_out_w)

        new_award = run_env(env, new_in_w, new_in_b, new_hid_w, new_out_w) #bias
        new_award_set = np.append(new_award_set,new_award)

    new_generation = [new_input_weight,new_input_bias,new_hidden_weight,new_output_weight]

    return new_generation, new_award_set

```

```python
def evolution(env,test_run,n_of_generations):
    gen_award = rand_run(env, test_run)
    current_gens = gen_award[0]
    current_award_set = gen_award[1]
    best_gen =[]
    A =[]
    for n in range(n_of_generations):
        new_generation, new_award_set = reproduce(current_award_set, current_gens)
        current_gens = new_generation
        current_award_set = new_award_set
        avg = np.average(current_award_set)
        a = np.amax(current_award_set)
        print(f"generation: {n+1}, score: {a}")
        if np.amax(current_award_set) >= 200:
            print("|---------------------------------Solved---------------------------------|")
            print(f"generation: {n}/{n_of_generations}, score: {np.amax(current_award_set)}")
            break

        A = np.append(A, a)

    Best_award = np.amax(A)

```

Executing Model

```python
print("----------Method 2: Using NN with Genetic Algorithm-------")

evolution(env, test_run, n_of_generations)
```

```
----------Method 2 Using NN with Genetic Algorithm-------
generation: 1, score: 10.0
generation: 2, score: 27.0
generation: 3, score: 37.0
generation: 4, score: 110.0
generation: 5, score: 109.0
generation: 6, score: 107.0
generation: 7, score: 160.0
generation: 8, score: 144.0
generation: 9, score: 158.0
generation: 10, score: 150.0
generation: 11, score: 177.0
generation: 12, score: 176.0
generation: 13, score: 300.0
|---------------------------------Solved---------------------------------|
generation: 12/1000, score: 300.0
```