Alex Beyer - ENME743 - ILA5 - Linear Policy Gradients

In [ ]:
```
#Alex Beyer - ILA5 - Linear policy Gradients
#implements linear policy gradients with a gaussian potential (i think?) via a NN u

#necessary imports
import sklearn.preprocessing
import numpy as np
import gym
import matplotlib.pyplot as plt
import torch.nn.functional as func
import torch.optim as optim
import torch.nn as nn
import torch

#boilerplate setup
device = torch.device("cpu") #i ran this via CUDA but that throws errors for unsupp

#plotter class with data storage for later
class Plotter():
    def __init__(self):
        self.data = []

#actor-critic agent
class ActorCritic(nn.Module):
    def __init__(self, nStates, nActions, nHidden = 16):
        super(ActorCritic, self).__init__()
        #build NN
        self.nActions = nActions
        self.layer1 = nn.Linear(nStates, nHidden)
        self.layer2 = nn.Linear(nHidden, nHidden)
        self.layer3 = nn.Linear(nHidden, nActions)
        self.value = nn.Linear(nHidden, 1)
        self.to(device)

    #forward pass - I'm trying a different way to build this than in the NN homewor
    #personally I think this makes more sense but that's just how I think about the
    #expirimented a bit and relu for all 3 layers gives some interesting properties
    # but it still seems to correspond to great model preformance
    def forward(self, x):
        x = func.relu(self.layer1(x))
        x = func.relu(self.layer2(x))
        mu = func.relu(self.layer3(x)) #try different activators
        sigma = func.softmax(self.layer3(x), dim = -1) + .0001 #offset to prevent s
        dist = torch.distributions.Normal(mu.view(self.nActions).data, sigma.view(s
        value = self.value(x)
        return dist, value

#build the A2C trainer as a class which calls the ActorCritic agent class in itself
class A2C:
    def __init__(self, envName, gamma = .5, learnRate = .05, nEps = 100, nSteps = 1
        #store all the variables we'll need later
        self.envName = envName
        self.env = gym.make(envName)
```

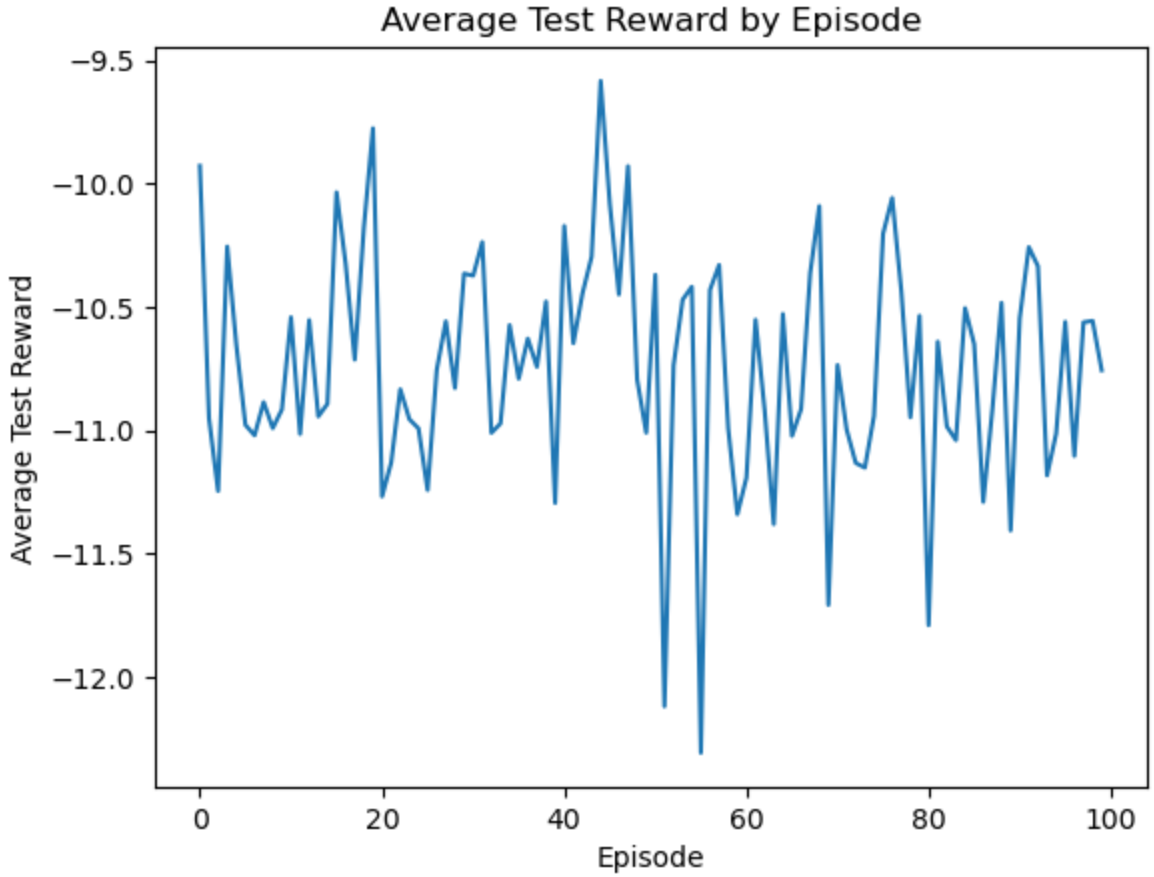
```
        self.model = ActorCritic(self.env.observation_space.shape[0], self.env.acti
        self.opt = optim.Adadelta(self.model.parameters(),learnRate) #tested across
        self.data = {"loss": []}
        self.nEps = nEps
        self.nEpsTest = nEpsTest
        self.nSteps = nSteps
        self.gamma = gamma

        #draw 10000 samples from the state space to initialize Q table
    def initStateScaler(self):
        ssSample = np.array([self.env.observation_space.sample() for x in range(100
        self.scaler = sklearn.preprocessing.StandardScaler()
        self.scaler.fit(ssSample)

    #continue to this as the model runs
    def scaleState(self, state):
        scaled = self.scaler.transform(np.array([state[0]]).reshape(1,-1))
        return scaled[0]

    #update the A2C policy
    def a2cUpdate(self, rewards, lProbs, values, state):
        qVals = []
        nextQ = 0
        pw = 0
        #apply the discount rate to the rewards found by the model, appending the t
        for reward in rewards[::-1]:
            nextQ += self.gamma ** pw * reward
            pw += 1
            qVals.append(nextQ)
        #update Q-table, stepping through in reverse to update
        qVals = qVals[::-1]
        qVals = torch.tensor(qVals)
        qVals = (qVals - qVals.mean()) / (qVals.std() + .000001) #offset to prevent
        #reset loss
        loss = 0
        for nextProb, value, nextQ in zip(lProbs, values, qVals):
            #update A2C parameters
            advantage = nextQ - value.item()
            actorLoss = -nextProb * advantage
            criticLoss = func.smooth_l1_loss(value[0], nextQ)
            loss += criticLoss + actorLoss

        self.opt.zero_grad()
        loss.min().backward()
        self.opt.step()

        #pass the current state into the actor-critic NN model to find the optimal
    def getNextAction(self, state):
        if type(state) is tuple:
            dist, value = self.model(torch.Tensor(state[0].reshape(1,-1)))
        else:
            dist, value = self.model(torch.Tensor(state))
        action = dist.sample().numpy()
        nextProb = dist.log_prob(torch.FloatTensor(action))
        return action, nextProb, value
```

```
    #also define a test function to be callled immediately after training, in which
    def test(self, nEps, tEp):
        testReward = []
        #for each testing episode...
        for e in range(self.nEpsTest):
            #reinitialize our environment...
            state = self.env.reset()
            nextReward = []
            #and for each iteration of the testing episode...
            for t in range(self.nSteps):
                #run the same logic as the training step, but without updating para
                action, _, _ = self.getNextAction(state)
                _, reward, truncated, terminated, _ = self.env.step(action)
                nextReward.append(reward)
                if truncated or terminated:
                    break
                #and appending the reward to a list for return <-- this is what im
            testReward.append(sum(nextReward))
        return np.mean(testReward), np.std(testReward)

    #setup train loop
    def train(self):
        #initialize storage and zero score
        score = 0.0
        rewards = []
        muRewards = []
        sigmaRewards = []
        #setup initial Q-table
        self.initStateScaler()
        #for each episode...
        for e in range(self.nEps):
            #initialize empty storage...
            state = self.env.reset()
            score = 0.0
            stepNum = 0
            rewards = []
            lProbs = []
            values = []
            #then for each training run in the episode...
            for t in range(self.nSteps):
                #run the model, updating parameters and ending the episode if the m
                stepNum += 1
                if type(state) is tuple:
                    state = self.scaleState(state[0].reshape(1,-1))
                else:
                    state = self.scaleState(state.reshape(1,-1))
                action, nextProb, value = self.getNextAction(state)
                state, reward, truncated, terminated, _ = self.env.step(action)
                score += reward
                rewards.append(reward)
                values.append(value)
                lProbs.append(nextProb)
                if truncated or terminated:
                    break
            #appending the score to the list...
            rewards.append(score)
```

```
            #updating the actor-critic parameters...
            self.a2cUpdate(rewards, lProbs, values, state)
            #while printing progress...
            if (e+1) % 2 == 0:
                print("ep {} got reward: {} in {} steps ".format(e+1, rewards[e], s
            if (e+1) % 10 == 0:
                nextMuReward, nextSigmaReward = self.test(self.nEpsTest,e)
                print('ave reward: {} \n w/stdev: {}'.format(nextMuReward, nextSigm
            #and saving episode mean statistics for plotting
            nextMuReward, nextSigmaReward = self.test(self.nEpsTest,e)
            muRewards.append(nextMuReward)
            sigmaRewards.append(nextSigmaReward)
        #finally, clean up
        self.env.close()
        return rewards, lProbs, values, muRewards, sigmaRewards

if __name__ == "__main__":
    A2C = A2C("MountainCarContinuous-v0")
    rewards, lProbs, values, muRewards, sigmaRewards = A2C.train()
    plt.plot(muRewards)
    plt.xlabel("Episode")
    plt.ylabel("Average Test Reward")
    plt.title("Average Test Reward by Episode")
    plt.show()
```

```
ep 2 got reward: -0.14875118833160741 in 100 steps
ep 4 got reward: -0.14048883137875237 in 100 steps
ep 6 got reward: -0.09316271620217301 in 100 steps
ep 8 got reward: -0.04384282112120311 in 100 steps
ep 10 got reward: -0.01466513933312541 in 100 steps
ave reward: -10.234614109530359
 w/stdev: 0.8931788845071033
ep 12 got reward: -0.0074187987758059576 in 100 steps
ep 14 got reward: -0.09975012672305326 in 100 steps
ep 16 got reward: -0.005493497946716364 in 100 steps
ep 18 got reward: -9.516945003804623e-05 in 100 steps
ep 20 got reward: -0.06444106109717183 in 100 steps
ave reward: -10.59537769752859
 w/stdev: 1.3961493565075058
ep 22 got reward: -0.318640788164295 in 100 steps
ep 24 got reward: -0.4565262380481329 in 100 steps
ep 26 got reward: -0.1163728341697734 in 100 steps
ep 28 got reward: -0.19777734274561143 in 100 steps
ep 30 got reward: -0.02890545001025977 in 100 steps
ave reward: -10.756248638869412
 w/stdev: 1.5040569151179666
ep 32 got reward: -0.18194471661192893 in 100 steps
ep 34 got reward: -0.05791988955024863 in 100 steps
ep 36 got reward: -0.19165637180563522 in 100 steps
ep 38 got reward: -0.023139638589158552 in 100 steps
ep 40 got reward: -0.005981211703700584 in 100 steps
ave reward: -11.337511414829258
 w/stdev: 1.2981939307212493
ep 42 got reward: -0.013416506964248143 in 100 steps
ep 44 got reward: -0.024675229892532347 in 100 steps
ep 46 got reward: -0.005743226350280595 in 100 steps
ep 48 got reward: -0.00020774738858477806 in 100 steps
ep 50 got reward: -0.06978535998084148 in 100 steps
ave reward: -10.645117903559894
 w/stdev: 1.3260759418039287
ep 52 got reward: -0.0591845025659282 in 100 steps
ep 54 got reward: -0.11877477066879152 in 100 steps
ep 56 got reward: -0.0013500826287383917 in 100 steps
ep 58 got reward: -0.023612506443451142 in 100 steps
ep 60 got reward: -0.03229933541879291 in 100 steps
ave reward: -10.150612412829776
 w/stdev: 1.2735640369599066
ep 62 got reward: -0.05643328134611103 in 100 steps
ep 64 got reward: -0.008998493376847794 in 100 steps
ep 66 got reward: -0.030828090625790595 in 100 steps
ep 68 got reward: -0.0022365681127710425 in 100 steps
ep 70 got reward: -0.024747164816802416 in 100 steps
ave reward: -10.727431695314417
 w/stdev: 0.8904930252233596
ep 72 got reward: -0.010637662194853093 in 100 steps
ep 74 got reward: -0.14545541070729087 in 100 steps
ep 76 got reward: -0.023554571564341487 in 100 steps
ep 78 got reward: -0.1085476946671463 in 100 steps
ep 80 got reward: -0.02678180158000032 in 100 steps
ave reward: -10.383562546524404
 w/stdev: 1.2948054112783502
```

```
ep 82 got reward: -0.011687905684772204 in 100 steps
ep 84 got reward: -0.45367207169124985 in 100 steps
ep 86 got reward: -0.4354669971853639 in 100 steps
ep 88 got reward: -0.004834789580066024 in 100 steps
ep 90 got reward: -0.14011046872397515 in 100 steps
ave reward: -10.525630952467273
 w/stdev: 1.518859588560474
ep 92 got reward: -0.08040949856749968 in 100 steps
ep 94 got reward: -0.07741812818488292 in 100 steps
ep 96 got reward: -0.0036146321388273784 in 100 steps
ep 98 got reward: -0.0001728330679662271 in 100 steps
ep 100 got reward: -0.0006844984400926402 in 100 steps
ave reward: -11.09894632141776
 w/stdev: 1.5055710550979748
```



My algorithm runs a number of training episodes on the MountainCar, then runs a smaller number of testing episodes, taking the average reward from those for datapoints on the plot. The outputs we see in text form here are stats form the training episodes while the plot is testing episode specific. The reason the graph looks like this is because it's converging on a solution within the first batch of training episodes - what we're seeing is a model preforming almost as well as it can. Because there is always some amount of random exploration going on we'll always see some amount of oscillation in the average episode reward, just from the model attempting to explore and improve it's policy. Overall the model preforms excellently.