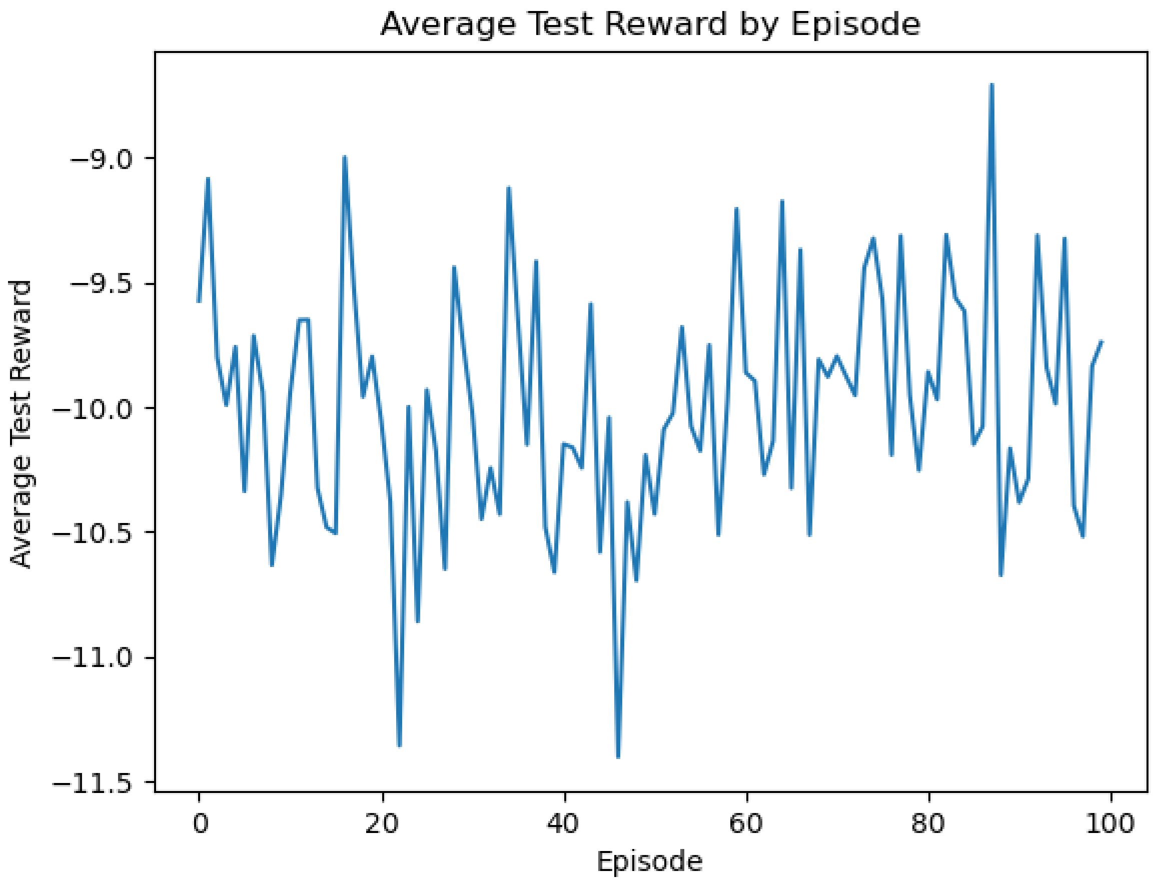
Alex Beyer - ENME743 - ILA5 - Linear Policy Gradients

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 F
import torch.optim as optim
import torch.nn as nn
import torch
import time

#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 = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        mu = F.relu(self.layer3(x)) #try different activators
        sigma = F.softmax(self.layer3(x), dim = -1) + .0001 #offset to prevent sing
        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.Adam(self.model.parameters(),learnRate)
        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 = F.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 called 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: -0.03687828643517186 in 100 steps
ep 4 got reward: -0.0007663472171788367 in 100 steps
ep 6 got reward: -0.026043660905097354 in 100 steps
ep 8 got reward: -0.019754501356337162 in 100 steps
ep 10 got reward: -0.35160063310548967 in 100 steps
ave reward: -10.57721523966972
 w/stdev: 1.2167888991818803
ep 12 got reward: -0.03382165581290373 in 100 steps
ep 14 got reward: -0.2541819198779422 in 100 steps
ep 16 got reward: -0.03163425507422204 in 100 steps
ep 18 got reward: -0.007102652099855167 in 100 steps
ep 20 got reward: -0.0010033315704291114 in 100 steps
ave reward: -10.140572384642525
 w/stdev: 1.7455312139178716
ep 22 got reward: -0.0320192517823731 in 100 steps
ep 24 got reward: -0.0021206248641335358 in 100 steps
ep 26 got reward: -0.05417013541392066 in 100 steps
ep 28 got reward: -0.01976220860223208 in 100 steps
ep 30 got reward: -0.03532728008078188 in 100 steps
ave reward: -10.755380517315656
 w/stdev: 1.4791357023190028
ep 32 got reward: -0.000691973591543521 in 100 steps
ep 34 got reward: -0.007294169328130895 in 100 steps
ep 36 got reward: -0.18640956833682054 in 100 steps
ep 38 got reward: -0.08907999047274906 in 100 steps
ep 40 got reward: -0.024344400852170624 in 100 steps
ave reward: -9.729323907077498
 w/stdev: 1.1218932056939646
ep 42 got reward: -0.2278896249807133 in 100 steps
ep 44 got reward: -0.05578884914055813 in 100 steps
ep 46 got reward: -0.11979802337624648 in 100 steps
ep 48 got reward: -0.19871664708489334 in 100 steps
ep 50 got reward: -0.2741130013964721 in 100 steps
ave reward: -9.591511114604295
 w/stdev: 1.4406944178432581
ep 52 got reward: -0.0013760995145119248 in 100 steps
ep 54 got reward: -0.1550844670960089 in 100 steps
ep 56 got reward: -0.04365257020470068 in 100 steps
ep 58 got reward: -0.006850077567153168 in 100 steps
ep 60 got reward: -0.02753429145517572 in 100 steps
ave reward: -9.426113637522118
 w/stdev: 1.1592682676774686
ep 62 got reward: -0.02024390540687229 in 100 steps
ep 64 got reward: -0.06148259834913575 in 100 steps
ep 66 got reward: -0.01314033633629066 in 100 steps
ep 68 got reward: -0.21207232947313204 in 100 steps
ep 70 got reward: -0.03715687864713715 in 100 steps
ave reward: -9.906087920728709
 w/stdev: 1.3829214130454057
ep 72 got reward: -0.0650921732257789 in 100 steps
ep 74 got reward: -0.10166113285100097 in 100 steps
ep 76 got reward: -0.10436753448494188 in 100 steps
ep 78 got reward: -0.15271872716148352 in 100 steps
ep 80 got reward: -0.11571121204797806 in 100 steps
ave reward: -9.80558106093804
 w/stdev: 1.1769923644385079
```

```
ep 82 got reward: -0.020013172422547944 in 100 steps
ep 84 got reward: -0.02890653957485121 in 100 steps
ep 86 got reward: -0.12659117443833026 in 100 steps
ep 88 got reward: -0.0028738234703082678 in 100 steps
ep 90 got reward: -0.3872361251743314 in 100 steps
ave reward: -9.070413907198613
 w/stdev: 0.9900829092211941
ep 92 got reward: -0.00177450320386221 in 100 steps
ep 94 got reward: -0.6473286978514864 in 100 steps
ep 96 got reward: -0.16872025488783182 in 100 steps
ep 98 got reward: -0.1556585971437258 in 100 steps
ep 100 got reward: -0.011667482210530357 in 100 steps
ave reward: -10.035762317567503
 w/stdev: 1.891576348147238
```



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.