In [5]:
```
import gym
import pybulletgym
import pybulletgym.envs
import numpy as np
import math
import matplotlib.pyplot as plt
import queue
import random
from collections import deque
import time
```

In [6]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
```

In [7]:
```
env = gym.make("modified_gym_env:ReacherPyBulletEnv-v1", rand_init =
False)
env.reset()
```

```
current_dir=/home/apurba/.virtualenvs/276c_assgn/lib/python3.6/site-p
ackages/pybullet_envs/bullet
options=
```

Out[7]:
```
array([ 0.3928371 ,  0.3928371 , -0.68091764,  0.26561381,  0.5
,
        0.        ,  0.08333333,  0.        ])
```

In [24]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable


class Actor(nn.Module) :
    def __init__(self, state_dim, action_dim, hidden_size_one, hidden
_size_two):

        super(Actor, self).__init__()
        self.input_size = state_dim;
        self.hidden_size_one = hidden_size_one;
        self.hidden_size_two = hidden_size_two;
        self.output_size = action_dim

        self.l1 = nn.Linear(self.input_size, self.hidden_size_one, bi
as = False)
        self.l2 = nn.Linear(self.hidden_size_one, self.hidden_size_tw
o, bias = False)
        self.l3 = nn.Linear(self.hidden_size_two, self.output_size, b
ias = False)

        self.model = torch.nn.Sequential(
                                        self.l1,
                                        nn.ReLU(),
                                        #nn.Tanh(),
                                        self.l2,
                                        nn.ReLU(),
                                        self.l3,
                                        nn.Tanh()
                                        )
        self.model.apply(self.weights_init_uniform)


    # takes in a module and applies the specified weight initializati
on
    def weights_init_uniform(self, m):
        classname = m.__class__.__name__
        # apply a uniform distribution to the weights and a bias=0
        if classname.find('Linear') != -1:
            m.weight.data.uniform_(-0.003, 0.003)
            #m.bias.data.fill_(0)

    def forward (self, state):



        return self.model(state)

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size_one, hidden
_size_two):

        super(Critic, self).__init__()
```

```
        self.input_size = (state_dim + action_dim);
        self.hidden_size_one = hidden_size_one;
        self.hidden_size_two = hidden_size_two;
        self.output_size = 1

        self.l1 = nn.Linear(self.input_size, self.hidden_size_one, bi
as = False)
        self.l2 = nn.Linear(self.hidden_size_one, self.hidden_size_tw
o, bias = False)
        self.l3 = nn.Linear(self.hidden_size_two, self.output_size, b
ias = False)
        self.model = torch.nn.Sequential(
                                    self.l1,
                                    nn.ReLU(),
                                    #nn.Tanh(),
                                    self.l2,
                                    nn.ReLU(),
                                    self.l3,
                                    nn.Tanh()
                                    )
        self.model.apply(self.weights_init_uniform)

    def weights_init_uniform(self, m):
        classname = m.__class__.__name__
        # apply a uniform distribution to the weights and a bias=0
        if classname.find('Linear') != -1:
            m.weight.data.uniform_(-0.0003, 0.0003)
            #m.bias.data.fill_(0)


    def forward (self, state, action):

        stateAction = torch.cat([state, action], 1)
        return self.model(stateAction)
```

```
UsageError: Cell magic `%%write` not found.
```

In [9]:
```
class replayBuffer:
    def __init__(self, buffer_size):
        self.buffer_size = buffer_size;
        self.buffer = deque(maxlen = buffer_size)

    def push (self, state, action, next_state, reward, done):
        samples = (state, action, next_state, reward, done)
        self.buffer.append(samples)

    def sample(self, batch_size):
        state_batch = []
        action_batch = []
        next_state_batch = []
        reward_batch = []
        done_batch = []

        batch_data = random.sample(self.buffer, batch_size)

        for samples in batch_data:
            state, action, next_state, reward, done = samples
            state_batch.append(state)
            action_batch.append(action)
            reward_batch.append(reward)
            next_state_batch.append(next_state)
            done_batch.append(done)
        return (state_batch, action_batch, next_state_batch, reward_b
atch, done_batch)

    def __len__(self):
        return len(self.buffer)
```

In [22]:
```
class DDPG():
    def __init__(self,
                 env,
                 action_dim,
                 state_dim,
                 actor,
                 critic,
                 actor_target,
                 critic_target,
                 noise = 1,
                 d_param = 0.001,
                 critic_lr = 0.0003,
                 actor_lr = 0.0003,
                 gamma = 0.99, batch_size = 500, buffer_size = 10000
):

        """
        param: env: An gym environment
        param: action_dim: Size of action space
        param: state_dim: Size of state space
        param: actor: actor model
        param: critic: critic model
        param: critic_lr: Learning rate of the critic
        param: actor_lr: Learning rate of the actor
        param: gamma: The discount factor
        param: batch_size: The batch size for training
        """


        self.env = env
        self.action_dim = action_dim
        self.state_dim = state_dim
        self.critic_lr = critic_lr
        self.actor_lr = actor_lr
        self.gamma = gamma
        self.batch_size = batch_size

        self.d = d_param
        self.noise = noise

        self.actor = actor
        self.critic = critic
        self.actor_target = actor_target
        self.critic_target = critic_target
        self.actor_optimizer = optim.Adam(self.actor.parameters())# l
r= self.actor_lr)
        self.critic_optimizer = optim.Adam(self.critic.parameters())#
lr = self.critic_lr)

        self.iterations = []
        self.return_history = []
        self.return_reward = []

        self.replay_buffer = replayBuffer(buffer_size)
        self.loss = nn.MSELoss()
```

```
    def updateQpolicy(self, batch_size, iterationNo):
        states, actions, state_next,rewards, _ = self.replay_buffer.s
ample(batch_size)
        states = torch.FloatTensor(states)
        actions = torch.FloatTensor(actions)
        rewards = torch.FloatTensor(rewards).reshape([batch_size,1])
        state_next = torch.FloatTensor(state_next)
        Q_pres = self.critic.forward(states, actions)
        action_next = self.actor_target.forward(state_next).detach()
        Q_next = self.critic_target.forward(state_next, action_next.d
etach()).detach()#while doing loss.backward we dont want target_polic
y parameters to be updated
        Q_nexttarget = rewards + Q_next * self.gamma
        #wrt Q parameter maps s and actions to theQ value
        criticLoss = self.loss(Q_nexttarget, Q_pres)
        #wrt policy parameter, maps states to actions
        actorLoss = -1 * self.critic.forward(states, actor.forward(st
ates)).mean()


        #update the Q paramters which maps states to actions to the Q
value
        self.critic_optimizer.zero_grad();
        criticLoss.backward();
        self.critic_optimizer.step();

        #update thw policy parameters which updates the states to act
ions
        self.actor_optimizer.zero_grad();
        actorLoss.backward();
        self.actor_optimizer.step();


        #update the target network weights with the original network
 weights
        for tar_param, src_param in zip(self.actor_target.parameters
(), self.actor.parameters()):
            tar_param.data.copy_(self.d * src_param.data + (1.0 - sel
f.d) * tar_param.data)

        for tar_param, src_param in zip(self.critic_target.parameters
(), self.critic.parameters()):
            tar_param.data.copy_(self.d * src_param.data + (1.0 - sel
f.d) * tar_param.data)

    def selectAction(self, state):
        #state = torch.FloatTensor(state)
        state = Variable(torch.from_numpy(state).float().unsqueeze(0
))
        action = self.actor.forward(state)
        action = action.detach().numpy()[0]
        return action


    def train(self, epochs):
        total_reward = 0
        for iterationNo in range(epochs):
```

```
            state = env.reset()
            batch_reward = 0

            steps = 0
            '''
            while(steps < self.batch_size):
                steps += 1
                action = self.selectAction(state)
                if(self.noise):
                    #mean = torch.zeros(2);
                    #variance = torch.diag([0.1, 0.1])
                    #c = MultivariateNormal(mean, variance)
                    #noise = c.sample()
                    noise = np.random.normal(0, 0.1)
                    action[0]+= noise
                    action[1]+= noise
                new_state, reward, done, _ = env.step(action)

                batch_reward += reward
                total_reward += reward
                self.replay_buffer.push(state, action, new_state, rew
ard, done)
                state = new_state



                if(done == True):
                    break;
            '''
            #fill up the buffer
            while(len(self.replay_buffer)< self.batch_size):
                action = self.selectAction(state)
                if(self.noise):
                    #mean = torch.zeros(2);
                    #variance = torch.diag([0.1, 0.1])
                    #c = MultivariateNormal(mean, variance)
                    #noise = c.sample()
                    noise = np.random.normal(0, 0.1)
                    action[0]+= noise
                    action[1]+= noise
                new_state, reward, done, _ = env.step(action)
                if(done == True):
                    state= env.reset()

                batch_reward += reward
                total_reward += reward
                self.replay_buffer.push(state, action, new_state, rew
ard, done)
                state = new_state

            #if(len(self.replay_buffer) >= self.batch_size):
            #if(iterationNo%self.batch_size == 0 and len(self.replay_
buffer)>= self.batch_size):
            action = self.selectAction(state)
            new_state, reward, done, _ = env.step(action)
            if(done == True):
                state = env.reset()
```

```
            batch_reward += reward
            total_reward += reward
            self.replay_buffer.push(state, action, new_state, reward,
done)
            state = new_state

            self.updateQpolicy(self.batch_size, iterationNo)
            if((iterationNo % 1000 == 0 and iterationNo!=0) or iterat
ionNo == 1):
                self.iterations.append(iterationNo)
                self.return_reward.append(total_reward/iterationNo)
                print("iteration No is", iterationNo, "reward is", to
tal_reward/iterationNo)
                #self.return_history.append(batch_reward)

            if(iterationNo%2000 == 0 and iterationNo!= 0):
                fileName = "model"+ str(iterationNo)
                torch.save(self.actor.state_dict(), fileName)


    ''''
    def train(self, epochs):
        batch_reward = 0
        for steps in range(epochs):

        for iterationNo in range(epochs):
            state = env.reset()
            #steps = 0
            done = False

            #while(steps < self.batch_size):
                #steps += 1
            action = self.selectAction(state)
            if(self.noise):
                    #mean = torch.zeros(2);
                    #variance = torch.diag([0.1, 0.1])
                    #c = MultivariateNormal(mean, variance)
                    #noise = c.sample()
                noise = np.random.normal(0, 0.1)
                action[0]+= noise
                action[1]+= noise
            new_state, reward, done, _ = env.step(action)

            batch_reward += reward
            self.replay_buffer.push(state, action, new_state, reward,
done)
            state = new_state


            if(done == True):
                state = env.reset()


            #if(len(self.replay_buffer) >= self.batch_size):
            if(iterationNo == self.batch_size and len(self.replay_buf
fer)>= self.batch_size):
                 self.updateQpolicy(self.batch_size, iterationNo)
```

```
            #    self.updateQpolicy(self.batch_size, iteration)
            if((iterationNo % 100 == 0 and iterationNo != 0) or itera
tionNo == 1):
                self.iterations.append(iterationNo)
                self.return_reward.append(batch_reward/iterationNo)
                print("iteration No is", iterationNo, "reward is", ba
tch_reward/iterationNo)


  '''
```

In [23]:
```
num_states = 8
num_actions = 2

actor = Actor(num_states, num_actions, 400, 300)
actor_target = Actor(num_states, num_actions, 400, 300)

critic = Critic(num_states, num_actions, 400, 300)
critic_target = Critic(num_states, num_actions, 400, 300)

for tar_param, src_param in zip(actor_target.parameters(), actor.parameters()):
    tar_param.data.copy_(src_param.data)

for tar_param, src_param in zip(critic_target.parameters(), critic.parameters()):
    tar_param.data.copy_(src_param.data)



#ddpgLinkArm = DDPG(env, num_actions, num_states, actor, critic, actor_target, critic_target, noise )
ddpgLinkArm = DDPG(env, num_actions, num_states, actor, critic, actor_target, critic_target)
ddpgLinkArm.train(200000)
```

```
iteration No is 1 reward is -99.06571923375013
iteration No is 1000 reward is -0.3088681302564913
iteration No is 2000 reward is -0.25310415399616515
iteration No is 3000 reward is -0.23451616190940816
iteration No is 4000 reward is -0.22522216586603008
iteration No is 5000 reward is -0.21964576824000323
iteration No is 6000 reward is -0.215928169822652
iteration No is 7000 reward is -0.21327274238168684
iteration No is 8000 reward is -0.21128117180096295
iteration No is 9000 reward is -0.20973217246039996
iteration No is 10000 reward is -0.20849297298794955
iteration No is 11000 reward is -0.2074790825104901
iteration No is 12000 reward is -0.20663417377927393
iteration No is 13000 reward is -0.20591925100670638
iteration No is 14000 reward is -0.20530646005879136
iteration No is 15000 reward is -0.2047753745705983
iteration No is 16000 reward is -0.2043106747684294
iteration No is 17000 reward is -0.20390064553122156
iteration No is 18000 reward is -0.2035361750981479
iteration No is 19000 reward is -0.20321006997381885
iteration No is 20000 reward is -0.2029165753619227
iteration No is 21000 reward is -0.2026510326178262
iteration No is 22000 reward is -0.202409630123193
iteration No is 23000 reward is -0.20218921914983223
iteration No is 24000 reward is -0.20198717575758487
iteration No is 25000 reward is -0.20180129583671733
iteration No is 26000 reward is -0.20162971437130112
iteration No is 27000 reward is -0.2014708426440639
iteration No is 28000 reward is -0.2013233188973436
iteration No is 29000 reward is -0.20118596920212126
iteration No is 30000 reward is -0.20105777615324708
iteration No is 31000 reward is -0.2009378536236551
iteration No is 32000 reward is -0.20082542625216263
iteration No is 33000 reward is -0.2007198126607606
iteration No is 34000 reward is -0.2006204116335587
iteration No is 35000 reward is -0.20052669066505405
iteration No is 36000 reward is -0.20043817641702188
iteration No is 37000 reward is -0.2003544467229374
iteration No is 38000 reward is -0.20027512385485735
iteration No is 39000 reward is -0.20019986882616603
iteration No is 40000 reward is -0.20012837654890928
iteration No is 41000 reward is -0.2000603716998114
iteration No is 42000 reward is -0.19999560517686102
iteration No is 43000 reward is -0.19993385105032693
iteration No is 44000 reward is -0.1998749039295444
iteration No is 45000 reward is -0.19981857668079667
iteration No is 46000 reward is -0.19976469844286404
iteration No is 47000 reward is -0.19971311289590726
iteration No is 48000 reward is -0.19966367674674038
iteration No is 49000 reward is -0.19961625839958028
iteration No is 50000 reward is -0.1995707367863066
iteration No is 51000 reward is -0.19952700033433776
iteration No is 52000 reward is -0.19948494605359848
iteration No is 53000 reward is -0.19944447872684937
iteration No is 54000 reward is -0.19940551018997987
iteration No is 55000 reward is -0.1993679586908147
iteration No is 56000 reward is -0.19933174831661973
```

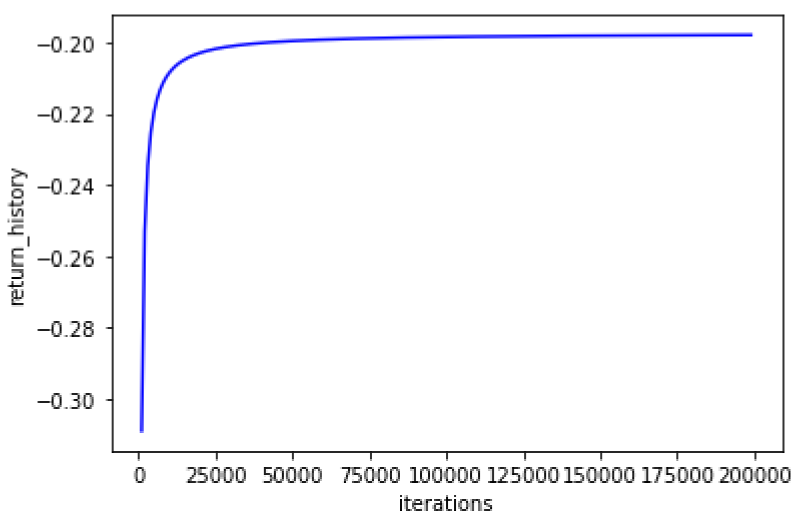
```
iteration No is 57000 reward is -0.19929680848187017
iteration No is 58000 reward is -0.19926307346900857
iteration No is 59000 reward is -0.19923048201590496
iteration No is 60000 reward is -0.19919897694457148
iteration No is 61000 reward is -0.19916850482639645
iteration No is 62000 reward is -0.1991390156797755
iteration No is 63000 reward is -0.1991104626965393
iteration No is 64000 reward is -0.19908280199402925
iteration No is 65000 reward is -0.19905599239005795
iteration No is 66000 reward is -0.19902999519832823
iteration No is 67000 reward is -0.1990047740421725
iteration No is 68000 reward is -0.19898029468472728
iteration No is 69000 reward is -0.19895652487387464
iteration No is 70000 reward is -0.19893343420047496
iteration No is 71000 reward is -0.19891099396857947
iteration No is 72000 reward is -0.19888917707645887
iteration No is 73000 reward is -0.19886795790741005
iteration No is 74000 reward is -0.19884731222941662
iteration No is 75000 reward is -0.19882721710283635
iteration No is 76000 reward is -0.1988076507953766
iteration No is 77000 reward is -0.19878859270369503
iteration No is 78000 reward is -0.19877002328103094
iteration No is 79000 reward is -0.19875192397033303
iteration No is 80000 reward is -0.19873427714240255
iteration No is 81000 reward is -0.19871706603861852
iteration No is 82000 reward is -0.19870027471785362
iteration No is 83000 reward is -0.1986838880072276
iteration No is 84000 reward is -0.19866789145637842
iteration No is 85000 reward is -0.198652271294961
iteration No is 86000 reward is -0.1986370143931114
iteration No is 87000 reward is -0.19862210822463766
iteration No is 88000 reward is -0.19860754083272014
iteration No is 89000 reward is -0.19859330079792437
iteration No is 90000 reward is -0.19857937720834626
iteration No is 91000 reward is -0.19856575963172593
iteration No is 92000 reward is -0.19855243808937995
iteration No is 93000 reward is -0.1985394030318156
iteration No is 94000 reward is -0.19852664531590156
iteration No is 95000 reward is -0.19851415618348045
iteration No is 96000 reward is -0.1985019272413181
iteration No is 97000 reward is -0.19848995044229314
iteration No is 98000 reward is -0.19847821806773808
iteration No is 99000 reward is -0.19846672271085078
iteration No is 100000 reward is -0.19845545726110123
iteration No is 101000 reward is -0.19844441488956452
iteration No is 102000 reward is -0.1984335890351168
iteration No is 103000 reward is -0.19842297339143505
iteration No is 104000 reward is -0.19841256189474715
iteration No is 105000 reward is -0.1984023487122819
iteration No is 106000 reward is -0.19839232823137262
iteration No is 107000 reward is -0.1983824950491719
iteration No is 108000 reward is -0.19837284396293786
iteration No is 109000 reward is -0.1983633699608549
iteration No is 110000 reward is -0.19835406821335527
iteration No is 111000 reward is -0.1983449340649097
iteration No is 112000 reward is -0.19833596302625778
iteration No is 113000 reward is -0.19832715076705104
```

```
iteration No is 114000 reward is -0.19831849310888303
iteration No is 115000 reward is -0.19830998601868313
iteration No is 116000 reward is -0.1983016256024522
iteration No is 117000 reward is -0.19829340809931925
iteration No is 118000 reward is -0.1982853298759004
iteration No is 119000 reward is -0.1982773874209424
iteration No is 120000 reward is -0.19826957734023365
iteration No is 121000 reward is -0.19826189635176805
iteration No is 122000 reward is -0.19825434128114616
iteration No is 123000 reward is -0.19824690905720102
iteration No is 124000 reward is -0.19823959670783567
iteration No is 125000 reward is -0.19823240135606016
iteration No is 126000 reward is -0.19822532021621758
iteration No is 127000 reward is -0.19821835059038825
iteration No is 128000 reward is -0.19821148986496254
iteration No is 129000 reward is -0.1982047355073729
iteration No is 130000 reward is -0.1981980850629769
iteration No is 131000 reward is -0.19819153615208315
iteration No is 132000 reward is -0.19818508646711203
iteration No is 133000 reward is -0.19817873376988485
iteration No is 134000 reward is -0.1981724758890342
iteration No is 135000 reward is -0.19816631071752946
iteration No is 136000 reward is -0.19816023621031156
iteration No is 137000 reward is -0.19815425038203116
iteration No is 138000 reward is -0.19814835130488526
iteration No is 139000 reward is -0.1981425371065472
iteration No is 140000 reward is -0.1981368059681854
iteration No is 141000 reward is -0.19813115612256632
iteration No is 142000 reward is -0.19812558585223766
iteration No is 143000 reward is -0.19812009348778772
iteration No is 144000 reward is -0.19811467740617736
iteration No is 145000 reward is -0.19810933602914094
iteration No is 146000 reward is -0.19810406782165296
iteration No is 147000 reward is -0.19809887129045733
iteration No is 148000 reward is -0.19809374498265622
iteration No is 149000 reward is -0.19808868748435582
iteration No is 150000 reward is -0.1980836974193661
iteration No is 151000 reward is -0.19807877344795238
iteration No is 152000 reward is -0.19807391426563623
iteration No is 153000 reward is -0.19806911860204315
iteration No is 154000 reward is -0.19806438521979544
iteration No is 155000 reward is -0.1980597129134477
iteration No is 156000 reward is -0.1980551005084634
iteration No is 157000 reward is -0.1980505468602305
iteration No is 158000 reward is -0.19804605085311444
iteration No is 159000 reward is -0.198041611399547
iteration No is 160000 reward is -0.1980372274391492
iteration No is 161000 reward is -0.19803289793788675
iteration No is 162000 reward is -0.1980286218872572
iteration No is 163000 reward is -0.19802439830350652
iteration No is 164000 reward is -0.19802022622687473
iteration No is 165000 reward is -0.1980161047208688
iteration No is 166000 reward is -0.19801203287156172
iteration No is 167000 reward is -0.19800800978691702
iteration No is 168000 reward is -0.1980040345961371
iteration No is 169000 reward is -0.1980001064490351
iteration No is 170000 reward is -0.1979962245154284
```

```
iteration No is 171000 reward is -0.19799238798455393
iteration No is 172000 reward is -0.1979885960645036
iteration No is 173000 reward is -0.19798484798167929
iteration No is 174000 reward is -0.19798114298026673
iteration No is 175000 reward is -0.19797748032172746
iteration No is 176000 reward is -0.19797385928430797
iteration No is 177000 reward is -0.19797027916256552
iteration No is 178000 reward is -0.19796673926691008
iteration No is 179000 reward is -0.1979632389231614
iteration No is 180000 reward is -0.19795977747212104
iteration No is 181000 reward is -0.19795635426915847
iteration No is 182000 reward is -0.19795296868381088
iteration No is 183000 reward is -0.19794962009939604
iteration No is 184000 reward is -0.19794630791263787
iteration No is 185000 reward is -0.19794303153330414
iteration No is 186000 reward is -0.1979397903838557
iteration No is 187000 reward is -0.19793658389910726
iteration No is 188000 reward is -0.1979334115258987
iteration No is 189000 reward is -0.19793027272277697
iteration No is 190000 reward is -0.19792716695968812
iteration No is 191000 reward is -0.19792409371767875
iteration No is 192000 reward is -0.19792105248860697
iteration No is 193000 reward is -0.19791804277486236
iteration No is 194000 reward is -0.19791506408909448
iteration No is 195000 reward is -0.19791211595394986
iteration No is 196000 reward is -0.19790919790181694
iteration No is 197000 reward is -0.19790630947457877
iteration No is 198000 reward is -0.1979034502233733
iteration No is 199000 reward is -0.19790061970836084
```

In [25]:

```
del ddpgLinkArm.iterations[0]
del ddpgLinkArm.return_reward[0]
plt.plot(ddpgLinkArm.iterations,ddpgLinkArm.return_reward, color='b'
);
plt.xlabel("iterations")
plt.ylabel("return_history")
plt.show()
```



In the previous case, we needed 500000 iterations to converge and reach the target in the 2DOF arm case. In this case the policy converged in 35000 iterations

In [1]:
```
import gym
import pybulletgym
import pybulletgym.envs
import numpy as np
import math
import matplotlib.pyplot as plt
import queue
import random
from collections import deque
```

In [2]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
```

In [3]:
```
env = gym.make("modified_gym_env:ReacherPyBulletEnv-v1", rand_init =
False)
env.reset()
```

```
current_dir=/home/apurba/.virtualenvs/276c_assgn/lib/python3.6/site-p
ackages/pybullet_envs/bullet
options=
```

Out[3]:
```
array([ 0.3928371 ,  0.3928371 , -0.68091764,  0.26561381,  0.5
,
        0.        ,  0.08333333,  0.        ])
```

In [25]:
```
class Actor(nn.Module) :
    def __init__(self, state_dim, action_dim, hidden_size_one, hidden
_size_two):

        super(Actor, self).__init__()
        self.input_size = state_dim;
        self.hidden_size_one = hidden_size_one;
        self.hidden_size_two = hidden_size_two;
        self.output_size = action_dim

        self.l1 = nn.Linear(self.input_size, self.hidden_size_one, bi
as = False)
        self.l2 = nn.Linear(self.hidden_size_one, self.hidden_size_tw
o, bias = False)
        self.l3 = nn.Linear(self.hidden_size_two, self.output_size, b
ias = False)

        self.model = torch.nn.Sequential(
                                    self.l1,
                                    nn.ReLU(),
                                    #nn.Tanh(),
                                    self.l2,
                                    nn.ReLU(),
                                    self.l3,
                                    nn.Tanh()
                                    )
        self.model.apply(self.weights_init_uniform)


    # takes in a module and applies the specified weight initializati
on
    def weights_init_uniform(self, m):
        classname = m.__class__.__name__
        # apply a uniform distribution to the weights and a bias=0
        if classname.find('Linear') != -1:
            m.weight.data.uniform_(-0.003, 0.003)
            #m.bias.data.fill_(0)

    def forward (self, state):



        return self.model(state)

class Critic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size_one, hidden
_size_two):

        super(Critic, self).__init__()
        self.input_size = (state_dim + action_dim);
        self.hidden_size_one = hidden_size_one;
        self.hidden_size_two = hidden_size_two;
        self.output_size = 1

        self.l1 = nn.Linear(self.input_size, self.hidden_size_one, bi
as = False)
```

```
        self.l2 = nn.Linear(self.hidden_size_one, self.hidden_size_tw
o, bias = False)
        self.l3 = nn.Linear(self.hidden_size_two, self.output_size, b
ias = False)
        self.model_one = torch.nn.Sequential(
                                    self.l1,
                                    nn.ReLU(),
                                    #nn.Tanh(),
                                    self.l2,
                                    nn.ReLU(),
                                    self.l3,
                                    nn.Tanh()
                                    )
        self.model_one.apply(self.weights_init_uniform)

        self.l4 = nn.Linear(self.input_size, self.hidden_size_one, bi
as = False)
        self.l5 = nn.Linear(self.hidden_size_one, self.hidden_size_tw
o, bias = False)
        self.l6 = nn.Linear(self.hidden_size_two, self.output_size, b
ias = False)
        self.model_two = torch.nn.Sequential(
                                    self.l4,
                                    nn.ReLU(),
                                    #nn.Tanh(),
                                    self.l5,
                                    nn.ReLU(),
                                    self.l6,
                                    nn.Tanh()
                                    )
        self.model_two.apply(self.weights_init_uniform)

    def weights_init_uniform(self, m):
        classname = m.__class__.__name__
        # apply a uniform distribution to the weights and a bias=0
        if classname.find('Linear') != -1:
            m.weight.data.uniform_(-0.0003, 0.0003)
            #m.bias.data.fill_(0)

    def Q1(self, state, action):
        stateAction = torch.cat([state, action], 1)
        return self.model_one(stateAction)

    def forward (self, state, action):

        stateAction = torch.cat([state, action], 1)
        return self.model_one(stateAction), self.model_two(stateActio
n)
```

In [26]:

```
class replayBuffer:
    def __init__(self, buffer_size):
        self.buffer_size = buffer_size;
        self.buffer = deque(maxlen = buffer_size)

    def push (self, state, action, next_state, reward, done):
        samples = (state, action, next_state, reward, done)
        self.buffer.append(samples)

    def sample(self, batch_size):
        state_batch = []
        action_batch = []
        next_state_batch = []
        reward_batch = []
        done_batch = []

        batch_data = random.sample(self.buffer, batch_size)

        for samples in batch_data:
            state, action, next_state, reward, done = samples
            state_batch.append(state)
            action_batch.append(action)
            reward_batch.append(reward)
            next_state_batch.append(next_state)
            done_batch.append(done)
        return (state_batch, action_batch, next_state_batch, reward_b
atch, done_batch)

    def __len__(self):
        return len(self.buffer)
```

In [33]:
```
class DDPG():
    def __init__(self,
                 env,
                 action_dim,
                 state_dim,
                 actor,
                 critic,
                 actor_target,
                 critic_target,
                 noise = 1,
                 d_param = 0.001,
                 critic_lr = 0.0003,
                 actor_lr = 0.0003,
                 gamma = 0.99, batch_size = 100, buffer_size = 10000,
pol_freq = 2):

        """
        param: env: An gym environment
        param: action_dim: Size of action space
        param: state_dim: Size of state space
        param: actor: actor model
        param: critic: critic model
        param: critic_lr: Learning rate of the critic
        param: actor_lr: Learning rate of the actor
        param: gamma: The discount factor
        param: batch_size: The batch size for training
        """


        self.env = env
        self.action_dim = action_dim
        self.state_dim = state_dim
        self.critic_lr = critic_lr
        self.actor_lr = actor_lr
        self.gamma = gamma
        self.batch_size = batch_size
        self.d = d_param
        self.noise = noise
        self.pol_freq = pol_freq

        self.actor = actor
        self.critic = critic
        self.actor_target = actor_target
        self.critic_target = critic_target

        self.actor_optimizer = optim.Adam(self.actor.parameters(), lr
= self.actor_lr)
        self.critic_optimizer = optim.Adam(self.critic.parameters(),
lr = self.critic_lr)

        self.iterations = []
        self.return_reward = []

        self.replay_buffer = replayBuffer(buffer_size)
        self.loss = nn.MSELoss()
```

```
    def updateQpolicy(self, batch_size, iteration):
        states, actions, state_next,rewards, _ = self.replay_buffer.s
ample(batch_size)
        states = torch.FloatTensor(states)
        actions = torch.FloatTensor(actions)
        rewards = torch.FloatTensor(rewards).reshape([batch_size,1])
        #rewards = torch.FloatTensor(rewards)
        state_next = torch.FloatTensor(state_next)
        #print("states size", states.size())
        Q_presone, Q_prestwo = self.critic.forward(states, actions)
        #Q_pres = critic.forward(states, actions)
        action_next = actor_target.forward(state_next)
        Q_nextone, Q_nexttwo = self.critic_target.forward(state_next,
action_next.detach())#while doing loss.backward we dont want target_p
olicy parameters to be updated
        #print("rewards", rewards.size())#100
        #print("q_next",Q_next.size())#100*1
        Q_nextchosen = torch.min(Q_nextone, Q_nexttwo)
        Q_nexttarget = rewards + Q_nextchosen * self.gamma
        #wrt Q parameter maps s and actions to theQ value
        #print("q_nexttaget",Q_nexttarget.size())#100*100
        #print("q_pres",Q_pres.size())#100*1
        criticLoss = self.loss(Q_nexttarget, Q_presone) + self.loss(Q
_nexttarget, Q_prestwo)



        #update the Q paramters which maps states to actions to the Q
value
        self.critic_optimizer.zero_grad();
        criticLoss.backward();
        self.critic_optimizer.step();

        if(iteration % self.pol_freq == 0):
            #wrt policy parameter, maps states to actions
            actorLoss = -1 * self.critic.Q1(states, actor.forward(sta
tes)).mean()

            #update thw policy parameters which updates the states to
actions
            self.actor_optimizer.zero_grad();
            actorLoss.backward();
            self.actor_optimizer.step();

            #update the target network weights with the original netw
ork weights
            for tar_param, src_param in zip(self.actor_target.paramet
ers(), self.actor.parameters()):
                tar_param.data.copy_(self.d * src_param.data + (1.0 -
self.d) * tar_param.data)

        for tar_param, src_param in zip(self.critic_target.parameters
(), self.critic.parameters()):
            tar_param.data.copy_(self.d * src_param.data + (1.0 - sel
f.d) * tar_param.data)

    def selectAction(self, state):
```

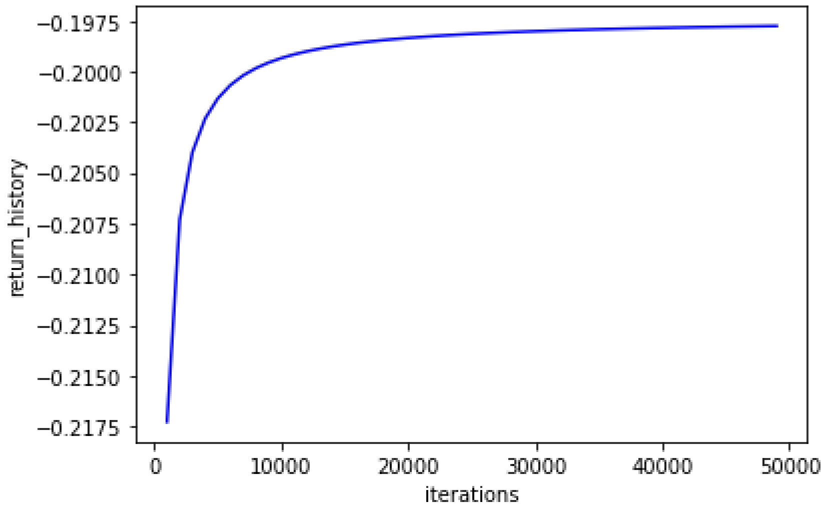
```
        #state = torch.FloatTensor(state)
        state = Variable(torch.from_numpy(state).float().unsqueeze(0
))
        action = self.actor.forward(state)
        action = action.detach().numpy()[0]
        return action


    def train(self, epochs):
        total_reward = 0
        for iterationNo in range(epochs):
            state = env.reset()
            batch_reward = 0

            steps = 0
            '''
            while(steps < self.batch_size):
                steps += 1
                action = self.selectAction(state)
                if(self.noise):
                    #mean = torch.zeros(2);
                    #variance = torch.diag([0.1, 0.1])
                    #c = MultivariateNormal(mean, variance)
                    #noise = c.sample()
                    noise = np.random.normal(0, 0.1)
                    action[0]+= noise
                    action[1]+= noise
                new_state, reward, done, _ = env.step(action)

                batch_reward += reward
                total_reward += reward
                self.replay_buffer.push(state, action, new_state, rew
ard, done)
                state = new_state



                if(done == True):
                    break;
            '''
            #fill up the buffer
            while(len(self.replay_buffer)< self.batch_size):
                action = self.selectAction(state)
                if(self.noise):
                    #mean = torch.zeros(2);
                    #variance = torch.diag([0.1, 0.1])
                    #c = MultivariateNormal(mean, variance)
                    #noise = c.sample()
                    noise = np.random.normal(0, 0.1)
                    action[0]+= noise
                    action[1]+= noise
                new_state, reward, done, _ = env.step(action)
                if(done == True):
                    state= env.reset()

                batch_reward += reward
                total_reward += reward
```

```
                self.replay_buffer.push(state, action, new_state, rew
ard, done)
                state = new_state

            #if(len(self.replay_buffer) >= self.batch_size):
            #if(iterationNo%self.batch_size == 0 and len(self.replay_
buffer)>= self.batch_size):
            action = self.selectAction(state)
            new_state, reward, done, _ = env.step(action)
            if(done == True):
                state = env.reset()
            batch_reward += reward
            total_reward += reward
            self.replay_buffer.push(state, action, new_state, reward,
done)
            state = new_state

            self.updateQpolicy(self.batch_size, iterationNo)
            if((iterationNo % 1000 == 0 and iterationNo!=0) or iterat
ionNo == 1):
                self.iterations.append(iterationNo)
                self.return_reward.append(total_reward/iterationNo)
                print("iteration No is", iterationNo, "reward is", to
tal_reward/iterationNo)
                #self.return_history.append(batch_reward)

            if(iterationNo%2000 == 0 and iterationNo!= 0):
                fileName = "model_td3"+str(iterationNo)
                torch.save(self.actor.state_dict(), fileName)
```

In [35]:
```
num_states = 8
num_actions = 2

actor = Actor(num_states, num_actions, 400, 300)
actor_target = Actor(num_states, num_actions, 400, 300)

critic = Critic(num_states, num_actions, 400, 300)
critic_target = Critic(num_states, num_actions, 400, 300)

for tar_param, src_param in zip(actor_target.parameters(), actor.parameters()):
    tar_param.data.copy_(src_param.data)

for tar_param, src_param in zip(critic_target.parameters(), critic.parameters()):
    tar_param.data.copy_(src_param.data)



#ddpgLinkArm = DDPG(env, num_actions, num_states, actor, critic, actor_target, critic_target, noise )
ddpgLinkArm = DDPG(env, num_actions, num_states, actor, critic, actor_target, critic_target)
ddpgLinkArm.train(50000)
```

```
iteration No is 1 reward is -20.128730691670775
iteration No is 1000 reward is -0.2172784238834863
iteration No is 2000 reward is -0.20730930080966267
iteration No is 3000 reward is -0.20398625978506435
iteration No is 4000 reward is -0.20232473927277222
iteration No is 5000 reward is -0.20132782696539694
iteration No is 6000 reward is -0.2006632187604801
iteration No is 7000 reward is -0.20018849861411092
iteration No is 8000 reward is -0.19983245850433404
iteration No is 9000 reward is -0.19955553841895202
iteration No is 10000 reward is -0.19933400235064638
iteration No is 11000 reward is -0.19915274556748724
iteration No is 12000 reward is -0.19900169824818795
iteration No is 13000 reward is -0.19887388897801164
iteration No is 14000 reward is -0.19876433817500336
iteration No is 15000 reward is -0.19866939414572954
iteration No is 16000 reward is -0.19858631812011493
iteration No is 17000 reward is -0.1985130157445726
iteration No is 18000 reward is -0.1984478580774239
iteration No is 19000 reward is -0.19838955911208034
iteration No is 20000 reward is -0.1983370900432711
iteration No is 21000 reward is -0.19828961802863418
iteration No is 22000 reward is -0.19824646165169155
iteration No is 23000 reward is -0.19820705800317867
iteration No is 24000 reward is -0.1981709379920419
iteration No is 25000 reward is -0.19813770758179605
iteration No is 26000 reward is -0.19810703335695373
iteration No is 27000 reward is -0.19807863129691455
iteration No is 28000 reward is -0.19805225795544962
iteration No is 29000 reward is -0.19802770346512016
iteration No is 30000 reward is -0.1980047859408127
iteration No is 31000 reward is -0.19798334696646053
iteration No is 32000 reward is -0.19796324792800538
iteration No is 33000 reward is -0.19794436701309298
iteration No is 34000 reward is -0.19792659674023425
iteration No is 35000 reward is -0.19790984191153885
iteration No is 36000 reward is -0.1978940179066599
iteration No is 37000 reward is -0.197879049253396
iteration No is 38000 reward is -0.1978648684239881
iteration No is 39000 reward is -0.19785141481660112
iteration No is 40000 reward is -0.19783863388958348
iteration No is 41000 reward is -0.19782647642242038
iteration No is 42000 reward is -0.19781489788226503
iteration No is 43000 reward is -0.1978038578788611
iteration No is 44000 reward is -0.19779331969379368
iteration No is 45000 reward is -0.1977832498725071
iteration No is 46000 reward is -0.19777361786953726
iteration No is 47000 reward is -0.19776439573903426
iteration No is 48000 reward is -0.19775555786396887
iteration No is 49000 reward is -0.197747080718498
```

In [37]:
```
del ddpgLinkArm.iterations[0]
del ddpgLinkArm.return_reward[0]
plt.plot(ddpgLinkArm.iterations,ddpgLinkArm.return_reward, color='b'
);
plt.xlabel("iterations")
plt.ylabel("return_history")
plt.show()
```



In the previous case, we needed 500000 iterations to converge and reach the target in the 2DOF arm case. In this case the policy converged in 20000 iterations

In [ ]:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
import time

import gym
import pybulletgym
import pybulletgym.envs
import numpy as np

from model import Actor


#env.render('human')
#env.reset()


for iterations in range(2000,20000,2000):

    modelActor = Actor(8,2,400,300)
    PATH = "model_td3"+str(4000)
    modelActor.load_state_dict(torch.load(PATH))


    env = gym.make("modified_gym_env:ReacherPyBulletEnv-v1", rand_ini
t=False)
    steps = 0
    env.render('human')
    state = env.reset()
    done = False
    while (steps<300 and done == False):
        state = Variable(torch.from_numpy(state).float().unsqueeze(0
))
        action = modelActor(state)
        action_np = action.detach().numpy()[0]
        state, r, done, info = env.step(action_np)
        steps+=1
        env.render('human')
        time.sleep(0.1)
```

```
current_dir=/home/apurba/.virtualenvs/276c_assgn/lib/python3.6/site-p
ackages/pybullet_envs/bullet
options=
```