Assignment 3 - Policy Gradient (DDPG and REINFORCE)modified

May 15, 2018

1 Assignment 3: Policy Gradients (DDPG and REINFORCE)

Name: Chuqiao Song
ID: A53239614

1.1 Background

This exercise requires you to solve various continous control problems in OpenAI-Gym.

DDPG is policy gradient actor critic method for continous control which is off policy. It tackles the curse of dimensionality / loss of performance faced when discretizing a continous action domain. DDPG uses similiar "tricks" as DQN to improve the stability of training, including a replay buffer and target networks.

Furthermore, you will implement REINFORCE for discrete and continous environments, and as a bonus compare the sample efficiency and performance with DQN and DDPG.

1.1.1 DDPG paper: https://arxiv.org/pdf/1509.02971.pdf

1.1.2 Environments:

InvertedPendulum-v2 environment:

Pendulum-v0 environment:

Halfcheetah-v2 environment:

1.1.3 Setup environment for Actor Critic

- inline plotting
- gym
- directory for logging videos

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        import random
        import math
```

```
#environment
import gym
import os
import time
#pytorch
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable

logging_interval = 20
animate_interval = logging_interval * 5
logdir='./DDPG/'
```

1.1.4 Set up gym environment

The code below does the following for you: - Wrap environment, log videos, setup CUDA variables (if GPU is available) - Record action and observation space dimensions - Fix random seed for determinisitic training

```
In [2]: VISUALIZE = True
        SEED = 0
        MAX_PATH_LENGTH = 500
        NUM_EPISODES = 12000
        GAMMA=0.99
        BATCH_SIZE = 128

        # Environments to be tested on
        # env_name = 'InvertedPendulum-v1'
        # env_name = 'Pendulum-v0'
        env_name = 'HalfCheetah-v1'

        # wrap gym to save videos
        env = gym.make(env_name)
        if VISUALIZE:
            if not os.path.exists(logdir):
                os.mkdir(logdir)
            env = gym.wrappers.Monitor(env, logdir, force=True, video_callable=lambda episode_i
        env._max_episodes_steps = MAX_PATH_LENGTH

        # check observation and action space
        discrete = isinstance(env.action_space, gym.spaces.Discrete)
        obs_dim = env.observation_space.shape[0]
        act_dim = env.action_space.n if discrete else env.action_space.shape[0]

        if discrete:
```

```
        print("This is a discrete action space, probably not the right algorithm to use")

    # set random seeds
    torch.manual_seed(SEED)
    np.random.seed(SEED)

    # make variable types for automatic setting to GPU or CPU, depending on GPU availabili
    use_cuda = torch.cuda.is_available()
    FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
    LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
    ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
    Tensor = FloatTensor
```

```
[2018-05-15 18:15:02,522] Making new env: HalfCheetah-v1
[2018-05-15 18:15:02,888] Clearing 4 monitor files from previous run (because force=True was pr
```

1.1.5 Demonstrate your understanding of the simulation:

For the environments mentioned above ('Pendulum-v0', 'HalfCheetah-v2', 'InvertedPendulum-v2'), - describe the reward system - describe the each state variable (observation space) - describe the action space - when is the environment considered "solved"?

Ans:

Pendulum-v0 - reward = -(theta^2 + 0.1*theta_dt^2 + 0.001*action^2) - x1 = cos(theta) within [-1,1]; x2 = sin(theta) within [-1,1]; x3 = theta_dot within [-8,8] where theta is the angular of the pendulum, and theta_dot is the angular velocity of the pendulum. - A joint effore within [-2,2] - In general, the pendulum starts randomly at angle from -pi to pi, with random velocity between -1 and 1. There is no specific termination, so I have to manual set up a maximum number of steps for one episode. Also, to optimize this model, we want the pendulum to remain at zero angle (vertical), with least angular velocity as well as the least effort.

HalfCheetah-v1 - To produce reward in each step, there are two ingredients, one is called reward_control and the other called reward_run. reward_ctrl = - 0.1 * np.square(action).sum(), where action has 6 dimensions, reward_run = (xposafter - xposbefore)/self.dt. Therefore, the reward = reward_ctrl + reward_run which is a scalar.

- State space

```
   (name)    (joint)     (parameter)
 - rootx     slider      position (m)
 - rootz     slider      position (m)
 - rooty     hinge       angle (rad)
 - bthigh    hinge       angle (rad)
 - bshin     hinge       angle (rad)
 - bfoot     hinge       angle (rad)
 - fthigh    hinge       angle (rad)
 - fshin     hinge       angle (rad)
 - ffoot     hinge       angle (rad)
 - rootx     slider      velocity (m/s)
```

```
- rootz     slider      velocity (m/s)
- rooty     hinge       angular velocity (rad/s)
- bthigh    hinge       angular velocity (rad/s)
- bshin     hinge       angular velocity (rad/s)
- bfoot     hinge       angular velocity (rad/s)
- fthigh    hinge       angular velocity (rad/s)
- fshin     hinge       angular velocity (rad/s)
- ffoot     hinge       angular velocity (rad/s)
```

- Action space

```
  (name)   (actuator)   (parameter):
- bthigh    hinge       torque (N m)
- bshin     hinge       torque (N m)
- bfoot     hinge       torque (N m)
- fthigh    hinge       torque (N m)
- fshin     hinge       torque (N m)
- ffoot     hinge       torque (N m)
```

- When the total reward with in 500 steps for one episode greater than 1500, we consider the cheetah is being well trained.

InvertedPendulum-v1 - reward is 1 for each step - x1 = cart position, x2 = pole position, x3 = cart velocity, x4 = pole angular velocity - a1 = the force impose on cart - when the average reward greater than 500, we consider the inverted pendulum is being well balanced

1.1.6 Implement an action normalization class:

To train across various environments, it is useful to normalize action inputs and outputs between [-1, 1]. This class should take in actions and implement forward and reverse functions to map actions between [-1, 1] and [action_space.low, action_space.high].

Using the following gym wrapper, implement this class. - https://github.com/openai/gym/blob/78c416ef7bc829ce55b404b6604641ba0cf47d10/gym/core.py - i.e. we are overriding the outputs scale of actions.

```
In [4]: class NormalizeAction(gym.ActionWrapper):
            def action(self, act):
                # [-1, 1] => [action_space.low, action_space.high]
                #tanh outputs (-1,1) from tanh, need to be [action_space.low, action_space.high]
                act = (act + 1)/2  #[-1, 1] => [0,1]
                act = act * (self.action_space.high - self.action_space.low)
                act = act + self.action_space.low
                return act

            def reverse_action(self, act):
                # [action_space.low, action_space.high] => [-1,1]
                #reverse of that above
                act = act - self.action_space.low
```

```
        act = act / (self.action_space.high - self.action_space.low)
        act = act * 2 - 1
        return act
```

2 DDPG

2.0.1 Write a weight syncing function

In contrast to DQN, DDPG uses soft weight sychronization. At each time step following training, the actor and critic target network weights are updated to track the rollout networks. - target_network.weights <= target_network.weights * (1 - tau) + source_network.weights * (tau)

```
In [5]: def weightSync(target_model, source_model, tau = 0.001):
            # soft update
            for parameter_target, parameter_source in zip(target_model.parameters(), source_mod
                parameter_target.data.copy_((1 - tau) * parameter_target.data + tau * paramete
```

2.0.2 Write a Replay class that includes all the functionality of a replay buffer

DDPG is an off policy actor-critic method and an identical replay buffer to that used for the previous assignment is applicable here as well (do not include the generate_minibatch method in your Replay class this time). Like before, your constructor for Replay should create an initial buffer of size 1000 when you instantiate it.

The replay buffer should kept to some maximum size (60000), allow adding of samples and returning of samples at random from the buffer. Each sample (or experience) is formed as (state, action, reward, next_state, done).

```
In [6]: class Replay():
            def __init__(self):
                self.capacity = 60000
                self.memory = []
                self.position = 0
                self.gamma = 0.99

            def initialize(self, init_length, envir):
                st = envir.reset()
                for _ in range(init_length):
                    a = envir.action_space.sample()
                    st1, r, done, info = envir.step(a)
                    # normalizing action
                    # [action_space.low, action_space.high] => [-1,1]
                    a = envir.reverse_action(a)
                    self.push((st, a, st1, r, done))
                    if done: st = envir.reset()
                    else : st = st1

            def push(self, transition):
                if len(self.memory) < self.capacity:
```

```
            self.memory.append(None)
        self.memory[self.position] = transition
        self.position = (self.position + 1) % self.capacity

    def generateMinibatch(self, batch_size):
        batch_memory = random.sample(self.memory, batch_size) #return a list
        batch_memory = list(zip(*batch_memory))

        batch_st = Variable(FloatTensor(batch_memory[0]))
        batch_at = Variable(FloatTensor(batch_memory[1]))
        batch_st1 = Variable(FloatTensor(batch_memory[2]))
        batch_r = Variable(torch.unsqueeze(FloatTensor(batch_memory[3]),1))
        batch_done = torch.unsqueeze(FloatTensor(batch_memory[4]),1)

        return batch_st, batch_at, batch_st1, batch_r, batch_done

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

2.0.3 Write an Ornstein Uhlenbeck process class for exploration noise

The proccess is described here: - https://en.wikipedia.org/wiki/Ornstein–Uhlenbeck_process - http://math.stackexchange.com/questions/1287634/implementing-ornstein-uhlenbeck-in-matlab

You should implement: - a step / sample method - reset method

Use theta = 0.15, mu = 0, sigma = 0.3, dt = 0.01

In [8]:
```
class OrnsteinUhlenbeckProcess():
#     def __init__(self, mu=np.zeros(act_dim), sigma=0.3, theta=.15, dimension=1e-2, x
    # for inverted pendulum and pendulum above
    def __init__(self, mu=np.zeros(act_dim), sigma=0.05, theta=.25, dimension=1e-2, x0=
        self.theta = theta
        self.mu = mu
        self.sigma = sigma
        self.dt = dimension
        self.x0 = x0
        self.reset()

    def step(self):
        x = self.x_prev + self.theta * (self.mu - self.x_prev) * self.dt + \
                self.sigma * np.sqrt(self.dt) * np.random.normal(size=self.mu.shape)
        self.x_prev = x
        return x

    def reset(self):
        self.x_prev = self.x0 if self.x0 is not None else np.zeros_like(self.mu)

    def __repr__(self):
```

```python
        return 'OrnsteinUhlenbeckActionNoise(mu={}, sigma={})'.format(self.mu, self.si
```

2.0.4 Write a Deep Neural Network class that creates a dense network of a desired architecture for actor and critic networks

Actor

- input and hidden layer activation function: ReLU
- output activation function: Tanh
- hidden_state sizes: 400
- state and action sizes: variable
- number of hidden layers: 2
- batch normalization applied to all hidden layers
- weight initialization: normal distribution with small variance.

Critic

- input and hidden layer activation function: ReLU
- output activation function: None
- hidden_state sizes: 300, 300 + action size
- state and action sizes: variable
- number of hidden layers: 2
- batch normalization applied to all hidden layers prior to the action input
- weight initialization: normal distribution with small variance.

Good baselines can be found in the paper.

In [9]:
```python
# ----------------------------------------------------
# actor model, MLP
# ----------------------------------------------------
# 2 hidden layers, 400 units per layer, tanh output to bound outputs between -1 and 1

class actor(nn.Module):
    def __init__(self, input_size, output_size):
        super(actor, self).__init__()
        self.fc1 = nn.Linear(input_size, 400)
        self.bn1 = nn.BatchNorm1d(400) # batchnormalization
        self.fc2 = nn.Linear(400, 400)
        self.bn2 = nn.BatchNorm1d(400) # batchnormalization
        self.fc3 = nn.Linear(400, output_size)
```

```
        # parameters initialization
#         nn.init.xavier_normal_(self.fc1.weight)
#         nn.init.xavier_normal_(self.fc2.weight)
#         nn.init.xavier_normal_(self.fc3.weight)
#         nn.init.normal_(self.fc1.bias)
#         nn.init.normal_(self.fc2.bias)
#         nn.init.normal_(self.fc3.bias)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.bn1(x)  # turn off for inverted-pendulum -v1
        x = F.relu(self.fc2(x))
        x = self.bn2(x)  # turn off for inverted-pendulum -v1

        outputs = F.tanh(self.fc3(x))
        return outputs


# ----------------------------------------------------
# critic model, MLP
# ----------------------------------------------------
# 2 hidden layers, 300 units per layer, ouputs rewards therefore unbounded
# Action not to be included until 2nd layer of critic (from paper). Make sure to formu

class critic(nn.Module):
    def __init__(self, state_size, action_size, output_size):
        super(critic, self).__init__()

        self.fc1 = nn.Linear(state_size, 300)
        self.bn1 = nn.BatchNorm1d(300) # batchnormalization
        self.fc2 = nn.Linear(300 + action_size, 300)
        self.fc3 = nn.Linear(300, output_size)

        # parameters initialization
#         nn.init.xavier_normal_(self.fc1.weight)
#         nn.init.xavier_normal_(self.fc2.weight)
#         nn.init.xavier_normal_(self.fc3.weight)
#         nn.init.normal_(self.fc1.bias)
#         nn.init.normal_(self.fc2.bias)
#         nn.init.normal_(self.fc3.bias)

    def forward(self, states, actions):
        x = F.relu(self.fc1(states))
        x = self.bn1(x)  # turn off for inverted-pendulum -v1
        x = torch.cat((x, actions), 1) # actions only join at second layer
        x = F.relu(self.fc2(x))
```

```
            outputs = self.fc3(x)
            return outputs
```

2.0.5 Define DDPG class to encapsulate definition, rollouts, and training

- gamma = 0.99
- actor_lr = 1e-4
- critic_lr = 1e-3
- critic l2 regularization = 1e-2
- noise decay
- noise class
- batch_size = 128
- optimizer: Adam
- loss (critic): mse

Furthermore, you can experiment with action versus parameter space noise. The standard implimentation works with action space noise, howeve parameter space noise has shown to produce excellent results.

```
In [10]: class DDPG:
             def __init__(self, obs_dim, act_dim, critic_lr = 1e-3, actor_lr = 1e-4, gamma = GA
                 self.gamma = GAMMA
                 self.batch_size = BATCH_SIZE

                 # actor
                 self.actor = actor(input_size = obs_dim, output_size = act_dim).type(FloatTens
                 self.actor_target = actor(input_size = obs_dim, output_size = act_dim).type(Fl
                 self.actor_target.load_state_dict(self.actor.state_dict())

                 # critic
                 self.critic = critic(state_size = obs_dim, action_size = act_dim, output_size
                 self.critic_target = critic(state_size = obs_dim, action_size = act_dim, outpu
                 self.critic_target.load_state_dict(self.critic.state_dict())

                 # optimizers
                 self.optimizer_actor = torch.optim.Adam(self.actor.parameters(), lr = actor_lr
                 self.optimizer_critic = torch.optim.Adam(self.critic.parameters(), lr = critic

                 # critic loss
                 self.critic_loss = nn.MSELoss()

                 # noise
                 self.noise = OrnsteinUhlenbeckProcess(dimension = act_dim, num_steps = NUM_EPI
```

```
    # replay buffer
    self.replayBuffer = Replay()


def train(self):
    # sample from Replay
    b_st, b_at, b_st1, b_r, b_d = self.replayBuffer.generateMinibatch(self.batch_s

    ## update critic (create target for Q function)
    # below is for target actor network
    targetActorPredict_b_at1 = self.actor_target(b_st1)

    #below is for target critic network
    mask = 1 - b_d   # if done is true, change the target to just reward
    batch_Q_next = self.critic_target(b_st1, targetActorPredict_b_at1)
    QQ_next = Variable((batch_Q_next.data * mask).view(self.batch_size, 1))
    b_Q_critic_target = b_r + self.gamma*(QQ_next)


    # below is for behavior critic network
    b_Q_critic_behaviorQ = self.critic(b_st, b_at)

    ## critic optimizer and backprop step (feed in target and predicted values to
    critic_loss = self.critic_loss(b_Q_critic_behaviorQ, b_Q_critic_target.detach
    self.optimizer_critic.zero_grad()
    critic_loss.backward()
    self.optimizer_critic.step()

    ## update actor (formulate the loss wrt which actor is updated)
    # below is for behavior actor network
    b_at_actor_behavior = self.actor(b_st)
    # below is for behavior critic network
    b_Q_critic_behaviorP = self.critic(b_st, b_at_actor_behavior)

    ## actor optimizer and backprop step (loss_actor.backward())
    loss_actor = -1. * b_Q_critic_behaviorP
    loss_actor = loss_actor.mean()

    self.optimizer_actor.zero_grad()
    loss_actor.backward()
    self.optimizer_actor.step()

    # sychronize target network with fast moving one
    weightSync(self.critic_target, self.critic)
    weightSync(self.actor_target, self.actor)
```

2.0.6 Create an instance of your DDPG object

- Print network architectures, confirm they are correct

```
In [11]: ddpg = DDPG(obs_dim = obs_dim, act_dim = act_dim)
         print(ddpg.actor)
         print(ddpg.critic)
```

```
actor(
  (fc1): Linear(in_features=17, out_features=400, bias=True)
  (bn1): BatchNorm1d(400, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=400, out_features=400, bias=True)
  (bn2): BatchNorm1d(400, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc3): Linear(in_features=400, out_features=6, bias=True)
)
critic(
  (fc1): Linear(in_features=17, out_features=300, bias=True)
  (bn1): BatchNorm1d(300, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=306, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=1, bias=True)
)
```

2.0.7 Train DDPG on different environments

Early stopping conditions: - avg_val > 500 for "InvertedPendulum" - avg_val > -150 for "Pendulum" - avg_val > 1500 for "HalfCheetah"

```
In [12]: env = NormalizeAction(env) # remap action values for the environment
         avg_val = 0

         #for plotting
         running_rewards_ddpg = []
         step_list_ddpg = []
         step_counter = 0

         # set term_condition for early stopping according to environment being used
         # term_condition = -150 # Pendulum
         # term_condition = 500 # inverted pendulum
         term_condition = 1500 # halfcheetah
         ddpg.replayBuffer.initialize(1000, env)

         for itr in range(NUM_EPISODES):
             state = env.reset() # get initial state
             animate_this_episode = (itr % animate_interval == 0) and VISUALIZE
             total_reward = 0
             while True: # for each episode, we loop each step in this episode
                 ddpg.noise.reset()
                 if animate_this_episode:
```

```
                env.render()
                time.sleep(0.05)
            # use actor to get action, add ddpg.noise.step() to action
            # remember to put NN in eval mode while testing (to deal with BatchNorm layer
            # to train mode after you're done getting the action
            var_state = Variable(torch.unsqueeze(FloatTensor(state),0), requires_grad=Fals
            ddpg.actor.eval()
            cuda_tensor_action = ddpg.actor(var_state)
            ddpg.actor.train()

            action = cuda_tensor_action.data[0].cpu().numpy()
            action = action + ddpg.noise.step()
            # below already include [-1,1] => [action_space.low, action_space.high]
            new_state, reward, done, _ = env.step(action)
            total_reward += reward

            ddpg.replayBuffer.push((state, action, new_state, reward, done))
            # step action, get next state, reward, done (keep track of total_reward)
            # populate ddpg.replayBuffer
            ddpg.train() ###################### update network (per step) in one episode
            step_counter += 1
            state = new_state
            if done: break

        if avg_val > term_condition and itr >100 : break

        running_rewards_ddpg.append(total_reward) # return of this episode
        step_list_ddpg.append(step_counter)

        avg_val = avg_val * 0.95 + 0.05*running_rewards_ddpg[-1]
        print("Average value: {} for episode: {}".format(avg_val,itr))

    print('Done')
```

```
[2018-05-15 18:16:02,781] Starting new video recorder writing to /datasets/home/85/185/chs140/
[2018-05-15 18:16:02,784] GLFW error: 65544, desc: b'X11: RandR gamma ramp support seems broke
[2018-05-15 18:16:02,825] GLFW error: 65544, desc: b'Linux: Failed to watch for joystick conne
[2018-05-15 18:16:02,826] GLFW error: 65544, desc: b'Linux: Failed to open joystick device dir


Average value: -17.843247096416302 for episode: 0
Average value: -36.08468524747259 for episode: 1
Average value: -54.33420902731596 for episode: 2
Average value: -71.70057412910461 for episode: 3
Average value: -59.74418617579752 for episode: 4
Average value: -62.580942535612834 for episode: 5
Average value: -73.82833487779708 for episode: 6
Average value: -81.68708376415621 for episode: 7
```

```
Average value: -89.20755442592052 for episode: 8
Average value: -97.05297431747917 for episode: 9
Average value: -105.00531571765299 for episode: 10
Average value: -112.9256708127256 for episode: 11
Average value: -114.88980914193526 for episode: 12
Average value: -113.91333299704293 for episode: 13
Average value: -108.82201041270366 for episode: 14
Average value: -102.4959228355163 for episode: 15
Average value: -87.32240365216148 for episode: 16

[2018-05-15 18:20:02,707] Starting new video recorder writing to /datasets/home/85/185/chs140/l

Average value: -82.14447466813796 for episode: 17
Average value: -72.65859360094848 for episode: 18
Average value: -62.916032997105276 for episode: 19
Average value: -50.68419508170968 for episode: 20
Average value: -23.730825868157385 for episode: 21
Average value: 3.33701179927807 for episode: 22
Average value: 37.51692646332775 for episode: 23
Average value: 77.393253713093 for episode: 24
Average value: 105.84814635585346 for episode: 25
Average value: 153.6958216598512 for episode: 26
Average value: 199.96461066785744 for episode: 27
Average value: 235.49672492374873 for episode: 28
Average value: 276.5310388019493 for episode: 29
Average value: 327.32799303986496 for episode: 30
Average value: 362.99732955705167 for episode: 31
Average value: 408.6795544487022 for episode: 32
Average value: 464.0515609979857 for episode: 33
Average value: 518.8006462658099 for episode: 34
Average value: 571.8692505307743 for episode: 35
Average value: 625.817907880608 for episode: 36

[2018-05-15 18:23:11,418] Starting new video recorder writing to /datasets/home/85/185/chs140/l

Average value: 671.2171233547612 for episode: 37
Average value: 699.5765734889196 for episode: 38
Average value: 741.2872792774488 for episode: 39
Average value: 784.2766717970711 for episode: 40
Average value: 757.9701989956791 for episode: 41
Average value: 743.1667150743076 for episode: 42
Average value: 731.5552601616956 for episode: 43
Average value: 762.155545193832 for episode: 44
Average value: 794.1670752052572 for episode: 45
Average value: 833.5056208380971 for episode: 46
```

```
Average value: 873.9512387038728 for episode: 47
Average value: 890.4033359383803 for episode: 48
Average value: 933.8845001740074 for episode: 49
Average value: 977.3198761357222 for episode: 50
Average value: 1011.761068478279 for episode: 51
Average value: 1046.2865188527262 for episode: 52
Average value: 1084.5085816534997 for episode: 53
Average value: 1123.2315669012737 for episode: 54
Average value: 1154.5983312170194 for episode: 55
Average value: 1183.100673613629 for episode: 56

[2018-05-15 18:26:24,254] Starting new video recorder writing to /datasets/home/85/185/chs140/

Average value: 1213.4982745261248 for episode: 57
Average value: 1246.1704659024365 for episode: 58
Average value: 1276.568013502378 for episode: 59
Average value: 1307.3044995795185 for episode: 60
Average value: 1335.7459614607842 for episode: 61
Average value: 1362.6491562446 for episode: 62
Average value: 1389.2224576939045 for episode: 63
Average value: 1423.4640086873464 for episode: 64
Average value: 1455.8337191432397 for episode: 65
Average value: 1479.3046715230057 for episode: 66
Average value: 1506.4469132844429 for episode: 67
Average value: 1533.1042491466683 for episode: 68
Average value: 1560.0639126659487 for episode: 69
Average value: 1579.5877185930685 for episode: 70
Average value: 1602.220868005994 for episode: 71
Average value: 1625.9271730181622 for episode: 72
Average value: 1650.7482825115624 for episode: 73
Average value: 1673.5260637469962 for episode: 74
Average value: 1694.709390414398 for episode: 75
Average value: 1713.271310457521 for episode: 76

[2018-05-15 18:29:38,795] Starting new video recorder writing to /datasets/home/85/185/chs140/

Average value: 1732.0232539771855 for episode: 77
Average value: 1746.449897059003 for episode: 78
Average value: 1766.8924265983137 for episode: 79
Average value: 1791.1038966048711 for episode: 80
Average value: 1800.5436560373835 for episode: 81
Average value: 1823.7681636216178 for episode: 82
Average value: 1845.3434709124267 for episode: 83
Average value: 1860.894163645346 for episode: 84
Average value: 1879.2853836295815 for episode: 85
```

```
Average value: 1899.9127730848775 for episode: 86
Average value: 1915.939940025595 for episode: 87
Average value: 1943.2840037322642 for episode: 88
Average value: 1952.1476900849407 for episode: 89
Average value: 1949.8644110067514 for episode: 90
Average value: 1970.982376402673 for episode: 91
Average value: 1994.1006107290766 for episode: 92
Average value: 2013.2643065128843 for episode: 93
Average value: 2026.466342157543 for episode: 94
Average value: 2043.7369095249148 for episode: 95
Average value: 2054.986204252428 for episode: 96
```

```
[2018-05-15 18:32:50,700] Starting new video recorder writing to /datasets/home/85/185/chs140/
```

```
Average value: 2067.7772022385916 for episode: 97
Average value: 2073.6238591946585 for episode: 98
Average value: 2084.5534645084 for episode: 99
Average value: 2086.6673680545146 for episode: 100
Done
```

2.1 Plot rewards over multiple training runs

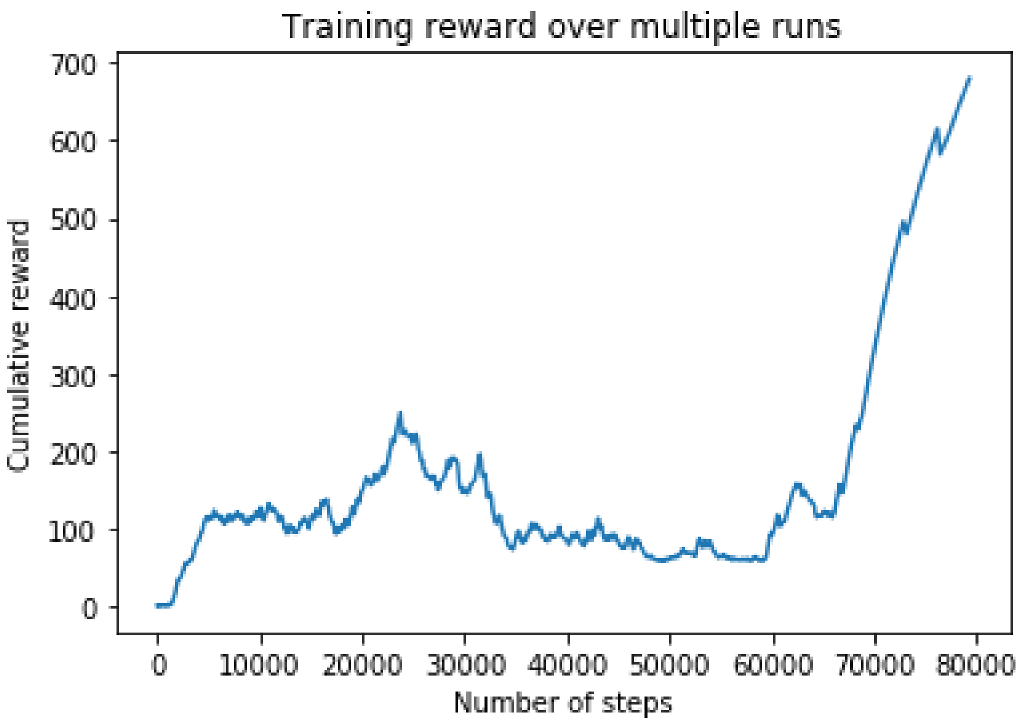
This is provided to generate and plot results for you.

```
In [15]: def numpy_ewma_vectorized_v2(data, window):
             alpha = 2 /(window + 1.0)
             alpha_rev = 1-alpha
             n = data.shape[0]

             pows = alpha_rev**(np.arange(n+1))

             scale_arr = 1/pows[:-1]
             offset = data[0]*pows[1:]
             pw0 = alpha*alpha_rev**(n-1)

             mult = data*pw0*scale_arr
             cumsums = mult.cumsum()
             out = offset + cumsums*scale_arr[::-1]
             return out
```

3 DDPG Inverted-Pendulum-v1

```
In [13]: np.save('ddpg_inverted_pendulum_reward.npy', running_rewards_ddpg)
         np.save('ddpg_inverted_pendulum_step.npy', step_list_ddpg)
```

In [14]:
```
plt.figure()
out = numpy_ewma_vectorized_v2(np.array(running_rewards_ddpg),20)
step_list_ddpg = np.array(step_list_ddpg)
plt.plot(step_list_ddpg, out)
plt.title('Training reward over multiple runs')
plt.xlabel('Number of steps')
plt.ylabel('Cumulative reward')
plt.show()
```



In [19]:
```
# env = NormalizeAction(env) # remap action values for the environment
state = env.reset() # get initial state
while True: # for each episode, we loop each step in this episode
    ddpg.noise.reset()
    env.render()
    time.sleep(0.05)
    # use actor to get action, add ddpg.noise.step() to action
    # remember to put NN in eval mode while testing (to deal with BatchNorm layers) a
    # to train mode after you're done getting the action
    var_state = Variable(torch.unsqueeze(FloatTensor(state),0), requires_grad=False)

    ddpg.actor.eval()
    cuda_tensor_action = ddpg.actor(var_state)
    action = cuda_tensor_action.data[0].cpu().numpy()
    action = action + ddpg.noise.step()
```

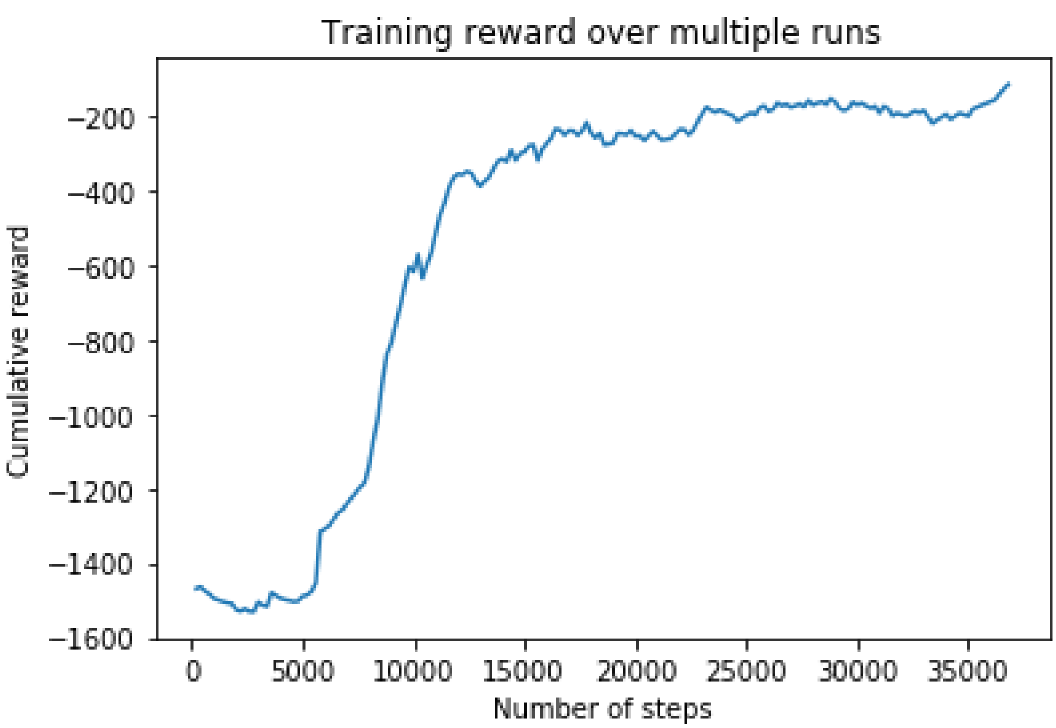
```
            # below already include [-1,1] => [action_space.low, action_space.high]
            new_state, reward, done, _ = env.step(action)
            # step action, get next state, reward, done (keep track of total_reward)
            # populate ddpg.replayBuffer
            state = new_state
            if done: break

        print('done')
```

```
done
```

4 DDPG Pendulum-v0

In [14]:
```
np.save('ddpg_pendulum_reward.npy', running_rewards_ddpg)
np.save('ddpg_pendulum_step.npy', step_list_ddpg)
```

In [16]:
```
plt.figure()
out = numpy_ewma_vectorized_v2(np.array(running_rewards_ddpg),20)
step_list_ddpg = np.array(step_list_ddpg)
plt.plot(step_list_ddpg, out)
plt.title('Training reward over multiple runs')
plt.xlabel('Number of steps')
plt.ylabel('Cumulative reward')
plt.show()
```



```
In [19]: # env = NormalizeAction(env) # remap action values for the environment
         state = env.reset() # get initial state
         while True: # for each episode, we loop each step in this episode
             ddpg.noise.reset()
             env.render()
             time.sleep(0.05)
             # use actor to get action, add ddpg.noise.step() to action
             # remember to put NN in eval mode while testing (to deal with BatchNorm layers) a
             # to train mode after you're done getting the action
             var_state = Variable(torch.unsqueeze(FloatTensor(state),0), requires_grad=False)

             ddpg.actor.eval()
             cuda_tensor_action = ddpg.actor(var_state)
             action = cuda_tensor_action.data[0].cpu().numpy()
             action = action + ddpg.noise.step()
             # below already include [-1,1] => [action_space.low, action_space.high]
             new_state, reward, done, _ = env.step(action)
             # step action, get next state, reward, done (keep track of total_reward)
             # populate ddpg.replayBuffer
             state = new_state
             if done: break

         print('done')
```

```
done
```

5 DDPG HalfCheetah-v1

```
In [13]: np.save('ddpg_halfcheetah_reward.npy', running_rewards_ddpg)
         np.save('ddpg_halfcheetah_step.npy', step_list_ddpg)
```

```
In [16]: plt.figure()
         out = numpy_ewma_vectorized_v2(np.array(running_rewards_ddpg),20)
         step_list_ddpg = np.array(step_list_ddpg)
         plt.plot(step_list_ddpg, out)
         plt.title('Training reward over multiple runs')
         plt.xlabel('Number of steps')
         plt.ylabel('Cumulative reward')
         plt.show()
```

In [ ]:
```
# env = NormalizeAction(env) # remap action values for the environment
state = env.reset() # get initial state
while True: # for each episode, we loop each step in this episode
    ddpg.noise.reset()
    env.render()
    time.sleep(0.05)
    # use actor to get action, add ddpg.noise.step() to action
    # remember to put NN in eval mode while testing (to deal with BatchNorm layers) an
    # to train mode after you're done getting the action
    var_state = Variable(torch.unsqueeze(FloatTensor(state),0), requires_grad=False)

    ddpg.actor.eval()
    cuda_tensor_action = ddpg.actor(var_state)
    action = cuda_tensor_action.data[0].cpu().numpy()
    action = action + ddpg.noise.step()
    # below already include [-1,1] => [action_space.low, action_space.high]
    new_state, reward, done, _ = env.step(action)
    # step action, get next state, reward, done (keep track of total_reward)
    # populate ddpg.replayBuffer
    state = new_state
    if done: break

print('done')
```

6 REINFORCE

In this section you will implement REINFORCE, with modifications for batch training. It will be for use on both discrete and continous action spaces.

6.1 Policy Parametrization

Define a MLP which outputs a distribution over the action preferences given input state. For the discrete case, the MLP outputs the likelihood of each action (softmax) while for the continuous case, the output is the mean and standard deviation parametrizing the normal distribution from which the action is sampled.

In [3]:
```
# -----------------------------------------------------
# Policy parametrizing model, MLP
# -----------------------------------------------------
# 1 or 2 hidden layers with a small number of units per layer (similar to DQN)
# use ReLU for hidden layer activations
# softmax as activation for output if discrete actions, linear for continuous control
# for the continuous case, output_dim=2*act_dim (each act_dim gets a mean and std_dev)

class mlp(nn.Module):
    # For discrete, it is the number of actions for outputs
    # For continuous, it is the dimension of action
    def __init__(self, Dim_state, num_outputs, disct):
        super(mlp, self).__init__()
        self.disct = disct
        if self.disct == True:
            self.fc1 = nn.Linear(Dim_state, 50)
            self.fc2 = nn.Linear(50, 50)
            self.fc3 = nn.Linear(50, num_outputs)
            # parameters initialization
#             nn.init.xavier_normal_(self.fc1.weight)
#             nn.init.xavier_normal_(self.fc2.weight)
#             nn.init.xavier_normal_(self.fc3.weight)
#             nn.init.normal_(self.fc1.bias)
#             nn.init.normal_(self.fc2.bias)
#             nn.init.normal_(self.fc3.bias)

        else:
            self.fc1 = nn.Linear(Dim_state, 50)
            self.fc2 = nn.Linear(50, 50)
            self.fc_mu = nn.Linear(50, num_outputs)
            self.fc_sigma = nn.Linear(50, num_outputs)


            # parameters initialization
#             nn.init.xavier_normal_(self.fc1.weight)
#             nn.init.xavier_normal_(self.fc2.weight)
```

```
#               nn.init.xavier_normal_(self.fc_mu.weight)
#               nn.init.xavier_normal_(self.fc_sigma.weight)

#               nn.init.normal_(self.fc1.bias)
#               nn.init.normal_(self.fc2.bias)

#               nn.init.normal_(self.fc_mu.bias)
#               nn.init.normal_(self.fc_sigma.bias)


        def forward(self, x):
            if self.disct == True:
                x = F.relu(self.fc1(x))
                x = F.relu(self.fc2(x))
                x = self.fc3(x)
                actions_prob = F.softmax(x, dim=1)
                return actions_prob
            else:
                x = F.relu(self.fc1(x))
                x = F.relu(self.fc2(x))

                mu = self.fc_mu(x)
                sigma = self.fc_sigma(x)
                return [mu, sigma]
```

Define a function that samples an action from the policy distribtion parameters obtained as output of the MLP. The function should return the action and the log-probability (log_odds) of taking that action.

```
In [4]: def sample_action(logit, disct):
            # logit is the output of the softmax/linear layer
            # discrete is a flag for the environment type
            # Hint: use Categorical and Normal from torch.distributions to sample action and g
            # Note that log_probability in this case translates to ln(\pi(a|s))
            if disct == True:
                action_distribution=torch.distributions.Categorical(logit)
                action = action_distribution.sample()
                log_odds = action_distribution.log_prob(action)

            else : # continuous
                mean = logit[0]
                cov = F.softplus(logit[1])
                action_distribution = torch.distributions.normal.Normal(mean, cov)
                action = action_distribution.sample()
                log_odds = action_distribution.log_prob(action)
            return action, log_odds
```

Create a function update_policy that defines the loss function and updates the MLP according to the REINFORCE update rule (ref. slide 24 of Lec 7 or page 330 of Sutton and Barto (2018)). The update algorithm to be used below is slightly different: instead of updating the network at every time-step, we take the gradient of the loss averaged over a batch of timesteps (this is to make SGD more stable). We also use a baseline to reduce variance.

The discount factor is set as 1 here.

In [5]:
```
def reward2go(rewards, gamma =1):
    r2g = []
    acc_r = 0
    for r in reversed(rewards):
        acc_r = acc_r * gamma + r
        r2g.append(acc_r)
    return r2g[::-1]


def update_policy(paths, net):
    # paths: a list of paths (complete episodes, used to calculate return at each time
    # net: MLP object

    num_paths = len(paths)
    rew_cums = []
    log_odds = []
    # calculated as "reward to go"

    for path in paths:
        # rew_cums should record return at each time step for each path
        rew_cums += reward2go(path['reward'])
        # log_odds should record log_odds obtained at each timestep of path
        log_odds += path['log_odds']
        # calculated as "reward to go"

    # make log_odds, rew_cums each a vector
    rew_cums = np.array(rew_cums)
    log_odds = np.array(log_odds)
    rew_cums = (rew_cums - rew_cums.mean()) / (rew_cums.std() + 1e-5) # create baseli
    # calculate policy loss and average over paths
    policy_loss = -rew_cums.dot(log_odds)/ num_paths

    # take optimizer step
    optimizer.zero_grad()
    policy_loss.sum().backward()  # sum for cheetah, or may delete this sum for other
    optimizer.step()
```

Set up environment and instantiate objects. Your algorithm is to be tested on one discrete and two continuous environments.

In [6]:
```
use_cuda = torch.cuda.is_available()
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

```
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
Tensor = FloatTensor

# Select Environment

######discrete environment:
# env_name='CartPole-v0'

######continous environments:
# env_name='InvertedPendulum-v1'
env_name = 'HalfCheetah-v1'


# Make the gym environment
env = gym.make(env_name)
visualize = False
# animate=visualize
learning_rate = 1e-3

max_path_length=None
min_timesteps_per_batch = 2000  # sets the batch size for updating network

# Set random seeds
seed=0
torch.manual_seed(seed)
np.random.seed(seed)


# Saving parameters
logdir='./REINFORCE/'

if visualize:
    if not os.path.exists(logdir):
        os.mkdir(logdir)
    env = gym.wrappers.Monitor(env, logdir, force=True, video_callable=lambda episode_i

env._max_episodes_steps = min_timesteps_per_batch


# Is this env continuous, or discrete?
discrete = isinstance(env.action_space, gym.spaces.Discrete)


# Get observation and action space dimensions
obs_dim = env.observation_space.shape[0]
act_dim = env.action_space.n if discrete else env.action_space.shape[0]
```

```
# Maximum length for episodes
max_path_length = max_path_length or env.spec.max_episode_steps

# Make network object (remember to pass in appropriate flags for the type of action spa
# net = mlp(*args)
net = mlp(Dim_state = obs_dim, num_outputs = act_dim, disct = discrete).type(FloatTenso

# Make optimizer
optimizer = torch.optim.Adam(net.parameters(), lr = learning_rate)
print(net)
```

```
[2018-05-15 14:26:39,297] Making new env: HalfCheetah-v1
[2018-05-15 14:26:39,630] Finished writing results. You can upload them to the scoreboard via g


mlp(
  (fc1): Linear(in_features=17, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=50, bias=True)
  (fc_mu): Linear(in_features=50, out_features=6, bias=True)
  (fc_sigma): Linear(in_features=50, out_features=6, bias=True)
)
```

6.2 Run REINFORCE

Run REINFORCE for CartPole, InvertedPendulum, and HalfCheetah.

In [7]:
```
n_iter = 1000
min_timesteps_per_batch = 2000  # sets the batch size for updating network
avg_reward = 0
avg_rewards = []
step_list_reinforce = []
total_steps = 0
episodes = 0

for itr in range(n_iter): # loop for number of optimization steps
    paths = []
    steps = 0
    while True: # loop to get enough timesteps in this batch --> if episode ends this
        ob = env.reset()
        animate_this_episode = (itr % animate_interval == 0) and visualize
        obs, acs, rews, log_odds = [], [], [], []
        obs.append(ob)

        while True: # loop for episode inside batch
            if animate_this_episode:
                env.render()
                time.sleep(0.05)
            # get parametrized policy distribution from net using current state ob
```

```
            net.eval()
            var_ob = Variable(torch.unsqueeze(FloatTensor(ob),0), requires_grad=False)
            distribution_parameters = net(var_ob)

            net.train()
            # sample action and get log-probability (log_odds) from distribution
            cuda_tensor_ac, log_odd= sample_action(logit = distribution_parameters , di
            ac = cuda_tensor_ac.data[0].cpu().numpy()

            # step environment, record reward, next state
            ob, rew, done, _ = env.step(ac)
            # append to obs, acs, rewards, log_odds
            obs.append(ob)
            acs.append(ac)
            rews.append(rew)
            log_odds.append(log_odd)

            # if done, restart episode till min_timesteps_per_batch is reached
            steps += 1

            if done:
                episodes = episodes + 1
                break

        path = {"observation" : obs,
                "reward" : np.array(rews),
                "action" : (acs),
                "log_odds" : log_odds}

        paths.append(path)
        if steps > min_timesteps_per_batch: break

    update_policy(paths, net)  # use all complete episodes (a batch of timesteps) reco
    if itr == 0: avg_reward = path['reward'].sum()

    else: avg_reward = avg_reward * 0.95 + 0.05 * path['reward'].sum()

    if avg_reward > 1500: break
    # inverted 500, half_cheetah 1500, cartpole 200
    total_steps += steps
    print(avg_reward,end='\r')
    avg_rewards.append(avg_reward)
    step_list_reinforce.append(total_steps)
    if itr % logging_interval == 0: print('Average reward: {}'.format(avg_reward))

env.close()
print('done')
```

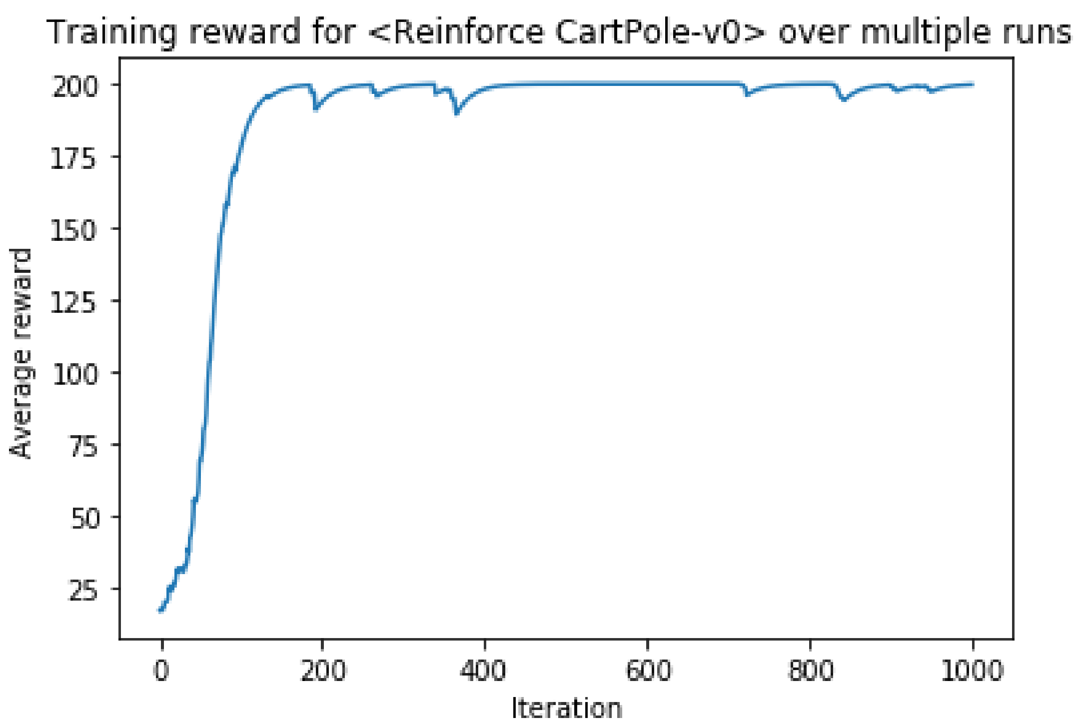
```
Average reward: -631.3082183173344
Average reward: -619.8083439120957
Average reward: -479.22020007268867
Average reward: -535.6534700193245
Average reward: -483.31511565534953
Average reward: -700.0253135620237
Average reward: -539.8485582303404
Average reward: -486.3342675842881
Average reward: -573.6608333884494
Average reward: -762.6001722762232
done.59865768538844
```

7 Reinforce CartPole-v0

In [22]:
```python
np.save('reinforce_cartpole_reward.npy', avg_rewards)
np.save('reinforce_cartpole_step.npy', step_list_reinforce)
```

In [14]:
```python
plt.plot(avg_rewards)
plt.title('Training reward for <Reinforce CartPole-v0> over multiple runs ')
plt.xlabel('Iteration')
plt.ylabel('Average reward')
```

Out[14]:
```
Text(0,0.5,'Average reward')
```



In [19]:
```
env_name='CartPole-v0'

# Make the gym environment
env = gym.make(env_name)
visualize = True
animate=visualize
learning_rate = 1e-3

max_path_length=None
min_timesteps_per_batch = 2000  # sets the batch size for updating network

# Set random seeds
seed=0
torch.manual_seed(seed)
np.random.seed(seed)

use_cuda = torch.cuda.is_available()
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
Tensor = FloatTensor

# Saving parameters
logdir='./REINFORCE/'

if visualize:
    if not os.path.exists(logdir):
        os.mkdir(logdir)
    env = gym.wrappers.Monitor(env, logdir, force=True, video_callable=lambda episode_

env._max_episodes_steps = min_timesteps_per_batch


# Is this env continuous, or discrete?
discrete = isinstance(env.action_space, gym.spaces.Discrete)

# Get observation and action space dimensions
obs_dim = env.observation_space.shape[0]
act_dim = env.action_space.n if discrete else env.action_space.shape[0]

# Maximum length for episodes
max_path_length = max_path_length or env.spec.max_episode_steps
```

```
[2018-05-15 10:00:46,789] Making new env: CartPole-v0
[2018-05-15 10:00:46,795] Clearing 2 monitor files from previous run (because force=True was p
```

In [21]:
```
########### for saving optimal model video
```

```
ob = env.reset()
while True:
    env.render()
    time.sleep(0.05)
    # get parametrized policy distribution from net using current state ob
    net.eval()
    var_ob = Variable(torch.unsqueeze(FloatTensor(ob),0), requires_grad=False)
    distribution_parameters = net(var_ob)
    # sample action and get log-probability (log_odds) from distribution
    cuda_tensor_ac, log_odd= sample_action(logit = distribution_parameters , disct = d
    ac = cuda_tensor_ac.data[0].cpu().numpy()
    # step environment, record reward, next state
    new_ob, rew, done, _ = env.step(ac)
    ob = new_ob
    if done: break
print('done')
```

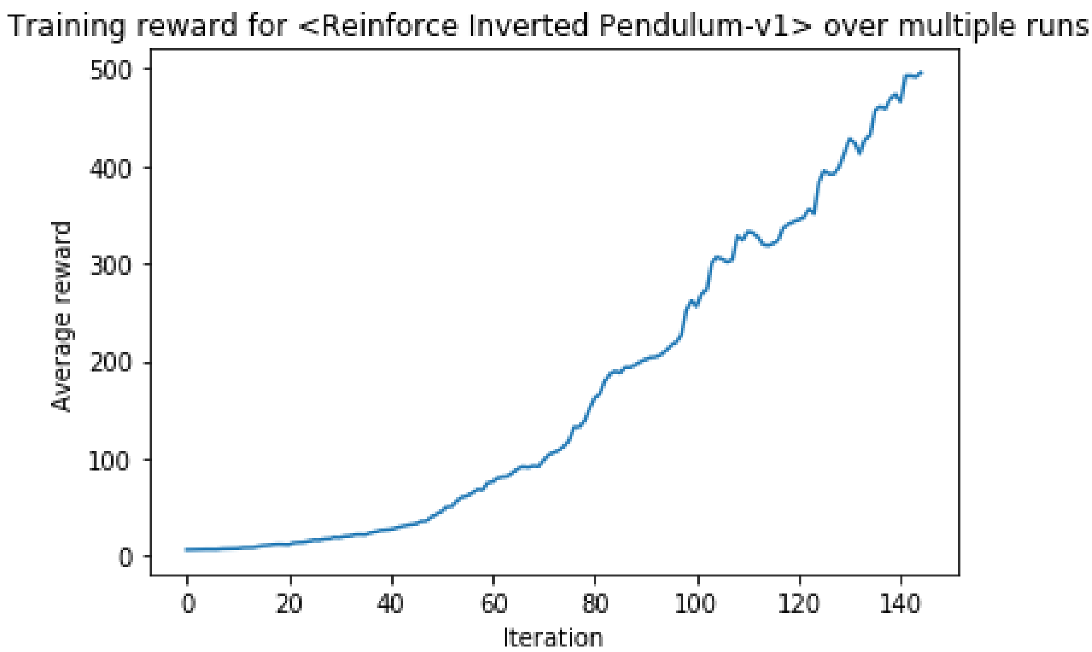
```
done
```

8 Reinforce Inverted Pendulum-v1

In [39]:
```
np.save('reinforce_inverted_pendulum_reward.npy', avg_rewards)
np.save('reinforce_inverted_pendulum_step.npy', step_list_reinforce)
```

In [12]:
```
plt.plot(avg_rewards)
plt.title('Training reward for <Reinforce Inverted Pendulum-v1> over multiple runs ')
plt.xlabel('Iteration')
plt.ylabel('Average reward')
```

Out[12]:
```
Text(0,0.5,'Average reward')
```




```
In [41]: env_name='InvertedPendulum-v1'

         # Make the gym environment
         env = gym.make(env_name)
         visualize = True
         animate=visualize
         learning_rate = 1e-3

         max_path_length=None
         min_timesteps_per_batch = 2000  # sets the batch size for updating network

         # Set random seeds
         seed=0
         torch.manual_seed(seed)
         np.random.seed(seed)

         use_cuda = torch.cuda.is_available()
         FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
         LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
         ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
         Tensor = FloatTensor

         # Saving parameters
         logdir='./REINFORCE/'

         if visualize:
```

```
    if not os.path.exists(logdir):
        os.mkdir(logdir)
    env = gym.wrappers.Monitor(env, logdir, force=True, video_callable=lambda episode_

env._max_episodes_steps = min_timesteps_per_batch


# Is this env continuous, or discrete?
discrete = isinstance(env.action_space, gym.spaces.Discrete)

# Get observation and action space dimensions
obs_dim = env.observation_space.shape[0]
act_dim = env.action_space.n if discrete else env.action_space.shape[0]

# Maximum length for episodes
max_path_length = max_path_length or env.spec.max_episode_steps
```

```
[2018-05-15 10:17:45,697] Making new env: InvertedPendulum-v1
[2018-05-15 10:17:45,705] Clearing 6 monitor files from previous run (because force=True was pr
```

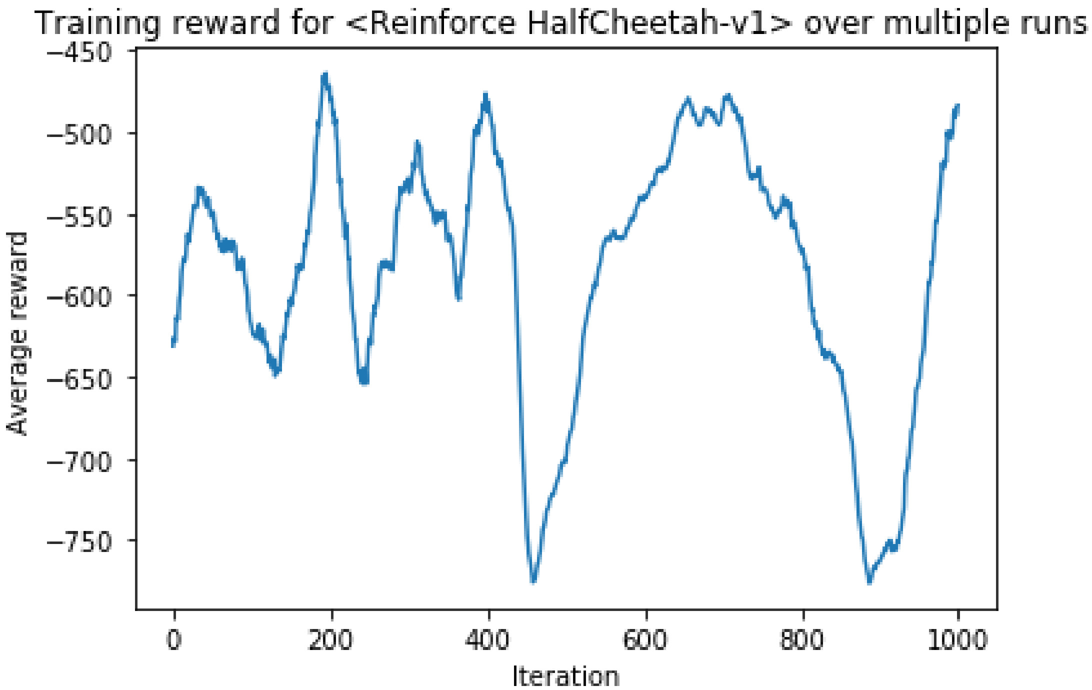
In [43]:
```
########### for saving optimal model video

ob = env.reset()
while True:
    env.render()
    time.sleep(0.05)
    # get parametrized policy distribution from net using current state ob
    net.eval()
    var_ob = Variable(torch.unsqueeze(FloatTensor(ob),0), requires_grad=False)
    distribution_parameters = net(var_ob)
    # sample action and get log-probability (log_odds) from distribution
    cuda_tensor_ac, log_odd= sample_action(logit = distribution_parameters , disct = d
    ac = cuda_tensor_ac.data[0].cpu().numpy()
    # step environment, record reward, next state
    new_ob, rew, done, _ = env.step(ac)
    ob = new_ob
    if done: break
print('done')
```

```
done
```

9 Reinforce HalfCheetah-v1

In [8]:
```
np.save('reinforce_halfcheetah_reward.npy', avg_rewards)
np.save('reinforce_halfcheetah_step.npy', step_list_reinforce)
```

```
In [10]: plt.plot(avg_rewards)
         plt.title('Training reward for <Reinforce HalfCheetah-v1> over multiple runs ')
         plt.xlabel('Iteration')
         plt.ylabel('Average reward')

Out[10]: Text(0,0.5,'Average reward')
```




```
In [11]: env_name='HalfCheetah-v1'

         # Make the gym environment
         env = gym.make(env_name)
         visualize = True
         animate=visualize
         learning_rate = 1e-3

         max_path_length=None
         min_timesteps_per_batch = 2000  # sets the batch size for updating network

         # Set random seeds
         seed=0
         torch.manual_seed(seed)
         np.random.seed(seed)

         use_cuda = torch.cuda.is_available()
```

```
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
Tensor = FloatTensor

# Saving parameters
logdir='./REINFORCE/'

if visualize:
    if not os.path.exists(logdir):
        os.mkdir(logdir)
    env = gym.wrappers.Monitor(env, logdir, force=True, video_callable=lambda episode_

env._max_episodes_steps = min_timesteps_per_batch


# Is this env continuous, or discrete?
discrete = isinstance(env.action_space, gym.spaces.Discrete)

# Get observation and action space dimensions
obs_dim = env.observation_space.shape[0]
act_dim = env.action_space.n if discrete else env.action_space.shape[0]

# Maximum length for episodes
max_path_length = max_path_length or env.spec.max_episode_steps
```

```
[2018-05-15 15:42:06,135] Making new env: HalfCheetah-v1
[2018-05-15 15:42:06,150] Clearing 4 monitor files from previous run (because force=True was p
```

In [13]:

```
ob = env.reset()
while True:
    env.render()
    time.sleep(0.05)
    # get parametrized policy distribution from net using current state ob
    net.eval()
    var_ob = Variable(torch.unsqueeze(FloatTensor(ob),0), requires_grad=False)
    distribution_parameters = net(var_ob)
    # sample action and get log-probability (log_odds) from distribution
    cuda_tensor_ac, log_odd= sample_action(logit = distribution_parameters , disct = 
    ac = cuda_tensor_ac.data[0].cpu().numpy()
    # step environment, record reward, next state
    new_ob, rew, done, _ = env.step(ac)
    ob = new_ob
    if done: break
print('done')
```

```
done
```

10 BONUS (15% extra)

Compare average returns for CartPole (discrete action space) when using REINFORCE and DQN. Since in REINFORCE we update the network after a set number of steps instead of after every episode, plot the average rewards as a function of steps rather than episodes for both DQN and REINFORCE. You will need to make minor edits to your DQN code from the previous assignment to record average returns as a function of time_steps.

Similarly, compare REINFORCE with DDPG on InvertedPendulum and HalfCheetah using steps for the x-axis.

You may use the example code provided below as a reference for the graphs.

11 DQN

11.1 1.1 DQN environment setup

In [25]:
```
# import your DQN and format your average returns as defined above
import gym
import numpy as np
import matplotlib.pyplot as plt

from collections import namedtuple
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable

# Create the CartPole game environment
env = gym.make('CartPole-v0')
env.reset()
use_cuda = torch.cuda.is_available()
# use_cuda = False
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor

class Net(nn.Module):
# Define your network here
    def __init__(self, state_size, action_size, hidden_size):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc1.weight.data.normal_(0, 0.1)   # initialization
        self.out = nn.Linear(hidden_size, action_size)
        self.out.weight.data.normal_(0, 0.1)   # initialization
    def forward(self, x):
        x = self.fc1(x)
        x = F.tanh(x)
```

```
            Qs_actions = self.out(x) # Q value for one state, at different actions
            return Qs_actions

    class QNetwork:
        def __init__(self, learning_rate, state_size, action_size, hidden_size, alpha_deca
            self.LR = learning_rate
            self.state_size = state_size
            self.action_size = action_size
            self.hidden_size = hidden_size
            self.alpha_decay = alpha_decay
            self.model = Net(self.state_size, self.action_size, self.hidden_size)
            self.optimizer = torch.optim.Adam(self.model.parameters(), lr=self.LR)
            self.criterion = nn.MSELoss()

        def learn(self, batch_Q_behavior, batch_Q_target):
            loss = self.criterion(batch_Q_behavior, batch_Q_target)
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
```

```
[2018-05-15 13:02:24,780] Making new env: CartPole-v0
```

11.2 1.2 DQN replay buffer

```
In [26]: class Replay():
             def __init__(self, max_size):
                 self.capacity = max_size
                 self.memory = []
                 self.position = 0
                 self.gamma = 0.99

             def initialize(self, init_length, envir):
                 st = env.reset()
                 for _ in range(init_length):
                     a = np.random.randint(2, size=1)
                     st1, r, done, info = env.step(int(a))
                     self.push((st, a, st1, r, done))
                     if done: st = env.reset()
                     else : st = st1

             def push(self, transition):
                 if len(self.memory) < self.capacity:
                     self.memory.append(None)
                 self.memory[self.position] = transition
                 self.position = (self.position + 1) % self.capacity
```

```
    def generate_minibatch(self, DQN, targetDQN, batch_size):

        batch_memory = random.sample(self.memory, batch_size) #return a list
        batch_memory = list(zip(*batch_memory))

        batch_st = Variable(FloatTensor(batch_memory[0]))
        batch_at = Variable(LongTensor(batch_memory[1]))
        batch_st1 = Variable(FloatTensor(batch_memory[2]))
        batch_r = Variable(torch.unsqueeze(FloatTensor(batch_memory[3]),1))
        batch_done = FloatTensor(batch_memory[4])

        batch_Q_behavior = DQN.model(batch_st).gather(1, batch_at)
        mask = 1. - batch_done
        batch_Q_next = targetDQN.model(batch_st1).detach()

        QQ_next = Variable((batch_Q_next.max(1)[0].data * mask).view(batch_size, 1))
        batch_Q_target = batch_r + self.gamma*(QQ_next)
        return batch_Q_behavior, batch_Q_target

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

11.3 1.3 DQN training

```
In [27]: learning_rate = 0.01
         action_size = env.action_space.n
         state_size = env.observation_space.shape[0]
         hidden_size = 64
         alpha_decay = 0.1
         batch_size = 500

         DQN = QNetwork(learning_rate, state_size, action_size, hidden_size, alpha_decay)
         targetDQN = QNetwork(learning_rate, state_size, action_size, hidden_size, alpha_decay
         # set targetDQN weights to DQN weights
         # for ex. targetDQN.model.weights = DQN.model.weights (syntax given here is for repre
         targetDQN.model.load_state_dict(DQN.model.state_dict())
         replay = Replay(max_size=10000) ## Initialize Replay Buffer
         replay.initialize(init_length=1000, envir=env) ## Populate the initial experience buf
         if use_cuda:
             print('run gpu !')
             targetDQN.model.cuda()
             DQN.model.cuda()
         else:
             print('gpu not activited !')

         # Runtime parameters
```

```
num_episodes = 2000            # max number of episodes to learn from
gamma = 0.99                   # future reward discount
max_steps = 500                # cut off simulation after this many steps
# Exploration parameters
min_epsilon = 0.01             # minimum exploration probability
decay_rate = 5/num_episodes    # exponential decay rate for exploration prob
returns = np.zeros(num_episodes)
step_list_DQN = []
total_steps = 0
avg_reward = 0
avg_rewards = []
logging_interval = 100


for ep in range(1, num_episodes): # ep now is for one iteration
    paths = []
    steps = 0
    while True: #  paths = a number of episode, but restricted by step> 2000 break
        total_reward = 0
        epsilon = min_epsilon + (1.0 - min_epsilon)*np.exp(-decay_rate*ep)
    # --> start episode
        state = env.reset()
        rews = []
        for step in range(max_steps): # path = one episode
            # generate the steps in each episode
            # explore/exploit and get action using DQN
            if random.random()<= epsilon:
                action = np.random.randint(2, size=1)
            else:
                var_state = Variable(torch.unsqueeze(FloatTensor(state),0))# here cha
                DQN.model.eval()
                Qs_actions = DQN.model.forward(var_state) # shape of (1, 2) variable
                DQN.model.train()
                cuda_tensor_action = torch.max(Qs_actions,1)[1].data
                action = cuda_tensor_action.cpu().numpy()

            new_state, reward, done, _ = env.step(int(action))
            rews.append(reward)
            replay.push((state, action, new_state, reward, done))
            steps += 1
        # perform action and record new_state, action, reward
        # populate Replay experience buffer
            if done:  break
            else: state = new_state
        # <-- end episode
        path={'reward':np.array(rews)}
        paths.append(path)
        if steps > 2000: break
```

```
            batch_Q_behavior, batch_Q_target = replay.generate_minibatch(DQN, targetDQN, batcl
            DQN.learn(batch_Q_behavior, batch_Q_target)
            targetDQN.model.load_state_dict(DQN.model.state_dict())

            avg_reward = avg_reward * 0.95 + 0.05 * path['reward'].sum()
            total_steps += steps
            avg_rewards.append(avg_reward)
            step_list_DQN.append(total_steps)
            if ep % logging_interval == 0: print('Average reward: {}'.format(avg_reward))
        print('finished training')
```

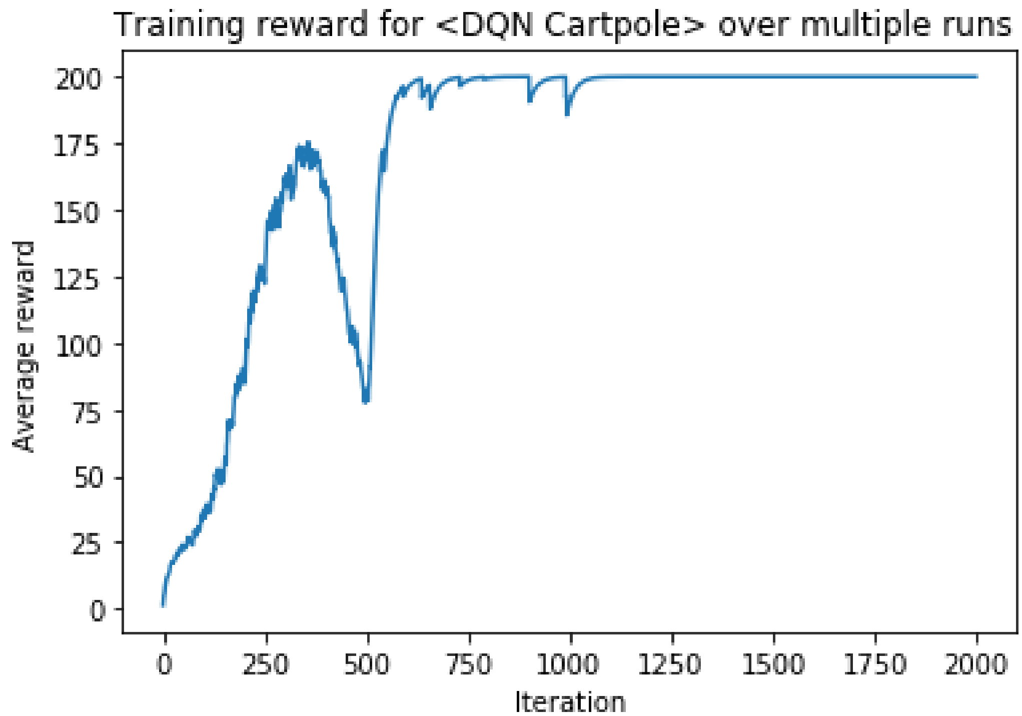
```
run gpu !
Average reward: 35.93400687794351
Average reward: 88.21826325214823
Average reward: 162.31065543738222
Average reward: 158.0463836316632
Average reward: 80.92988438852923
Average reward: 195.15084026015444
Average reward: 198.58993299416161
Average reward: 199.4181045682817
Average reward: 199.99655487109317
Average reward: 189.37563464654048
Average reward: 199.93709813447714
Average reward: 199.999627587667
Average reward: 199.99999779512174
Average reward: 199.9999999869458
Average reward: 199.99999999992258
Average reward: 199.99999999999935
Average reward: 199.99999999999955
Average reward: 199.99999999999955
Average reward: 199.99999999999955
finished training
```

11.4 1.4 DQN Cartpole Results

```
In [28]: np.save('DQN_cartpole_reward.npy', avg_rewards)
         np.save('DQN_cartpole_step.npy', step_list_DQN)
```

```
In [29]: plt.plot(avg_rewards)
         plt.title('Training reward for <DQN Cartpole> over multiple runs ')
         plt.xlabel('Iteration')
         plt.ylabel('Average reward')
```

```
Out[29]: Text(0,0.5,'Average reward')
```



12 DQN vs Reinforce

12.1 CartPole (discrete action space)

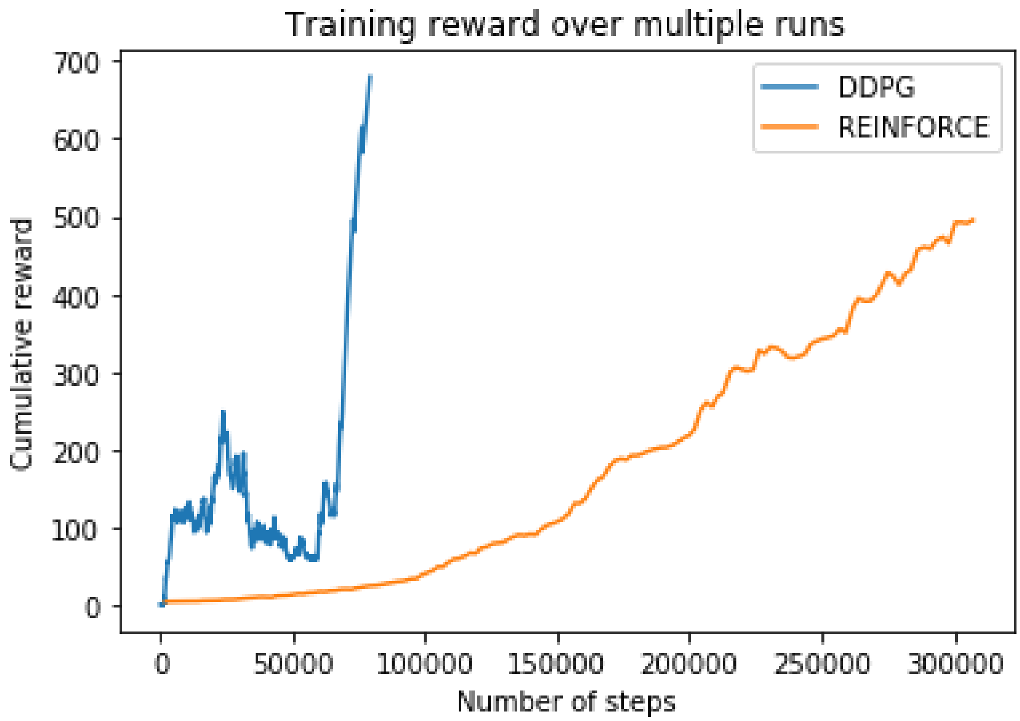
```
In [4]: running_rewards_ddpg=np.load('DQN_cartpole_reward.npy')
        step_list_DQN=np.load('DQN_cartpole_step.npy')

        avg_rewards = np.load('reinforce_cartpole_reward.npy')
        step_list_reinforce = np.load('reinforce_cartpole_step.npy')

In [5]: plt.figure()
        out = numpy_ewma_vectorized_v2(np.array(running_rewards_ddpg),20)
        plt.plot(step_list_DQN, out)
        plt.title('Training reward over multiple runs')
        plt.xlabel('Number of steps')
        plt.ylabel('Cumulative reward')
        plt.plot(step_list_reinforce, avg_rewards)
        plt.legend(['DQN', 'REINFORCE'])
        plt.show()
```

13 DDPG vs Reinforce

13.1 Inverted Pendulum-v1

```
In [17]: running_rewards_ddpg=np.load('ddpg_inverted_pendulum_reward.npy')
         step_list_ddpg=np.load('ddpg_inverted_pendulum_step.npy')

         avg_rewards = np.load('reinforce_inverted_pendulum_reward.npy')
         step_list_reinforce = np.load('reinforce_inverted_pendulum_step.npy')
```

```
In [18]: plt.figure()

         out = numpy_ewma_vectorized_v2(np.array(running_rewards_ddpg),20)
         plt.plot(step_list_ddpg, out)
         plt.title('Training reward over multiple runs')
         plt.xlabel('Number of steps')
         plt.ylabel('Cumulative reward')
         plt.plot(step_list_reinforce, avg_rewards)
         plt.legend(['DDPG', 'REINFORCE'])

         plt.show()
```



13.2 HalfCheetah-v1

```
In [17]: running_rewards_ddpg=np.load('ddpg_halfcheetah_reward.npy')
         step_list_ddpg=np.load('ddpg_halfcheetah_step.npy')

         avg_rewards = np.load('reinforce_halfcheetah_reward.npy')
         step_list_reinforce = np.load('reinforce_halfcheetah_step.npy')

In [18]: plt.figure()
         out = numpy_ewma_vectorized_v2(np.array(running_rewards_ddpg),20)
         plt.plot(step_list_ddpg, out) # or plt.plot(step_list_DQN, out)
         plt.title('Training reward over multiple runs')
         plt.xlabel('Number of steps')
         plt.ylabel('Cumulative reward')
         plt.plot(step_list_reinforce, avg_rewards)
         plt.legend(['DDPG', 'REINFORCE'])

         plt.show()
```


Training reward over multiple runs
DDPG
REINFORCE
2000
1500
1000
500
0
−500
Cumulative reward
0
500000
1000000
1500000
2000000
2500000
3000000
Number of steps