Report of Assignment 5

717030210005 Zhexin Jin

A3C

1. Packages and Modules

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.multiprocessing as mp
import torch.optim as optim
import numpy as np
import gym
import math, os
import matplotlib.pyplot as plt
```

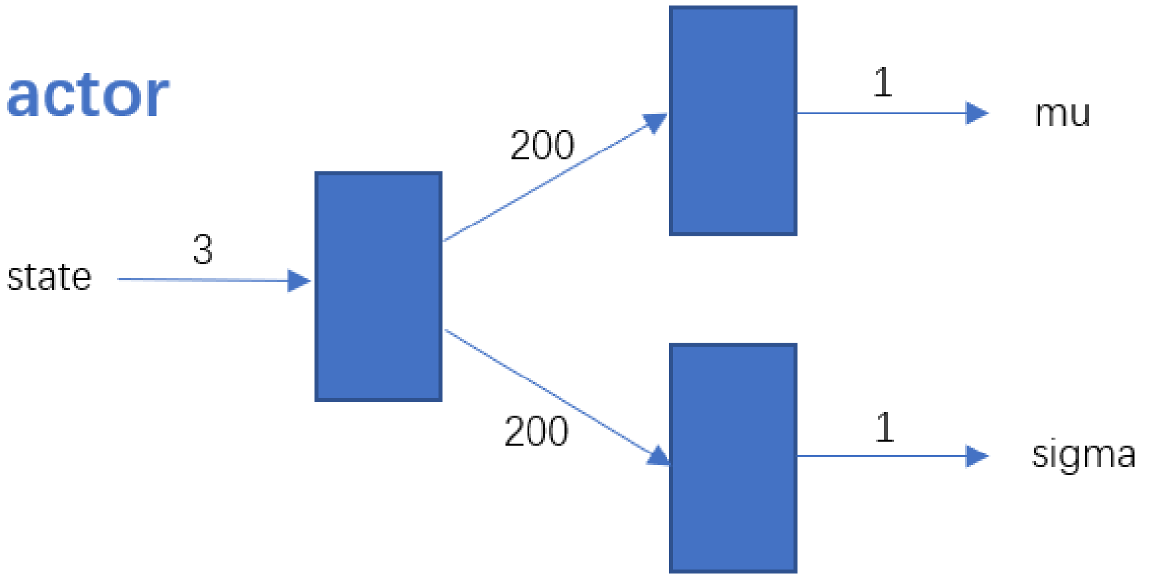
2. Neural network

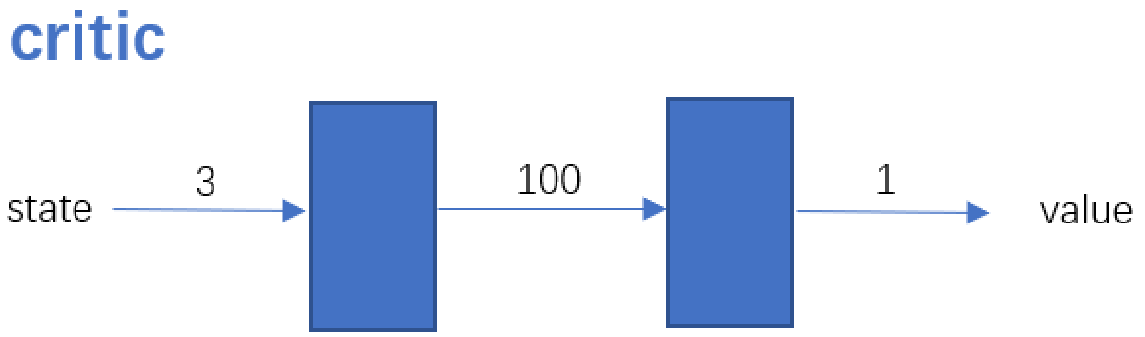
```
class Net(nn.Module):

    def __init__(self, s_dim, a_dim):
        super(Net, self).__init__()
        self.s_dim = s_dim
        self.a_dim = a_dim

        # actor net:
        # input state
        # output mu and sigma of a normal distribution
        # mu and sigma can describe the possibility of actions
        self.a1 = nn.Linear(s_dim, 200)
        self.mu = nn.Linear(200, a_dim)
        self.sigma = nn.Linear(200, a_dim)

        # critic net:
        # input state
        # output value
        self.c1 = nn.Linear(s_dim, 100)
        self.v = nn.Linear(100, 1)

        # set distribution function
        self.distribution = torch.distributions.Normal

    # input state

    # output mu, sigma, values
```

```
    def forward(self, x):
        # x -> a1
        a1 = F.relu6(self.a1(x))
        # a1 -> mu
        mu = 2 * torch.tanh(self.mu(a1))
        # a1 -> sigma
        # + 0.0001 to avoid 0
        sigma = F.softplus(self.sigma(a1)) + 0.0001
        # x -> c1
        c1 = F.relu6(self.c1(x))
        # c1 -> values
        values = self.v(c1)
        return mu, sigma, values

    # choose an action through normal distribution
    def choose_action(self, s):
        self.training = False
        mu, sigma, _ = self.forward(s)
        m = self.distribution(mu, sigma)
        # use clip to limit the action in [-2, 2]
        return m.sample().numpy().clip(-2, 2)

    # calculate loss
    def loss_func(self, s, a, R):
        self.train()

        # critic loss
        mu, sigma, values = self.forward(s)
        advantage = R - values
        c_loss = advantage.pow(2)

        # actor loss
        m = self.distribution(mu, sigma)
        log_prob = m.log_prob(a)
        entropy = 0.5 + 0.5 * math.log(2 * math.pi) + torch.log(m.scale)
        exp_v = log_prob * advantage.detach() + 0.005 * entropy
        a_loss = -exp_v

        total_loss = (a_loss + c_loss).mean()
        return total_loss
```

This is the definition of actor network and critic network.

The architecture of networks:





In loss function, the calculation of actor loss is different from the formula that teacher gives to us. But it works better than the original formula.

This formula is taken from https://github.com/MorvanZhou/pytorch-A3C .

3. Process

```
class SharedAdam(torch.optim.Adam):

    def __init__(self, params, lr=1e-3, betas=(0.9, 0.9), eps=1e-8,
                 weight_decay=0):
        super(SharedAdam, self).__init__(params, lr=lr, betas=betas, eps=eps,
weight_decay=weight_decay)
        # State initialization
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['step'] = 0
                state['exp_avg'] = torch.zeros_like(p.data)
                state['exp_avg_sq'] = torch.zeros_like(p.data)
```

```
                # share in memory
                state['exp_avg'].share_memory_()
                state['exp_avg_sq'].share_memory_()
```

This is an optimizer function. It is taken from https://github.com/MorvanZhou/pytorch-A3C .

I don't know what all these parameters are, but it works well.

```
class Worker(mp.Process):

    def __init__(self, gnet, global_ep, global_ep_r, res_queue, id):
        super(Worker, self).__init__()
        # the process id
        self.id = id
        # g_ep: global episode count
        # g_ep_r: global average reward of last 10 episodes
        # res_queue: store every g_ep_r
        self.g_ep, self.g_ep_r, self.res_queue = global_ep, global_ep_r, res_queue
        # global net
        self.gnet = gnet
        # optimizer
        self.opt = SharedAdam(gnet.parameters(), lr=0.0002)
        # local net
        self.lnet = Net(N_S, N_A)
        # environment
        self.env = gym.make('Pendulum-v0').unwrapped

    # the main function of this process
    # when process start, this function will be run
    def run(self):
        total_step = 1
        # loop until g_ep >= MAX_EP
        while self.g_ep.value < MAX_EP:
            # reset the env
            s = self.env.reset()
            # buffers to store state, action, reward of every transition
            buffer_s, buffer_a, buffer_r = [], [], []
            # total reward of this episode
            ep_r = 0.0
            for t in range(MAX_EP_STEP):
                # show the movement in process 1
                if self.id == 1:
                    self.env.render()
                # choose an action from local net
                a = self.lnet.choose_action(torch.FloatTensor(s))
                # take the action
                s_, r, done, _ = self.env.step(a)
                # done if already take 200 steps
                if t == MAX_EP_STEP - 1: done = True

                # update episode reward
                ep_r += r

                # store acion, state, reward
```

```
                buffer_a.append(a)
                buffer_s.append(s)
                buffer_r.append((r+8.1)/8.1)

                if total_step % UPDATE_GLOBAL_ITER == 0 or done:
                    # learn every 5 steps
                    self.learn(s_, buffer_s, buffer_a, buffer_r)
                    # clear buffers
                    buffer_s, buffer_a, buffer_r = [], [], []
                    # if done, store episode reward and print
                    if done:
                        self.record(ep_r)
                        break
                # state <= next_state
                s = s_
                total_step += 1
        # if have finished MAX_EP episodes, return 0 through res_queue
        self.res_queue.put(0)

    # update lnet and gnet
    def learn(self, s_, bs, ba, br):
        # R <= 0 for terminal
        # R <= V(s_t) for non_terminal
        # but there is never terminal
        R = self.lnet.forward(torch.Tensor(s_))[-1][0].item()
        # R <= r_i + gamma * R
        buffer_R = []
        for r in br[::-1]:
            R = r + GAMMA * R
            buffer_R.append(R)
        buffer_R.reverse()
        # calculate loss
        loss = self.lnet.loss_func(
            torch.FloatTensor(bs),
            torch.FloatTensor(ba),
            torch.Tensor(buffer_R).view(-1,1))
        # update global net
        self.opt.zero_grad()
        loss.backward()
        for lp, gp in zip(self.lnet.parameters(), self.gnet.parameters()):
            gp._grad = lp.grad
        self.opt.step()
        # update local net
        # copy parameters from global net to local net
        self.lnet.load_state_dict(self.gnet.state_dict())

    # update g_ep, g_ep_r
    # return g_ep_r through res_queue
    # print g_ep and g_ep_r
    def record(self, ep_r):
        with self.g_ep.get_lock():
            self.g_ep.value += 1

        with self.g_ep_r.get_lock():
```

```
            if self.g_ep_r.value == 0.0:
                self.g_ep_r.value = ep_r
            else:
                self.g_ep_r.value = self.g_ep_r.value * 0.9 + ep_r * 0.1
        self.res_queue.put(self.g_ep_r.value)
        print("Ep:", self.g_ep.value, "| Ep_r: %d" % self.g_ep_r.value)
        # self.res_queue.put(ep_r)
        # print("Ep:", self.g_ep.value, "| Ep_r: %d" % ep_r)

    # when finish training
    # show the result of the policy
    def show(self):
        while True:
            s = self.env.reset()
            for t in range(MAX_EP_STEP):
                self.env.render()
                a = self.gnet.choose_action(torch.FloatTensor(s))
                s, _, _, _ = self.env.step(a)
```

Process.run() will be run when process starts.

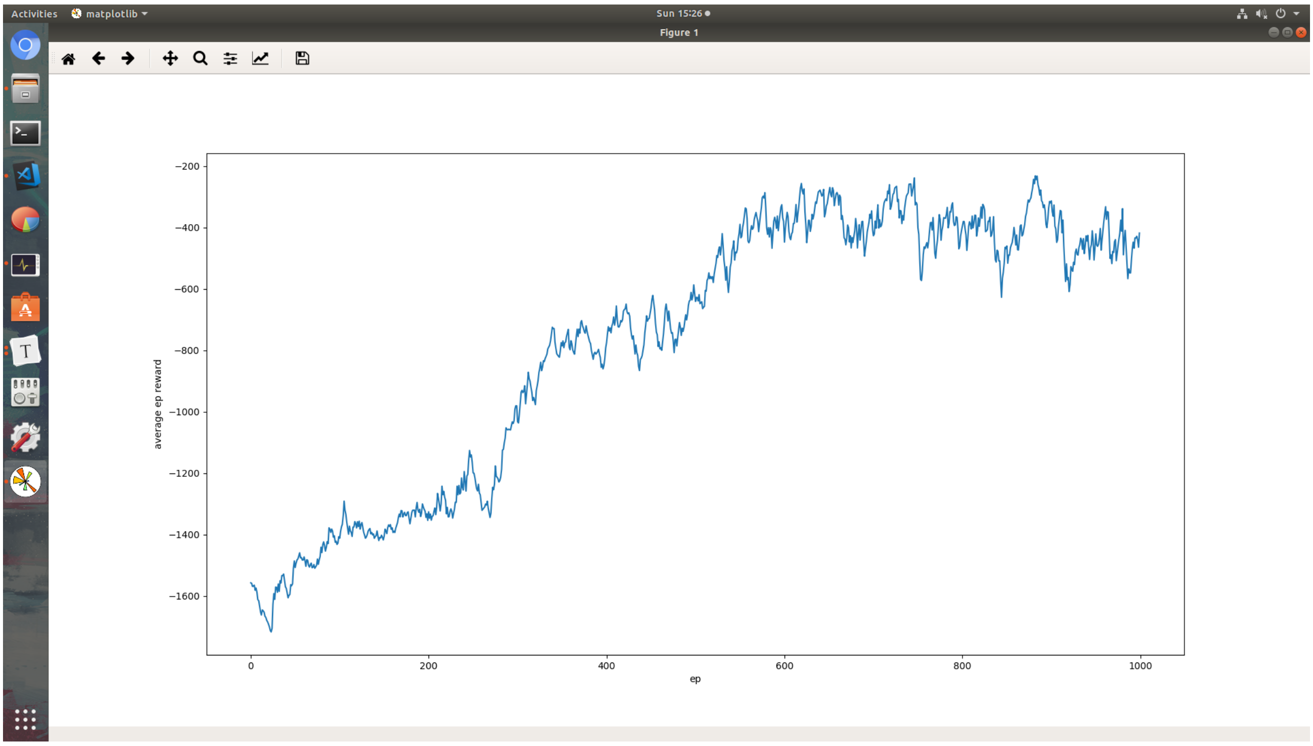
The res_queue is used to store rewards of episodes and share with other processes.

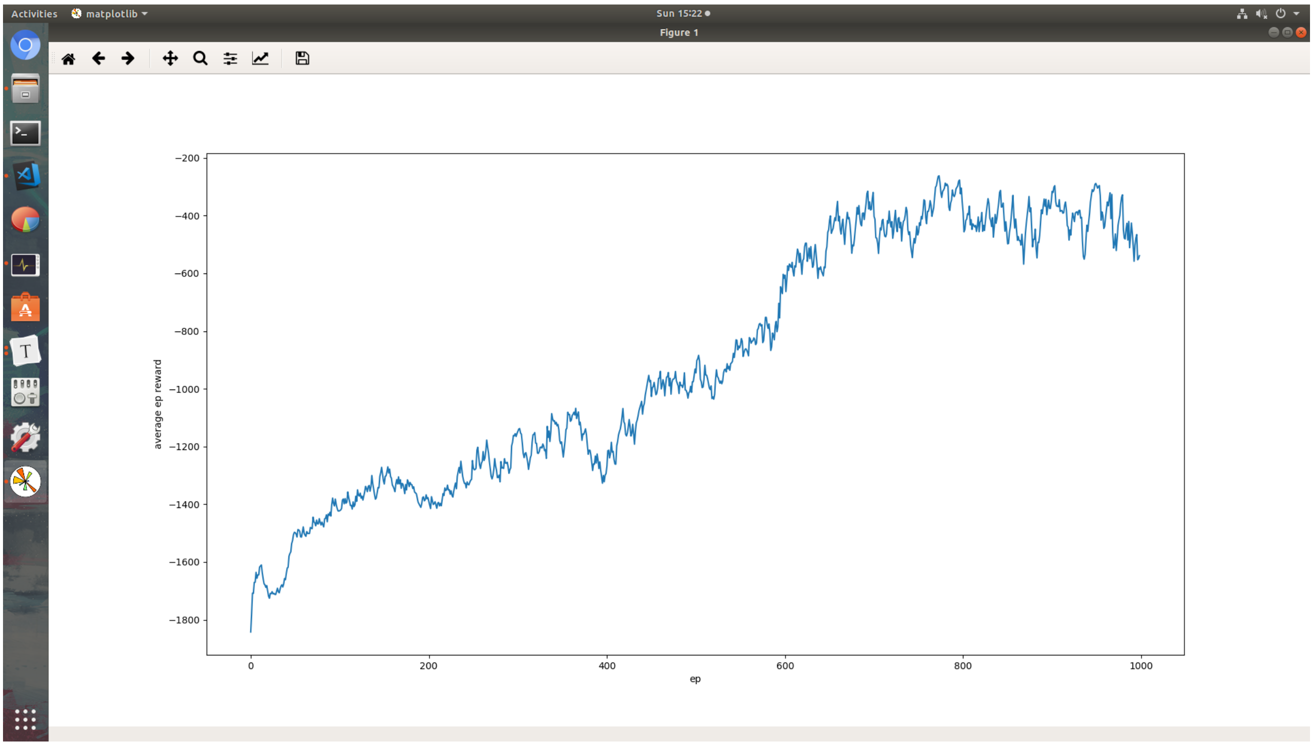
4. Main

```
# some parameters
UPDATE_GLOBAL_ITER = 5
GAMMA = 0.9
MAX_EP = 1000
MAX_EP_STEP = 200
N_CPU = mp.cpu_count()
env = gym.make('Pendulum-v0')
N_S = env.observation_space.shape[0]
N_A = env.action_space.shape[0]

if __name__ == "__main__":
    # initialize
    gnet = Net(N_S, N_A)
    gnet.share_memory()
    global_ep, global_ep_r, res_queue = mp.Value('i', 0), mp.Value('d', 0.), mp.Queue()
    workers = [Worker(gnet, global_ep, global_ep_r, res_queue, i) for i in range(N_CPU)]
    # start processes
    [w.start() for w in workers]
    res = []
    # get average rewards from res_queue
    while True:
        r = res_queue.get()
        if r:
            res.append(r)
        else:
            break
    [w.join() for w in workers]

    # processes stop
```

```
    # draw chart
    plt.plot(res)
    plt.ylabel('average ep reward')
    plt.xlabel('ep')
    plt.show()
    # show the movement of the pendulum using trained policy
    workers[0].show()
```

5. Results and conclusion





According to my observation, when reward is above -600, it means that the policy can keep the pendulum upright successfully. And when reward is above -400, it means that the policy can swing the pendulum up and keep it upright as soon as possible.

It seems that A3C does work, but this method doesn't work best. And it needs about 700 episodes to converge.

DDPG

1. Packages and Modules

```
import gym
import numpy as np
import torch
from torch.autograd import Variable
import torch.multiprocessing as mp
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim
import os
import random
import matplotlib.pyplot as plt
```

2. Memory

```
# memory to store past transitions
class Memory:

    def __init__(self):
        self.memory = []
        self.len = 0
        self.capacity = 10000
        self.batch_size = 32

    # get a batch of transitions from memory
    def sample(self):
        batch = random.sample(self.memory, self.batch_size)
        [s, a, r, s_] = zip(*batch)
        s = torch.Tensor(s)
        a = torch.Tensor(a)
        r = torch.Tensor(r)
        s_ = torch.Tensor(s_)
        return s, a, r, s_

    # push a transition into memory
    def push(self, s, a, r, s1):
        transition = (s,a,r,s1)
        if self.len >= self.capacity:

            self.len = 0
```

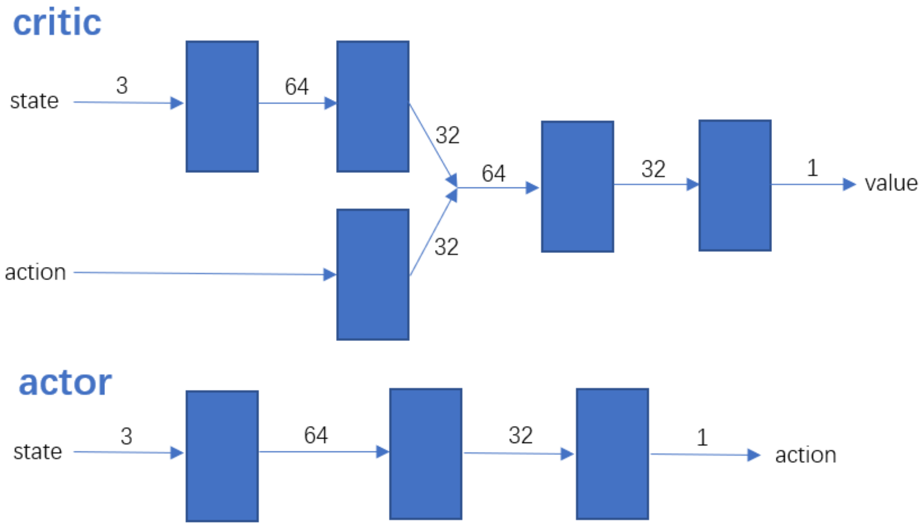
```
            self.memory[self.len] = transition
        else:
            self.len += 1
            self.memory.append(transition)
```

3. Neural network

```
class Critic(nn.Module):

    def __init__(self):
        super(Critic, self).__init__()
        self.state_dim = S_DIM
        self.action_dim = A_DIM

        self.fcs1 = nn.Linear(self.state_dim,64)
        self.fcs2 = nn.Linear(64,32)
        self.fca1 = nn.Linear(self.action_dim,32)
        self.fc2 = nn.Linear(64,32)
        self.fc3 = nn.Linear(32,1)

    def forward(self, state, action):
        s1 = F.relu(self.fcs1(state))
        s2 = F.relu(self.fcs2(s1))
        a1 = F.relu(self.fca1(action))
        x = torch.cat((s2,a1),dim=1)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
class Actor(nn.Module):

    def __init__(self):
        super(Actor, self).__init__()
        self.state_dim = S_DIM
        self.action_dim = A_DIM
        self.action_lim = A_MAX

        self.fc1 = nn.Linear(self.state_dim,64)
        self.fc2 = nn.Linear(64,32)
        self.fc3 = nn.Linear(32,self.action_dim)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        action = torch.tanh(self.fc3(x))
        action = action * self.action_lim
        return action
```

The architecture of critic network and actor network:



4. Agent

```
# return a noise
# Based on http://math.stackexchange.com/questions/1287634/implementing-ornstein-
uhlenbeck-in-matlab
class OrnsteinUhlenbeckActionNoise:

    def __init__(self, mu = 0, theta = 0.15, sigma = 0.2):
        self.action_dim = A_DIM
        self.mu = mu
        self.theta = theta
        self.sigma = sigma
        self.X = np.ones(self.action_dim) * self.mu

    def reset(self):
        self.X = np.ones(self.action_dim) * self.mu

    def sample(self):
        dx = self.theta * (self.mu - self.X)
        dx = dx + self.sigma * np.random.randn(len(self.X))
        self.X = self.X + dx
        return self.X
```

This is a noise function taken from https://github.com/vy007vikas/PyTorch-ActorCriticRL based on http://math.stackexchange.com/questions/1287634/implementing-ornstein-uhlenbeck-in-matlab .

```
class Agent:

    def __init__(self, memory):
        # some parameters
        self.state_dim = S_DIM
        self.action_dim = A_DIM
        self.action_lim = A_MAX
        self.tau = 0.001

        self.lr = 0.001
```

```
        self.gamma = 0.99
        self.memory = memory

        self.noise = OrnsteinUhlenbeckActionNoise()

        self.actor = Actor()
        self.target_actor = Actor()
        self.actor_optimizer = optim.Adam(self.actor.parameters(),lr=self.lr)

        self.critic = Critic()
        self.target_critic = Critic()
        self.critic_optimizer = optim.Adam(self.critic.parameters(),lr=self.lr)

        # copy parameters from actor to target_actor
        self.hard_update(self.target_actor, self.actor)
        # copy parameters from critic to target_critic
        self.hard_update(self.target_critic, self.critic)

    # get an action with noise
    def get_exploration_action(self, state):
        state = torch.Tensor(state)
        action = self.actor.forward(state).detach()
        new_action = action + torch.Tensor(self.noise.sample() * self.action_lim)
        return new_action.numpy()

    # copy parameters from source network to target network
    def hard_update(self, target, source):
        for target_param, param in zip(target.parameters(), source.parameters()):
            target_param.data.copy_(param.data)

    # target <= tau * source + (1 - tau) * target
    def soft_update(self, target, source):
        for target_param, param in zip(target.parameters(), source.parameters()):
            target_param.data.copy_(target_param.data * (1.0 - self.tau) + param.data *
self.tau)

    # update neural network
    def learn(self):
        # get a batch of transition from memory
        s1,a1,r1,s2 = self.memory.sample()
        s1 = Variable(s1)
        a1 = Variable(a1)
        r1 = Variable(r1)
        s2 = Variable(s2)

        # optimize critic net
        # a2 = mu'(s2)
        a2 = self.target_actor.forward(s2).detach()
        # next_val = Q'(s2, a2)
        next_val = self.target_critic.forward(s2, a2).detach().view(-1)
        # y = r + gamma * Q'(s2, a2)
        y_expected = r1 + self.gamma * next_val

        # y_predicted = Q(s1, a1)
```

```
        y_predicted = self.critic.forward(s1, a1).view(-1)

        loss_critic = F.smooth_l1_loss(y_predicted, y_expected)

        self.critic_optimizer.zero_grad()
        loss_critic.backward()
        self.critic_optimizer.step()

        # optimize actor net
        pred_a1 = self.actor.forward(s1)
        loss_actor = -1 * torch.sum(self.critic.forward(s1, pred_a1))
        self.actor_optimizer.zero_grad()
        loss_actor.backward()
        self.actor_optimizer.step()

        # target <= tau * source + (1 - tau) * target
        self.soft_update(self.target_actor, self.actor)
        self.soft_update(self.target_critic, self.critic)
```

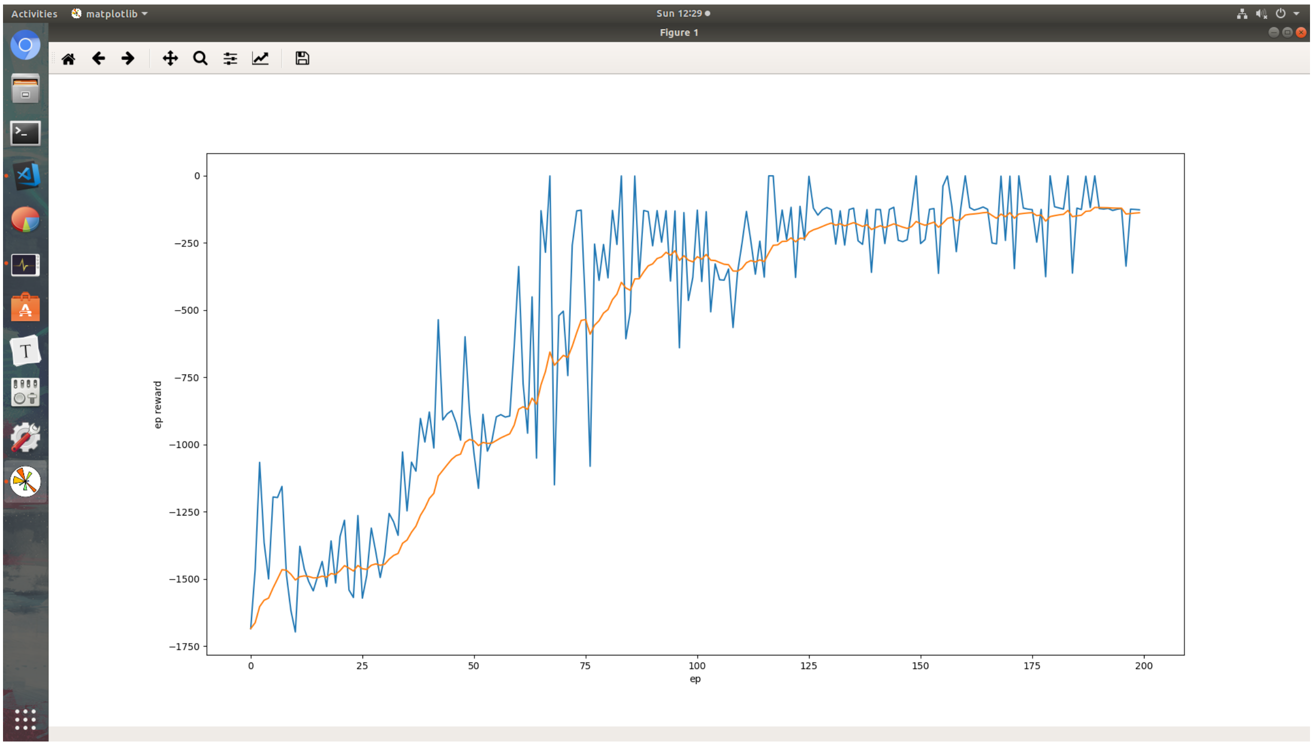
Agent.learn() is the main function of updating network.

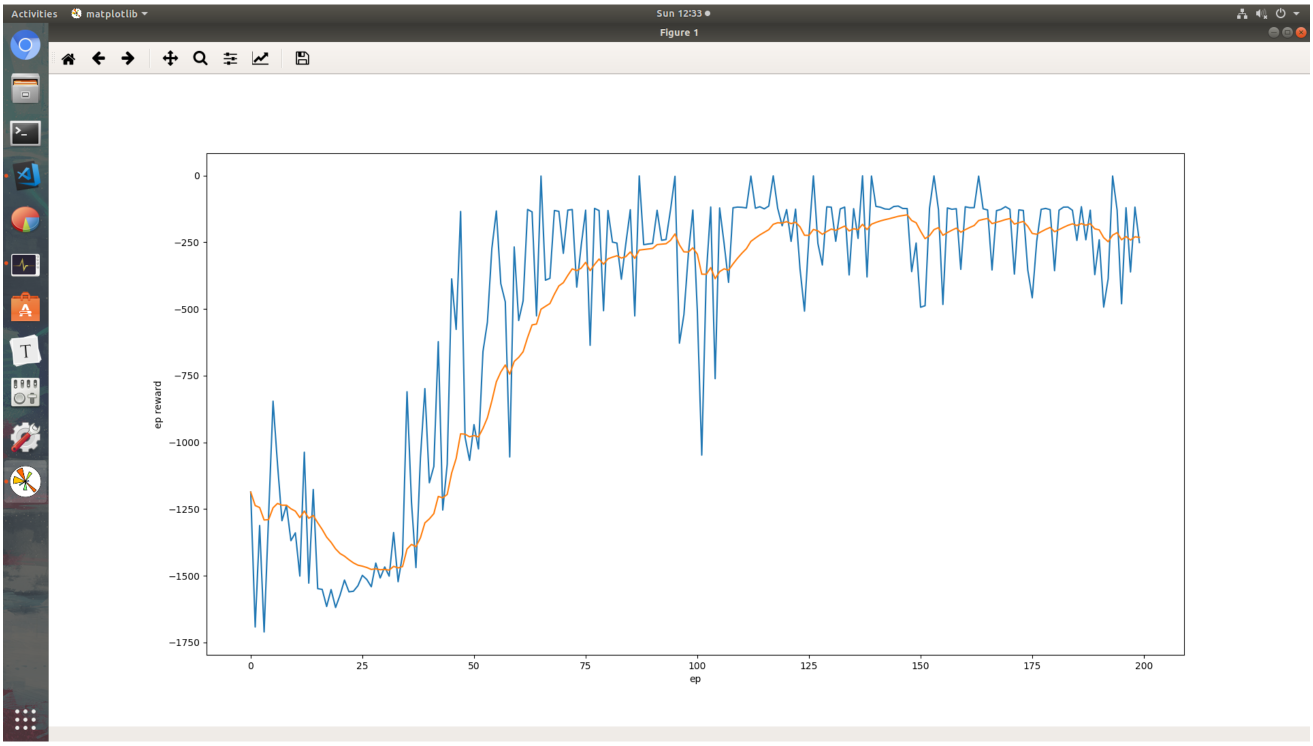
5. Main

```
def main():
    # initialize
    memory = Memory()
    agent = Agent(memory)
    res = []
    avg_res = []
    for episodes in range(MAX_EPISODES):
        state = env.reset()
        ep_r = 0
        for steps in range(MAX_STEPS):
            # show 1 episode every 10 episodes
            if episodes % 10 == 0:
                env.render()
            # get action
            action = agent.get_exploration_action(state)
            # take action
            next_state, reward, _, _ = env.step(action)
            # push transition into memory
            memory.push(state, action, reward, next_state)
            # start learn after 2 episodes
            # to fill memory at the beginning
            if episodes > 2:
                agent.learn()

            ep_r += reward
            state = next_state

        res.append(ep_r)
        # avg_res is average reward of the last 10 episodes approximately
        if avg_res:
```

```
        avg_res.append(avg_res[-1] * 0.9 + ep_r * 0.1)
    else:
        avg_res.append(ep_r)
    print("Ep:", episodes, "| Ep_r: %d" % ep_r)
# draw chart
plt.ion()
plt.figure()
plt.plot(res)
plt.plot(avg_res)
plt.ylabel("ep reward")
plt.xlabel("ep")
plt.ioff()
plt.show()
```

6. Result and conclusion





I think DDPG does better than A3C.

The blue line is reward of every episode.

The yellow line is average reward of the last 10 episodes.

The policy can swing the pendulum up and keep it upright as soon as possible.

And DDPG only needs about 120 episodes to converge.

Reference

https://github.com/MorvanZhou/pytorch-A3C

https://github.com/vy007vikas/PyTorch-ActorCriticRL