Report of Assignment 3

717030210005 Zhexin Jin

1. Packages and Modules

```
import gym
import numpy as np
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

import matplotlib
import matplotlib.pyplot as plt
```

2. Environment

```
class Env:

    def __init__(self):
        self.env = gym.make('MountainCar-v0').unwrapped
        # maximal number of steps in one episode
        self.max_step = 500

    def step(self, action):
        state, reward, done, _ = self.env.step(action)
        # if reach the goal, reward = 1000
        # else, reward = (h - 0.1) * 10
        if done:
            reward = 1000
        else:
            reward = (self.height(state[0])-0.1) * 10
        return state, reward, done, _

    def reset(self):
        return self.env.reset()

    def height(self, x):
        # return the height
        return np.sin(3 * x) * 0.45 + 0.55

    def render(self):
        return self.env.render()
```

```
    def close(self):
        return self.env.close()
```

This is the definition of Env.

To ease the description, I add some comments in the code.

3. Agent

```
class Agent:

    def __init__(self):
        # some parameters
        self.epsilon = 0.2
        self.gamma = 0.99
        self.lr = 0.0001
        # action_space contains the actions
        self.action_space = [0,1,2]
        # the state has two elements
        self.n_state = 2
        # before starting e-greedy policy,
        # generate 1000 pieces of data randomly
        self.n_random_learn = 1000
        # update the net every 4 steps
        self.learn_interval = 4
        # update the target net every 5 steps
        self.target_update_interval = 5
        # show the car's movement every 50 episodes
        self.show_interval = 50
        # action-value network
        self.net = NET()
        # target action-value network
        self.target_net = NET()
        # choose Adam as optimizer
        self.optimizer = torch.optim.Adam(self.net.parameters(), lr=self.lr)

    # if e_greedy == True, choose an action by e-greedy policy
    # else, choose an action by greedy policy
    def get_action(self, state, e_greedy = True):
        if e_greedy:
            if random.random() <= self.epsilon:
                return random.choice(self.action_space)
            else:
                with torch.no_grad():
                    # transform type
                    var = Variable(torch.FloatTensor(state.reshape(1,self.n_state)))
                # return the action which has the max value
                return self.net(var).max(1)[1].data[0].item()
        else:
            with torch.no_grad():
                var = Variable(torch.FloatTensor(state.reshape(1,self.n_state)))
            return self.net(var).max(1)[1].data[0].item()
```

```
    # update net
    def learn(self, memory):
        # get 32 samples from memory
        batch = memory.sample()
        # separate states, actions, rewards, next_states and dones
        [states, actions, rewards, next_states, dones] = zip(*batch)
        # transform type
        state_batch = Variable(torch.Tensor(states))
        action_batch = Variable(torch.LongTensor(actions))
        reward_batch = Variable(torch.Tensor(rewards))
        with torch.no_grad():
            next_state_batch = Variable(torch.Tensor(next_states))

        # get Q's value
        q = self.net(state_batch).gather(1, action_batch.view(-1,1)).view(-1)
        # get target_Q's value
        target_q = self.target_net(next_state_batch).max(1)[0]

        # calculate loss
        with torch.no_grad():
            y = (target_q * self.gamma) + reward_batch
        loss = F.mse_loss(q, y)

        # gradient descent
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    # update target net
    def target_update(self):
        # target_net's state <= net's state
        self.target_net.load_state_dict(self.net.state_dict())
```

The core code of DQN is in this part.

I think the comments are clear enough.

4. NET

```
class NET(nn.Module):

    def __init__(self):
        super().__init__()
        self.hidden_size = 128
        self.fc1 = nn.Linear(2, self.hidden_size)
        self.fc2 = nn.Linear(self.hidden_size, self.hidden_size)
        self.fc3 = nn.Linear(self.hidden_size, 3)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x))
        x = F.leaky_relu(self.fc2(x))
        x = F.leaky_relu(self.fc3(x))
```

```
        return x
```

This is the architecture of the network I use.

5. Memory

```
class Memory:

    def __init__(self):
        self.capacity = 100000
        self.memory = []
        self.len = 0
        self.batch_size = 32

    def push(self, transition):
        if self.len > self.capacity:
            self.memory[self.len % self.capacity] = transition
        else:
            self.len += 1
            self.memory.append(transition)

    def sample(self):
        return random.sample(self.memory, self.batch_size)
```

This is the definition of Memory.

6. Function show()

```
def show():
    global env, memory, agent
    state = env.reset()
    for n_step in range(1, env.max_step + 1):
        env.render()
        action = agent.get_action(state, e_greedy=False)
        state, _, done, _ = env.step(action)
        if done:
            break
    if done:
        print('Success!', 'Step:', n_step)
    else:
        print('Fail!', 'Step:', n_step)
```

This function will show the movement of car using current policy.

7. Initialize

```
# init
env = Env()
memory = Memory()
agent = Agent()
```

```
# fill memory
print('Fill memory!')
while memory.len < 1000:
    # reset the env
    state = env.reset()
    for n_step in range(1, env.max_step + 1):
        # get action randomly
        action = random.choice(agent.action_space)
        # take action
        next_state, reward, done, _ = env.step(action)
        # push the transition into memory
        memory.push([state, action, reward, next_state, done])
        # state <= next_state
        state = next_state
        if done:
            break
    print("\rPush: {} / {}".format(memory.len, 1000), end='', flush=True)
print('\nFinished!')

# random learn
print('Random learn!')
for n_learn in range(1, agent.n_random_learn + 1):
    print("\rLearn: {} / {}".format(n_learn, agent.n_random_learn), end='', flush=True)
    # update net
    agent.learn(memory)
    if n_learn % agent.target_update_interval == 0:
        # update target_net every 5 times
        agent.target_update()
print('\nFinished!')
```

1. Initialize env, memory and agent.
2. Generate 1000 transitions and push them into memory.
3. Update net and target_net.

The third part seems unnecessary, but it really affects the result. I will discuss about it in the last part.

8. DQN

```
# DQN learn
print('DQN learn!')
# plt for drawing
plt.ion()
plt.figure()
# record the max x coordinate of every episode
# x >= 0.5 means that the car reachs the goal
episode_xs = []
# the number of episodes
n_episode = 0
# the number of all steps
global_n_step = 0
# the number of continuous successful episodes
success_count = 0
# the number of all successful episodes
```

```
success_total = 0
# the success rate = success_total / n_episode
success_rate = []
# the number of continuous failed episodes
fail_count = 0
# loop forever
while True:
    # reset the env
    state = env.reset()
    n_episode += 1
    # max_x records the max x coordinate of this episode
    max_x = -float('inf')
    for n_step in range(1, env.max_step + 1):
        # get action using e-greedy policy
        action = agent.get_action(state)
        # take action
        next_state , reward , done, _ = env.step(action)

        # update max_x
        if max_x < next_state[0]:
            max_x = next_state[0]

        # push the new transition into memory
        memory.push([state, action, reward, next_state, done])
        # state <= next_state
        state = next_state

        # update net every 4 steps
        if global_n_step % agent.learn_interval == 0:
            agent.learn(memory)
        # update target_net every 5 steps
        if global_n_step % agent.target_update_interval == 0:
            agent.target_update()

        if done or n_step >= env.max_step:
            # if succeed
            if next_state[0] >= 0.5:
                fail_count = 0
                success_count += 1
                success_total += 1
                rate = success_total / n_episode
                print("Episode: {}   Success: {}   Success rate:
{}".format(n_episode,success_count,rate))
            # if fail
            else:
                success_count = 0
                fail_count += 1
                rate = success_total / n_episode
                print("Episode: {}   Fail: {}   Success rate:
{}".format(n_episode,fail_count,rate))
            episode_xs.append(max_x)
            success_rate.append(rate)

            # draw
```

```
            plt.clf()
            xs_t = torch.Tensor(episode_xs)
            rate_t = torch.Tensor(success_rate)
            plt.title('DQN')
            plt.xlabel('Episode')
            plt.ylabel('X and Rate')
            plt.plot(xs_t.numpy())
            plt.plot(rate_t.numpy())
            plt.pause(0.001)
            break
    # show every 50 episodes
    if n_episode % agent.show_interval == 0:
        show()

plt.ioff()
plt.show()
```

This is the main loop of DQN.

You will see a chart.

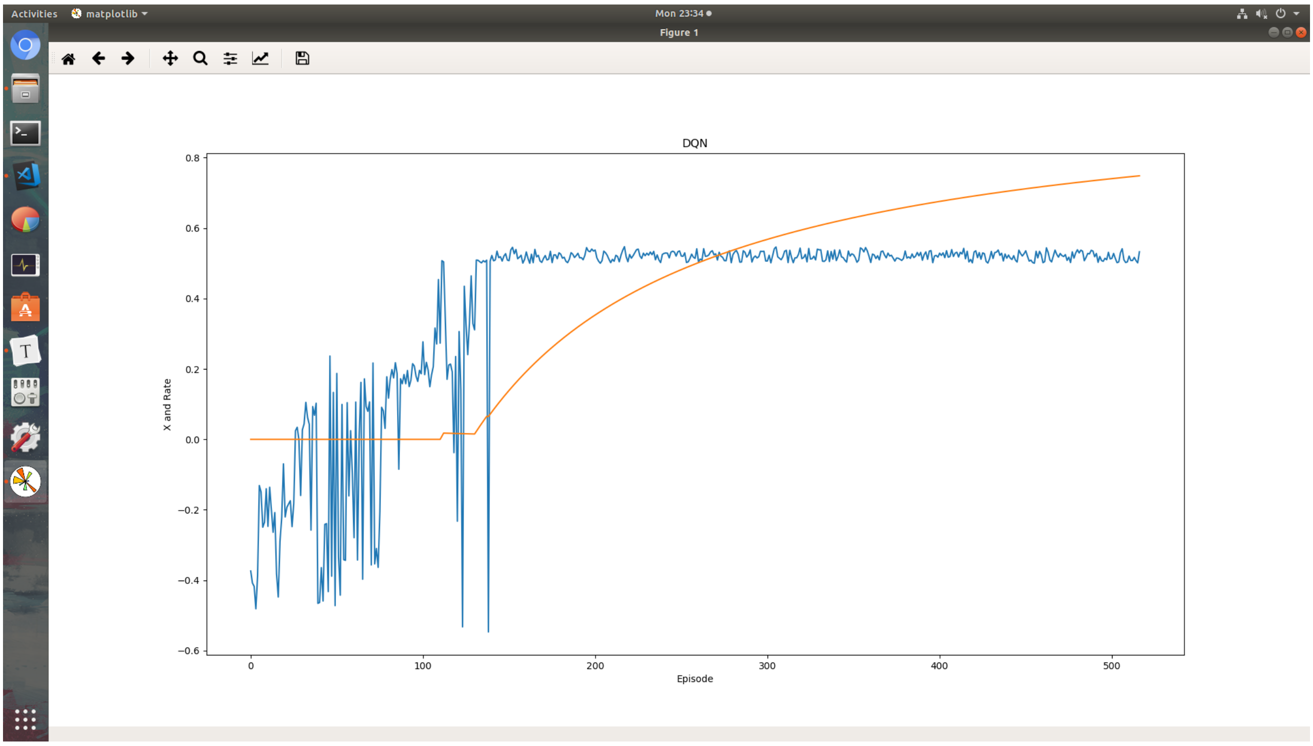
The blue line shows the learning effect of DQN. If x >= 0.5, this episode is successful.

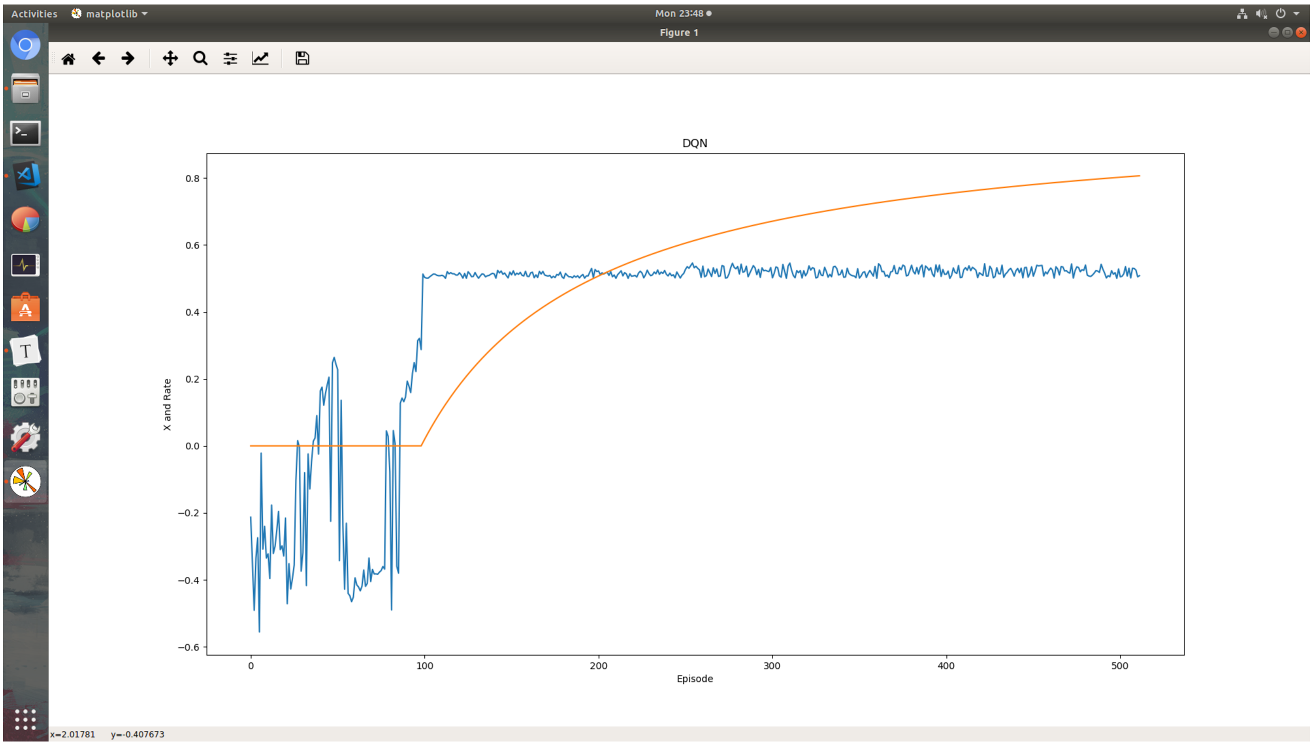
The yellow line shows the success rate.

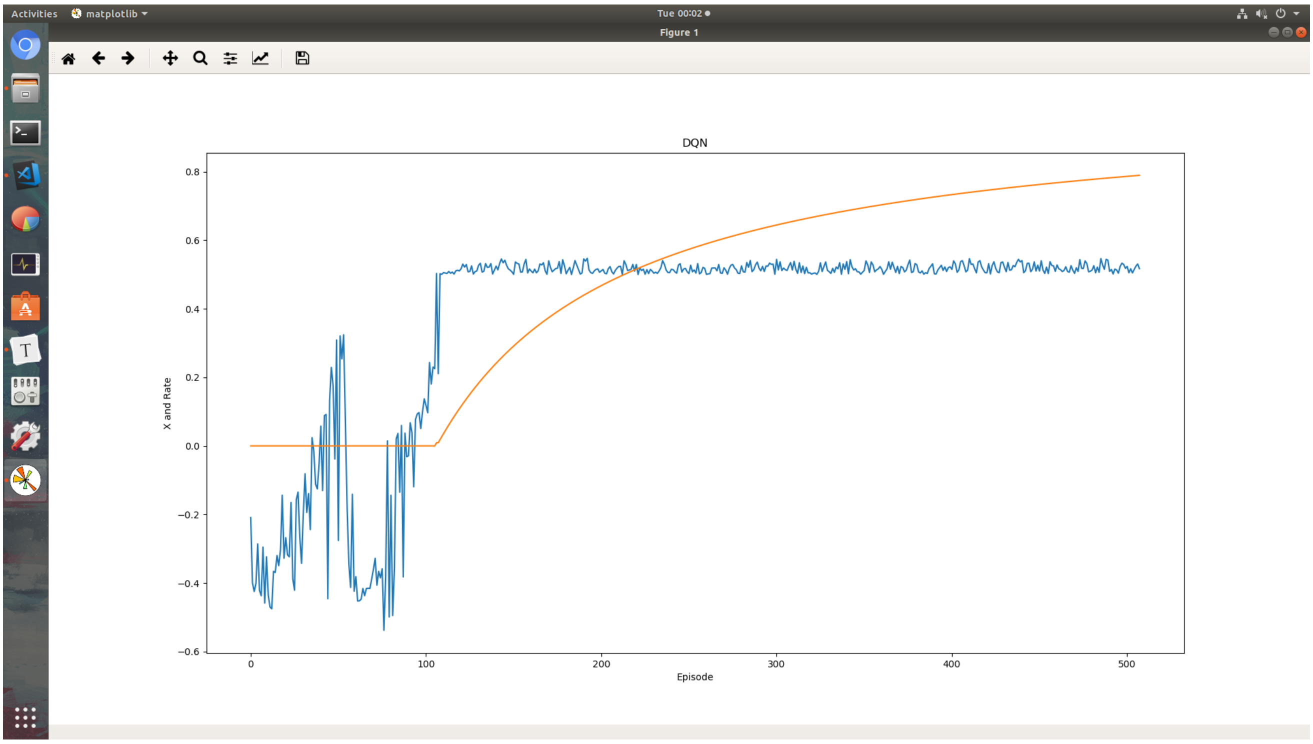
9. Result and conclusion

I tested several times, and it always showed the good outcome.

Here are some charts.

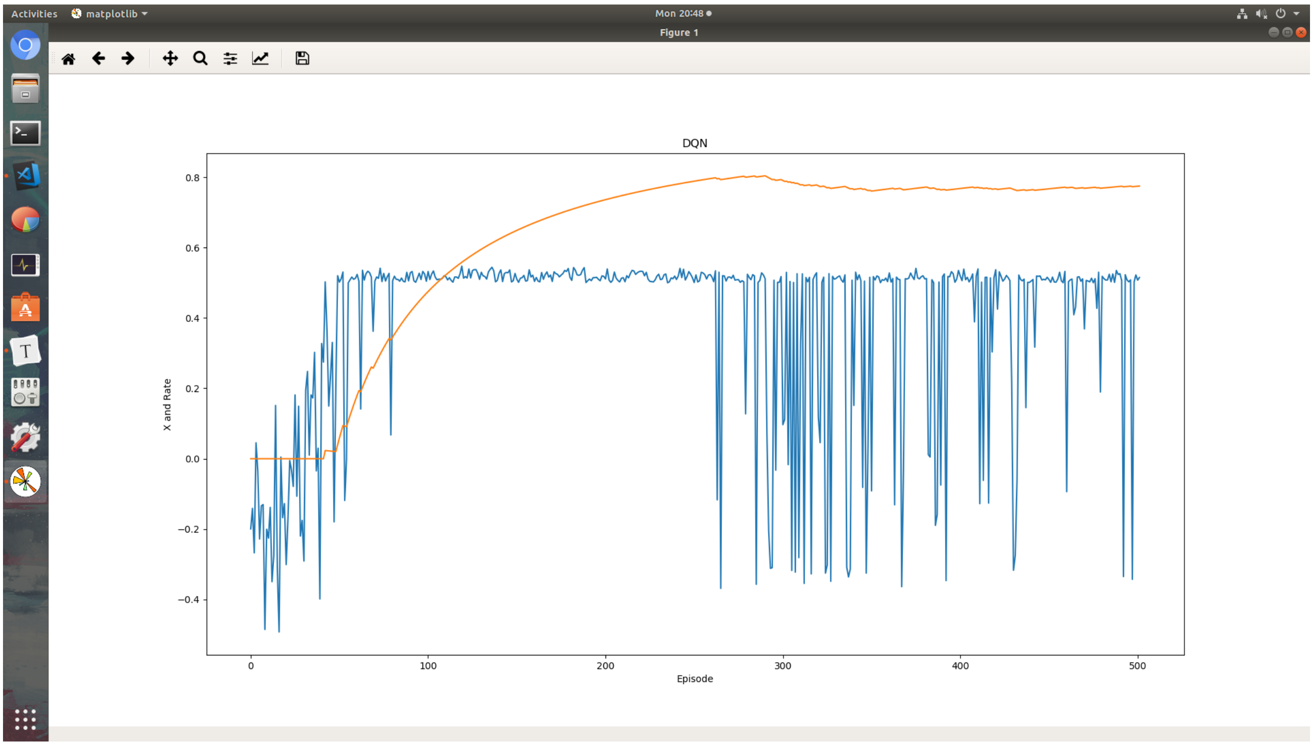


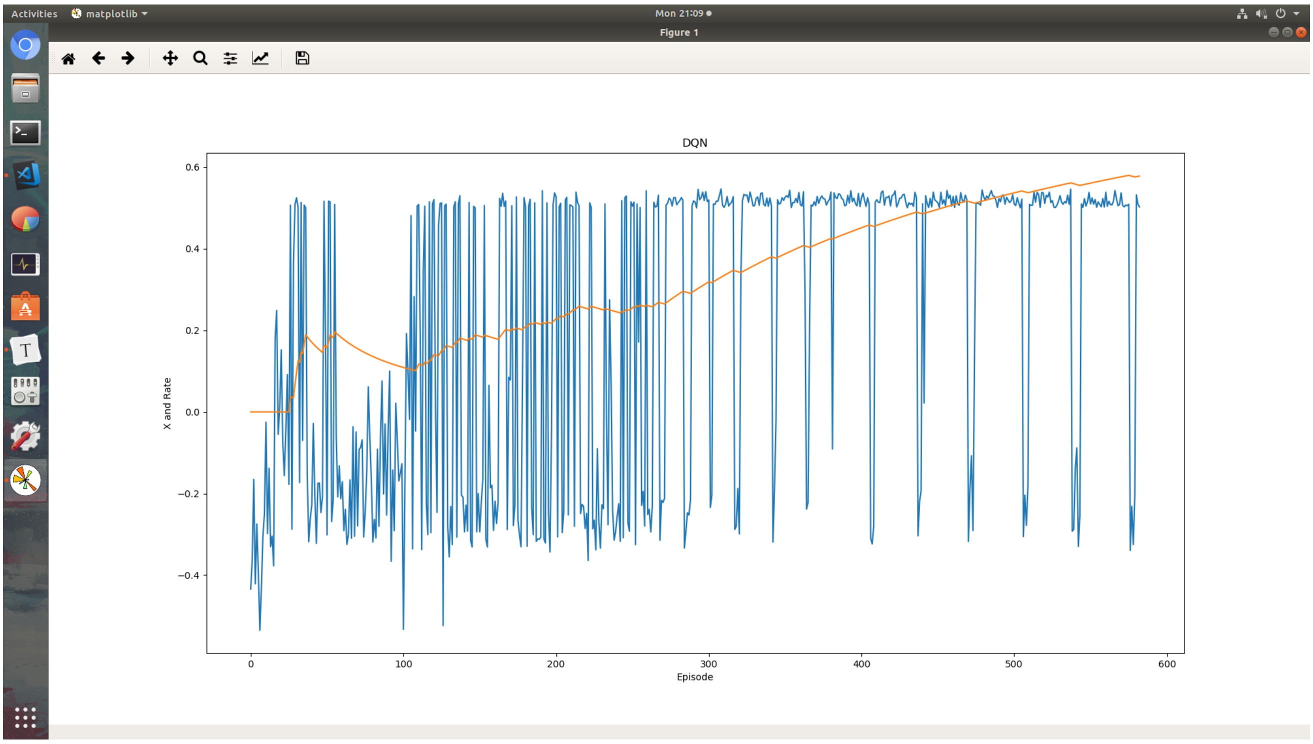


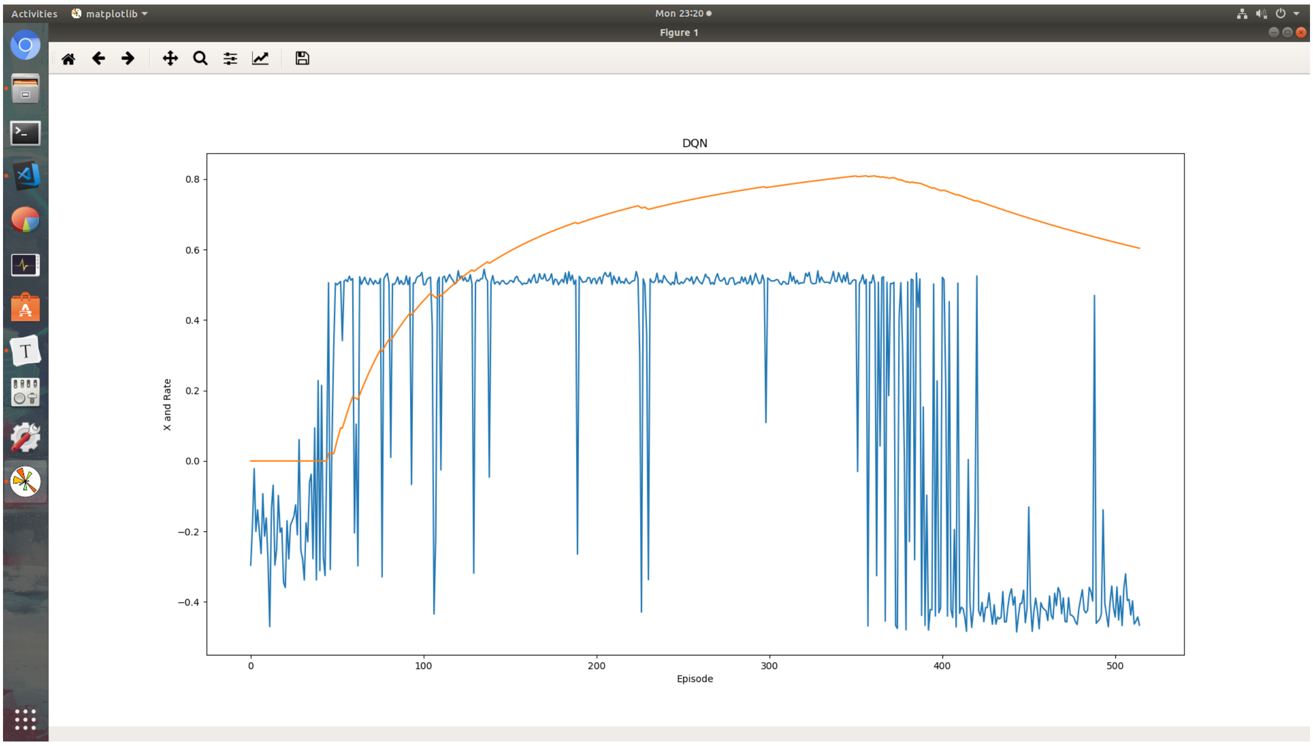


I have mentioned the updating of net before the main loop.

It's necessary because if I simply remove this part, I will get this kind of outcome below.


Activities
matplotlib
Mon 20:48
Figure 1
DQN
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
X and Rate
0
100
200
300
400
500
Episode


Activities
matplotlib
Mon 21:09
Figure 1
DQN
0.6
0.4
0.2
0.0
−0.2
−0.4
X and Rate
0
100
200
300
400
500
600
Episode



Some were hard to converge, some converged at first and finally diverged.

I don't know why.

I guess it's because if I let the net converge too quickly (remove the updating of net before the main loop), it might find a second-best solution instead of the best solution. And as a result, it will take a hard time to find the best solution (the real convergence).