基于 DQN 算法的平衡小车控制

姓名：林奇峰　　学号：19110977

1　摘要

平衡小车的控制问题（CartPole）是一个经典的强化学习问题。本文利用 DQN 算法来控制平衡小车的移动使其上面的杆维持竖直的状态。实验结果表明 DQN 算法可以学习到平衡小车的控制策略。本文也对训练过程中出现的现象和问题进行了分析，并认为 DQN 算法可以对连续状态空间进行一定的放大来获得比较好的效果。

2　引言

平衡小车的控制问题是一个经典的强化学习问题。在这个问题中，平衡小车上面的杆需要在小车移动的过程中保持竖直的状态，而小车的移动方向会影响杆的倾斜方向。因此，一个好的策略需要不断地控制小车的移动方向来使得杆保持竖直，如图 1所示：

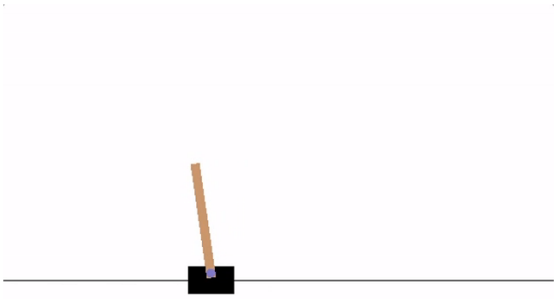


图 1: CartPole 示例

CartPole 问题可以建模成一个马尔可夫决策问题(Markov Process Decision, 简称 MDP)，因为控制小车的移动方向只与当前状态有关，与更早之前的状态和决策无关。

当前深度强化学习算法在许多应用上面获得了很多的成功，如 DQN 算法。因此本文研究了 DQN 算法在 CartPole 问题上的应用，并对训练过程进行了分析。实验结果表明 DQN 算法可以完成 CartPole 任务，且可以通过放大连续的状态向量来达到比较好的效果。

3 方法

强化学习是基于 MDP 框架的一类算法。MDP 可以用四元组 $< S, A, T, r >$ 表示。其中 S 代表有限的状态集合，A 代表动作空间，$T : S \times A \to S$ 代表状态转移函数，$r : S \times A \to \mathbb{R}$ 代表即时奖励。

DQN 算法是结合了经典的强化学习算法和深度学习算法。其方法主要是最小化 TD error：

$$\delta_t = r_t + \gamma Q'(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{1}$$

其中 t 代表时间戳，γ 为折扣因子。Q 函数是采用神经网络近似，为在线网络，而 Q' 函数需要每隔一段时间用 Q 函数的参数进行更新，为目标网络。因此，损失函数的公式为:

$$\mathcal{L} = [r + \gamma Q'(s', a') - Q(s, a)]^2 \tag{2}$$

另外，DQN 算法还采用了回放经验池 $\mathcal{D}$（Experience Replay Buffer）的方法来稳定训练过程。$\mathcal{D}$ 中的每个元素为四元组 (s, a, s', r)。

4 实验

实验采用了 gym 库中的 CartPole-V0 环境。其中状态表示为长度为 4 的向量，里面的元素分别为推车位置 $[-2.4, 2.4]$，车速 $[-\infty, \infty]$，杆子角度 $[-41.8, 41.8]$ 与杆子末端速度 $[-\infty, \infty]$。动作空间只有向左和向右，分别用 0 和 1 表示。每一步的移动都会得到 +1 的奖励。当连续 100 个 episode 的平均累积奖励超过 195 的时候，CartPole 问题就可以被视为解决了。

4.1 实验一：DQN 算法在 CartPole 上的初步实验

实验参数表 1所示。其中 lr 为学习率，num_hidden 为隐层神经元个数，batch_size 为一次训练使用的样本数量，max_size 为回放经验池的最大容量，gamma 为折扣因子，而 rate 为目标网络的更新速率。更新的公式为：

$$\theta' = rate * \theta + (1 - rate) * \theta' \tag{3}$$

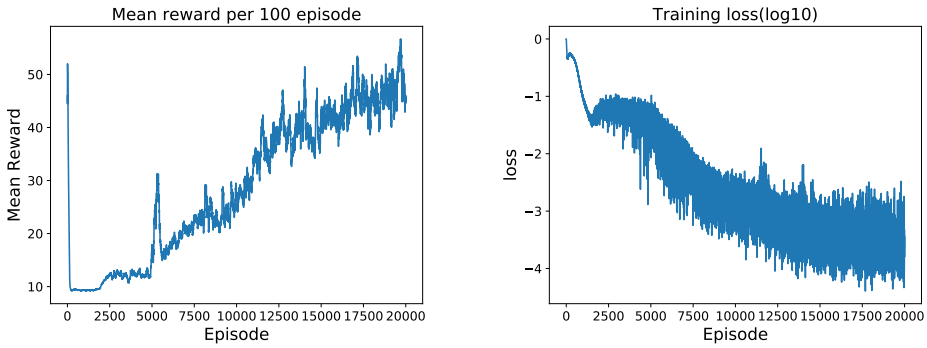
lr	num_hidden	batch_size	max_size	gamma	rate
1e-4	12	100	1e6	0.9	1e-3

表 1: 实验一参数设置

实验一共运行了 20000 个 episode。实验取每 100 个 Episode 的值的平均值作为实验结果，如图 2a-4b所示。

图 2a为奖励曲线，纵轴的数值为连续 100 个 Episode 的平均奖励。从曲线的上升趋势中我们可以看到 DQN 算法可以逐渐学习到比较好的策略来逐渐提高表现。

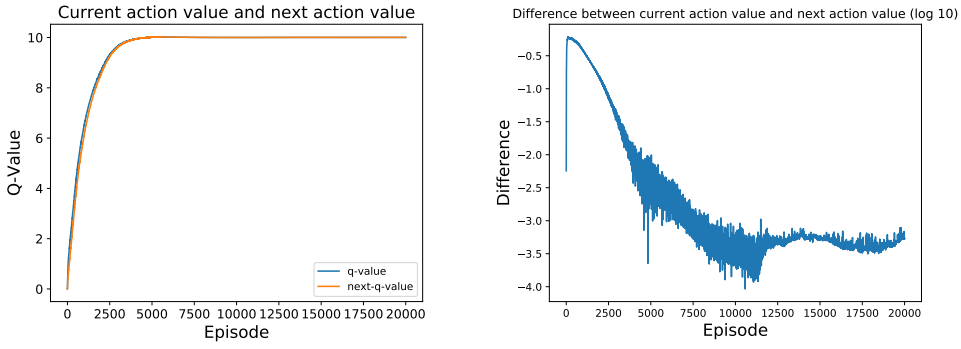
图 2b为损失函数曲线，纵轴的数值为进行开根号并对数处理后的损失函数数值。可以看到随着训练的进行，损失函数逐渐减小，而 TD error 也逐渐减小。这也符合直观的训练过程理解。



(a) 奖励曲线　　(b) 损失函数曲线

图 2: 奖励曲线和损失函数曲线

图 3a为当前 Q 值与下一步的 Q 值曲线图。首先，我们可以看到两者的曲线都在向上提升。这符合我们一开始的直观理解。因为 DQN 应该会引导智能体不断地选择 Q 值增大的方向学习。其次，我们可以看到 Q 和 Q' 的数值上很接近。按照一开始的理解，两者之间应该存在着近似 1 的差别。其实，一开始的时候，两者确实存在着近似 1 的差别。但随着训练的进行，两者越来越接近。我们把两者之间的差取对数以后可以得到图 3b的结果。可以看到，一开始两者之间的确是存在接近 1 的差别，但是后面就逐渐减小差别。这应该是归结于状态空间为连续空间。连续空间的变化可能不够剧烈和明显，从而导致 DQN 的数值变化也不明显。

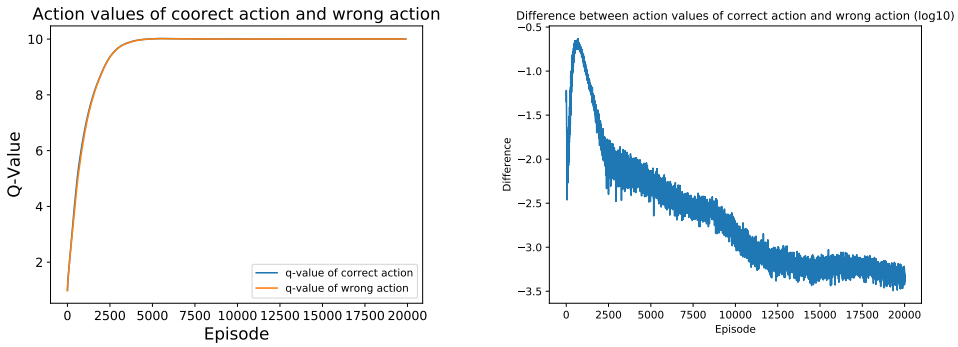


(a) 当前 Q 值与下一步的 Q 值曲线　　(b) 当前 Q 值与下一步的 Q 值之差的曲线

图 3: 当前 Q 值与下一步的 Q 值的相关曲线

图 4a为每一步正确动作和错误动作之间的 Q 值曲线。首先，正确动作的 Q 值曲线会上升比较符合直观的理解，但是错误动作的 Q 值曲线应该会一直下降，而实验结果却有点

出乎我的意料。我们可以看到两者之间的曲线在数值上也是非常接近的。为了更好地分析其中的变化，我们将两者之间的差进行了对数处理得到图 4b。可以看到一开始两者之间的差别是拉大的，但是随着训练的进行，两者之间的差别又缩小。这应该还是因为连续的状态空间导致变化对 DQN 算法来说不够明显和易于区别。



(a) 正确动作与错误动作的 Q 值曲线　　(b) 正确动作与错误动作的 Q 值之差的曲线

图 4: 正确动作与错误动作的 Q 值相关曲线

从上述结果我们可以看到对连续状态空间来说，我们可能需要对其变化进行一定的放大来使得状态之间的区分度更高一点。

4.2 实验二：状态放大对于 DQN 算法的影响

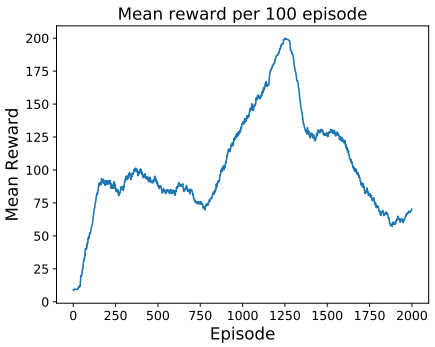
为了验证变化放大的分析，我对状态向量的数值乘上一个缩放因子 scale。上述实验一的结果可以看成设置为 scale=1.0 的结果。下面给出 scale=16.0 的结果。实验参数如表 2所示：

lr	num_hidden	batch_size	max_size	gamma	rate
1e-5	10	100	1e6	0.9	1e-3

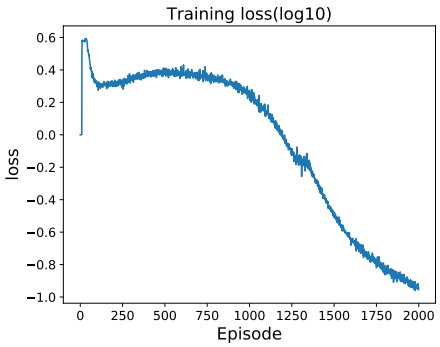
表 2: 实验二参数设置

本次实验只需要运行 2000 个 episode。实验结果如图 5a-7b所示。从图 5a可以看到经过状态放大，我们可以在 2000 个 episode 以内完成 CartPole 任务。但是 DQN 算法会存在过拟合的问题，需要一定的停止策略。从图 6a和 6b中我们可以看到训练的后期前后两步的 Q 值依然会逐渐缩小差距。这可能也是 DQN 算法过拟合的原因。从图 7a和 7b中，我们可以看到一开始正确动作和错误动作的差距会扩大，然后差距缩小到一定的地步的时候 DQN 算法会有比较好的结果。但是在训练后期由于过拟合两者的差距又会缩小。所以我们可以考虑在两者之间的差距开始缩写的时候停止 DQN 算法的训练。

因此，上述结果说明对于连续状态空间，进行程度的状态放大确实可以提高 DQN 算法的效果和减少训练时间。

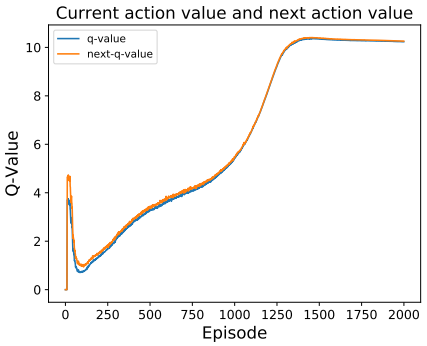


(a) 奖励曲线

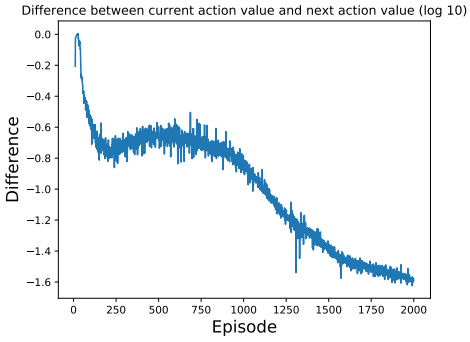


(b) 损失函数曲线

图 5: 奖励曲线和损失函数曲线

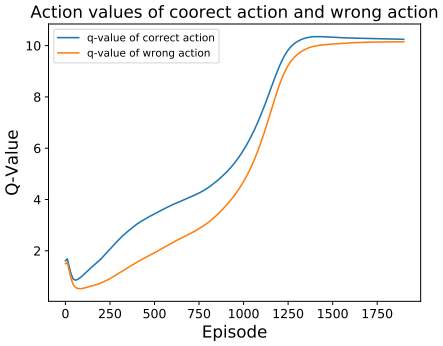


(a) 当前 Q 值与下一步的 Q 值曲线

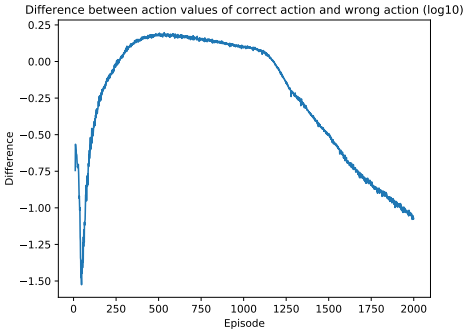


(b) 当前 Q 值与下一步的 Q 值之差的曲线

图 6: 当前 Q 值与下一步的 Q 值的相关曲线



(a) 正确动作与错误动作的 Q 值曲线



(b) 正确动作与错误动作的 Q 值之差的曲线

图 7: 正确动作与错误动作的 Q 值相关曲线

5 总结

本文研究了 DQN 算法在 CartPole 问题上的应用，并对训练过程中产生的问题进行了一定的分析。通过分析，我们认为连续状态空间对于 DQN 算法来说需要一定的处理才可以发挥出更好的效果，如状态放大。

6 源码

环境要求：pytorch, gym, numpy, matplotlib

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import gym
import numpy as np
import random
import matplotlib.pyplot as plt
import time

from collections import namedtuple
random.seed(0)
np.random.seed(0)
torch.manual_seed(0)

Transition = namedtuple('Transition', 'state, action, next_state, reward')

class MLP(nn.Module):
    def __init__(self, num_input, num_hidden, num_output):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(num_input, num_hidden)
        # self.fc2 = nn.Linear(num_hidden, num_hidden)
        self.fc3 = nn.Linear(num_hidden, num_output)

    def forward(self, x):
        x = self.fc1(x)
        # x = F.relu(x)
        # x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)

        return x

class DQNAgent(object):
    def __init__(self, n_state, n_action, args):
        self.online = MLP(n_state, args.num_hidden, n_action)
        self.target = MLP(n_state, args.num_hidden, n_action)
        self.args = args
        self.replay_buffer = ReplayBuffer(args.max_size)
        self.optimizer = torch.optim.Adam(self.online.parameters(), lr=args.lr)

        make_update_exp(self.online, self.target, rate=1.0)

    def act(self, state):
```

```
        output = self.online(torch.tensor(state[None], dtype=torch.float32))
        epsilon = 1e-6
        if random.random() < epsilon:
            action = random.randint(0, 1)
        else:
            action = torch.argmax(output, dim=1).numpy()[0]
        return action


    def update(self, transition):
        self.replay_buffer.add(transition)
        if len(self.replay_buffer) < self.args.batch_size:
            return None, None, None, None

        transitions = self.replay_buffer.sample(self.args.batch_size)
        state = torch.tensor(transitions.state, dtype=torch.float32)
        action = torch.tensor(transitions.action, dtype=torch.long).unsqueeze(1)
        next_state = torch.tensor(transitions.next_state, dtype=torch.float32)
        reward = torch.tensor(transitions.reward, dtype=torch.float32)

        q = torch.gather(self.online(state), dim=1, index=action)
        next_q = torch.max(self.target(next_state), dim=1)[0].detach()
        debug_q = [torch.mean(torch.gather(self.online(state), dim=1, index=action)).item(),
                   torch.mean(torch.gather(self.online(state), dim=1, index=(1-action))).
                                                                    item()]
        # debug_q = [torch.mean(q).item(),
        #              torch.mean(torch.gather(self.online(state), dim=1, index=(1-action))).
                                                    item()]

        td_error = reward + self.args.gamma * next_q - q


        loss = torch.mean(torch.pow(td_error, 2))

        self.optimizer.zero_grad()
        loss.backward()
        for param in self.online.parameters():
            param.grad.data.clamp_(-1, 1)
        self.optimizer.step()

        make_update_exp(self.online, self.target, rate=self.args.rate)



        return (loss.item(), torch.mean(q).item(), torch.mean(next_q).item(), debug_q)


class ReplayBuffer(object):
    def __init__(self, max_size):
        self.buffer = []
        self.max_size = int(max_size)
        self.index = 0

    def add(self, transition):
        if len(self.buffer) >= self.max_size:
            self.buffer[(self.index + 1) % self.max_size] = transition
        else:
            self.buffer.append(transition)
```

```
        self.index = (self.index + 1) % self.max_size

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

    def sample(self, batch_size):
        samples = random.sample(self.buffer, int(batch_size))
        transitions = Transition(*(zip(*samples)))

        return transitions


class Args(object):
    def __init__(self):
        self.num_hidden = 10
        self.batch_size = 100
        self.max_size = 1e6
        self.lr = 1e-5
        self.gamma = 0.9
        self.rate = 1e-3


def make_update_exp(source, target, rate=1e-2):
    """ Use values of parameters from the source model to update values of parameters from
                                                the target model. Each update just change
                                                values of paramters from the target model
                                                slightly, which aims to provide relative
                                                stable evaluation. Note that structures of
                                                 the two models should be the same.

    Parameters
    ----------
    source : torch.nn.Module
        The model which provides updated values of parameters.
    target : torch.nn.Module
        The model which receives updated values of paramters to update itself.
    """
    polyak = rate
    for tgt, src in zip(target.named_parameters(recurse=True), source.named_parameters(
                                                recurse=True)):
        assert src[0] == tgt[0] # The identifiers should be the same
        tgt[1].data = polyak * src[1].data + (1.0 - polyak) * tgt[1].data

def display(agent, env):
    display_epoch = 2
    for i in range(display_epoch):
        print('i:{}'.format(i))
        obs = env.reset()
        while True:
            action = agent.act(obs)
            _, _ , done, info = env.step(action)
            env.render()
            time.sleep(0.2)
            if done:
                obs = env.reset()
                break

def train():
```

```
env = gym.make('CartPole-v0')
env.seed(1)
n_state = env.observation_space.shape[0]
n_action = env.action_space.n

agent = DQNAgent(n_state, n_action, Args())


episode_reward = [0.0]
epoch = 0
losses = [0.0]
q_values = [0.0]
next_q_values = [0.0]
difference_q_next_q = [0.0]
debug_qs = [[0.0, 0.0]]
difference_q = [0.0]

obs = env.reset()
gap = 100
steps = 1
first_update = True
n_step = 1
max_epoch = 2000
start_time = time.time()
total_time = start_time
scale = 16.0

max_result = 0
while True:
    action = agent.act(obs * scale)
    next_obs, reward, done, _ = env.step(action)


    if done:
        reward_replay = reward
    else:
        reward_replay = reward

    transition = Transition(state=obs * scale, action=action, next_state=next_obs *
                                                    scale, reward=reward_replay)

    if done:
        obs = env.reset()
        epoch += 1
        n_step = 1
        episode_reward[-1] += reward
        if epoch % 500 == 0:
            print('Time elapses:{:.1f}; Total Time:{:.1f}min'.format(time.time()-
                                                            start_time, (time.time()-
                                                            total_time) / 60))
            start_time = time.time()
        if np.mean(episode_reward[-gap:]) > max_result:
            max_result = np.mean(episode_reward[-gap:])
        if epoch >= max_epoch or np.mean(episode_reward[-gap:]) >= 300:
            # display(agent, env)
            print('max result:{}'.format(max_result))
            break
        else:
```

```
            episode_reward.append(0.0)
            losses.append(0.0)
            q_values.append(0.0)
            next_q_values.append(0.0)
            difference_q_next_q.append(0.0)
            debug_qs.append([0.0, 0.0])
            difference_q.append(0.0)
    else:
        obs = next_obs
        episode_reward[-1] += reward
    loss, q_value, next_q_value, debug_q = agent.update(transition)
    if loss == None:
        losses[-1] =1.0
        q_values[-1] += 0.0
        next_q_values[-1] += 0.0
        debug_qs[-1][0] += 0.0
        debug_qs[-1][1] += 0.0
    else:
        if first_update:
            first_update = False
            print('First update: epoch {}; steps {}'.format(epoch + 1, steps + 1))
        losses[-1] = losses[-1] + (loss - losses[-1]) / n_step
        q_values[-1] = q_values[-1] + (q_value - q_values[-1]) / n_step
        next_q_values[-1] = next_q_values[-1] + (next_q_value - next_q_values[-1]) /
                                                    n_step
        difference_q_next_q[-1] = difference_q_next_q[-1] + (np.abs(q_value -
                                                    next_q_value) -
                                                    difference_q_next_q[-1]) / n_step
        debug_qs[-1][0] = debug_qs[-1][0] + (debug_q[0] - debug_qs[-1][0]) / n_step
        debug_qs[-1][1] = debug_qs[-1][1] + (debug_q[1] - debug_qs[-1][1]) / n_step
        difference_q[-1] = difference_q[-1] + (np.abs(debug_q[0] - debug_q[1]) -
                                                    difference_q[-1]) / n_step
        # losses[-1] += loss
        # q_values[-1] += q_value
    steps += 1
    n_step += 1

np.savetxt('episode_reward.txt', episode_reward, fmt='%.8lf', encoding='utf-8')
np.savetxt('losses.txt', losses, fmt='%.8lf', encoding='utf-8')
np.savetxt('q_values.txt', q_values, fmt='%.8lf', encoding='utf-8')
np.savetxt('next_q_values.txt', next_q_values, fmt='%.8lf', encoding='utf-8')
np.savetxt('debug_qs.txt', debug_qs, fmt='%.8lf', encoding='utf-8')
np.savetxt('difference.txt', difference_q_next_q, fmt='%.8lf', encoding='utf-8')

episode_reward = [np.mean(episode_reward[max(0,s-gap):s]) for s in range(0, len(
                                            episode_reward))]
# losses = [np.mean(losses[max(0,s-gap):s]) for s in range(0, len(losses))]
# q_values = [np.mean(q_values[max(0,s-gap):s]) for s in range(0, len(q_values))]
# next_q_values = [np.mean(next_q_values[max(0,s-gap):s]) for s in range(0, len(
                                            next_q_values))]
# difference_q_next_q = [np.mean(difference_q_next_q[max(0,s-gap):s]) for s in range(0,
                                            len(difference_q_next_q))]
# difference_q = [np.mean(difference_q[max(0,s-gap):s]) for s in range(0, len(
                                            difference_q))]

debug_q_correct = []
debug_q_wrong = []
for c, w in debug_qs:
```

```
        debug_q_correct.append(c)
        debug_q_wrong.append(w)
    debug_qs = [[np.mean(debug_q_correct[s-gap:s]), np.mean(debug_q_wrong[s-gap:s])] for s
                                                in range(gap, len(debug_qs))]
    print('total steps:{}'.format(steps))
    plt.subplot(321)
    plt.plot(episode_reward)
    plt.title('Mean reward per 100 episode')
    plt.subplot(322)
    plt.plot(np.log10(np.sqrt(np.abs(losses))))
    plt.title('Training loss(log10)')
    plt.subplot(323)
    plt.plot(q_values, label='q-value')
    plt.plot(next_q_values, label='next-q-value')
    plt.legend()
    plt.title('Current action value and next action value')
    plt.subplot(324)
    plt.plot(np.log10(np.abs(difference_q_next_q)))
    plt.title('Difference between current action value and next action value (log 10)')
    plt.subplot(325)
    plt.plot([x[0] for x in debug_qs], label='q-value of correct action')
    plt.plot([x[1] for x in debug_qs], label='q-value of wrong action')
    plt.title('Action values of coorect action and wrong action')
    plt.legend()
    plt.subplot(326)
    plt.plot(np.log10(np.abs(difference_q)))
    plt.title('Difference between action values of correct action and wrong action (log10)')
    plt.show()

    plt.clf()
    plt.plot(episode_reward)
    plt.title('Mean reward per 100 episode', fontsize=16)
    plt.tick_params(labelsize=12)
    plt.xlabel('Episode', fontsize=16)
    plt.ylabel('Mean Reward', fontsize=16)
    plt.savefig('episode.pdf')
    plt.clf()
    plt.plot(np.log10(np.sqrt(np.abs(losses))))
    plt.title('Training loss(log10)', fontsize=16)
    plt.xlabel('Episode', fontsize=16)
    plt.ylabel('loss', fontsize=16)
    plt.tick_params(labelsize=12)
    plt.savefig('loss.pdf')
    plt.clf()
    plt.plot(q_values, label='q-value')
    plt.plot(next_q_values, label='next-q-value')
    plt.legend()
    plt.title('Current action value and next action value', fontsize=16)
    plt.xlabel('Episode', fontsize=16)
    plt.ylabel('Q-Value', fontsize=16)
    plt.tick_params(labelsize=12)
    plt.savefig('cn.pdf', fontsize=16)
    plt.clf()
    plt.plot(np.log10(np.abs(difference_q_next_q)))
    plt.title('Difference between current action value and next action value (log 10)',
                                                fontsize=12)
    plt.xlabel('Episode', fontsize=16)
    plt.ylabel('Difference', fontsize=16)
```

```
    plt.tick_params(labelsize=10)
    plt.savefig('dcn.pdf')
    plt.clf()
    plt.plot([x[0] for x in debug_qs], label='q-value of correct action')
    plt.plot([x[1] for x in debug_qs], label='q-value of wrong action')
    plt.title('Action values of coorect action and wrong action', fontsize=16)
    plt.legend()
    plt.xlabel('Episode', fontsize=16)
    plt.ylabel('Q-Value', fontsize=16)
    plt.tick_params(labelsize=12)
    plt.savefig('cw.pdf')
    plt.clf()
    plt.plot(np.log10(np.abs(difference_q)))
    plt.title('Difference between action values of correct action and wrong action (log10)',
                                                    fontsize=11)
    plt.xlabel('Episode', fontsize=10)
    plt.ylabel('Difference', fontsize=10)
    plt.tick_params(labelsize=10)
    plt.savefig('dcw.pdf')


if __name__ == "__main__":
    train()
```