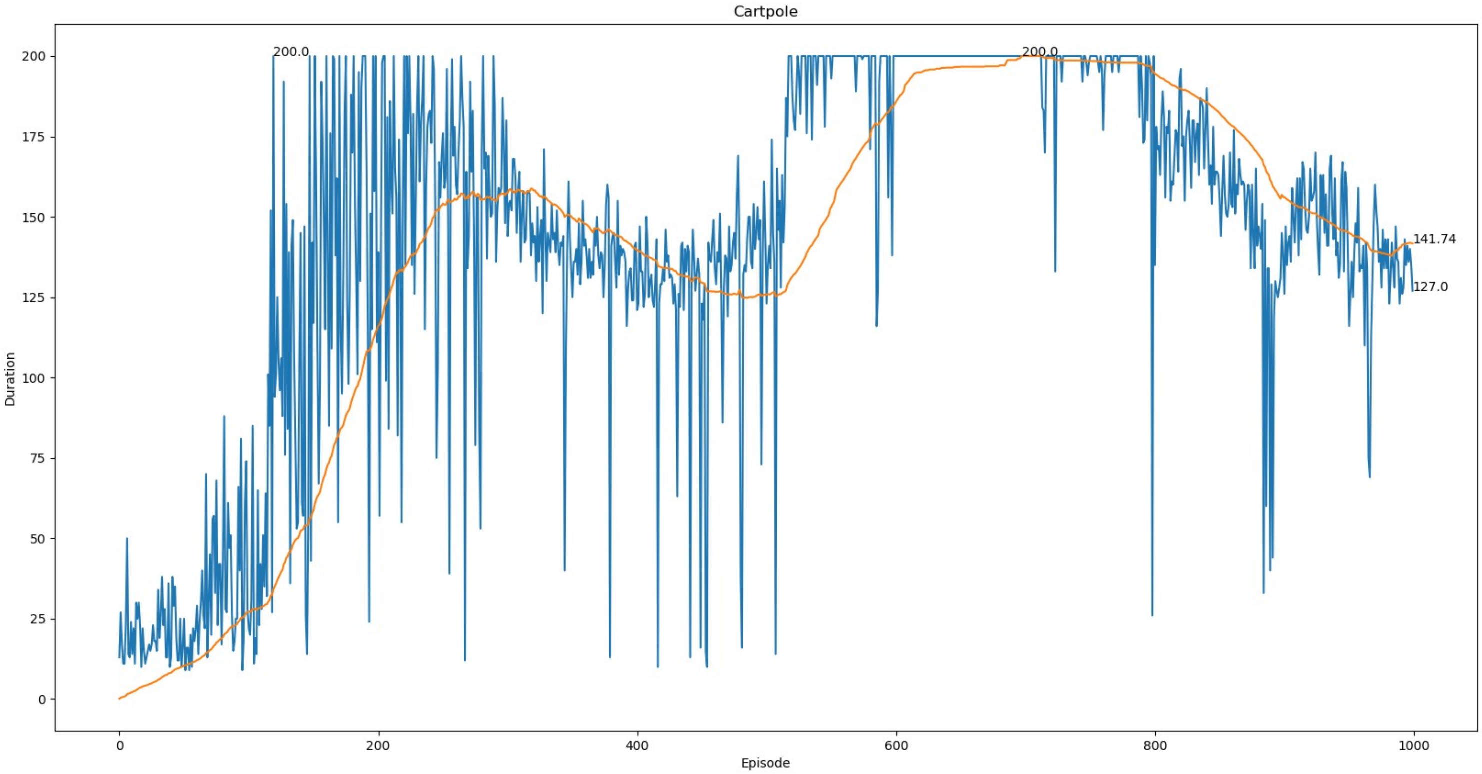
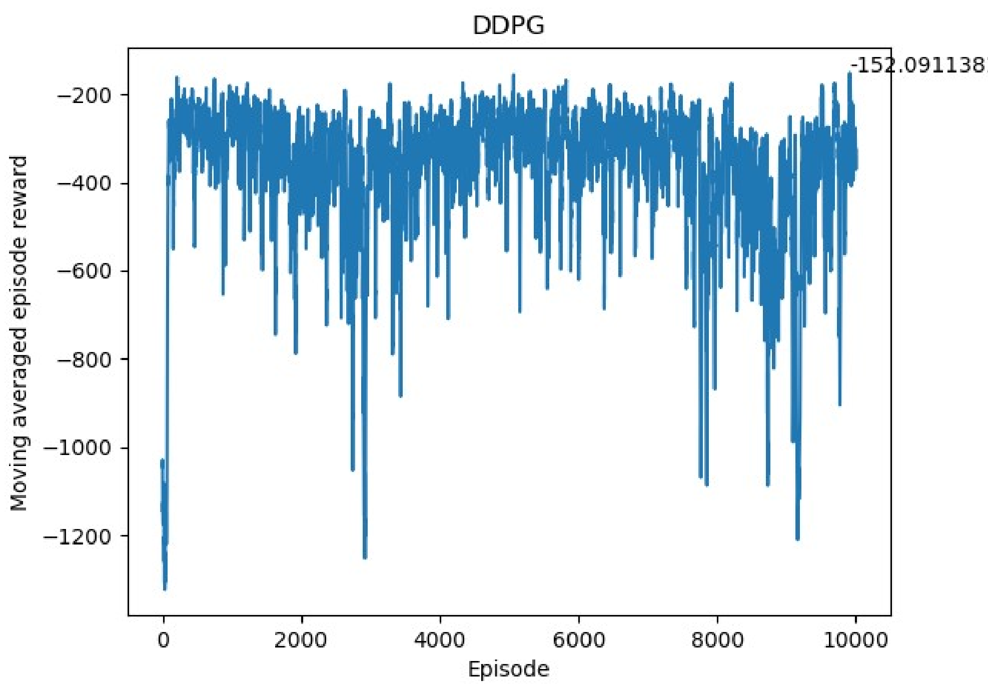
DLP_LAB8_0516003_李智嘉

一、 Plot at CartPole-v0 and Pendulum





二、Implement/adjustment of the network structure & each loss function

- CartPole-v0：Deep Q-learning with experience replay

```python
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)
    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

class Agent(object):
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)
        self.eval_net = Net(self.state_space_dim, 32, self.action_space_dim).to(device)
        self.target_net = Net(self.state_space_dim, 32, self.action_space_dim).to(device)
        self.target_net.load_state_dict(self.eval_net.state_dict())
```

在邊很單純的就是兩層 fully connect，中間夾了一個 relu activation 而已，latent vector 按照 spec 設成 32
比較特別的點是分成兩個 network，一個做主要的決定 action，一個來當作要 regression 的目標，而每過 K 個 iteration（這邊我設 50），就更新 regression 的目標成第一個 net 的 weight
如此就不會讓 regression 目標漂浮不定，比較好 train
而這邊 Loss 則是用 Mean Square Error Loss

- Pendulum-v0：DDPG

這邊則是用了叫做 Actor Critic 的技巧，Actor 負責決定動作，Critic 負責產生 Q 值
如此可以解決當 Action 是一個連續的值時，要找 maximum Q 的情況
而兩者的 model 也是很簡單，就是單純的 fully connect
Actor 的 fully connected 中間利用 relu 來連結，最後用 tanh 來輸出
Critic 中間用 relu，最後就不經過 activation function 了，直接輸出
而 Loss 一樣是用 Mean Square Error Loss

```python
class ActorNet(nn.Module):
    def __init__(self):
        super(ActorNet, self).__init__()
        self.fc = nn.Linear(3, 400)
        self.fc2 = nn.Linear(400, 300)
        self.mu_head = nn.Linear(300, 1)

    def forward(self, s):
        x = torch.relu(self.fc(s))
        x = torch.relu(self.fc2(x))
        u = torch.tanh(self.mu_head(x))
        return u


class CriticNet(nn.Module):
    def __init__(self):
        super(CriticNet, self).__init__()
        self.fc = nn.Linear(3, 400)
        self.fc2 = nn.Linear(401, 300)
        self.v_head = nn.Linear(300, 1)

    def forward(self, s, a):
        x = F.relu(self.fc(s))
        x = F.relu(self.fc2(torch.cat([x, a], dim=1)))
        state_value = self.v_head(x)
        return state_value
```

三、Implement the training process of deep Q-learning

```python
def learn(self):
    if (len(self.buffer)) < self.batch_size:
        return

    samples = random.sample(self.buffer, self.batch_size)
    s0, a0, r1, s1 = zip(*samples)
    s0 = torch.tensor(s0, dtype=torch.float).to(device)
    a0 = torch.tensor(a0, dtype=torch.long).view(self.batch_size, -1).to(device)
    r1 = torch.tensor(r1, dtype=torch.float).view(self.batch_size, -1).to(device)
    s1 = torch.tensor(s1, dtype=torch.float).to(device)

    y_true = r1 + self.gamma * torch.max(self.target_net(s1).detach(), dim=1)[0].view(self.batch_size, -1)
    y_pred = self.eval_net(s0).gather(1, a0)

    loss_fn = nn.MSELoss()
    loss = loss_fn(y_pred, y_true)

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

這邊一樣是用 TD learning 的技巧，不同的是 Q 值的計算變成了用 neural network 來取代
一開始先從歷史紀錄（buffer）sample 出 batch_size 份量的 episodes
predict 值會是經過 self.eval_net(s0)後，取這個 action(a0)後的值
而 true value 則是利用 target_net(s1)後最大的值，加上這步的 reward
下面就把這兩個值做 Mean Square Error Loss，再 back propogate 就可以了

四、Implement of epsilon-greedy action select method

```python
    def update_iter(self):
        self.steps += 1
        self.epsi *= self.decay
        if self.steps % self.copy_weight == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())

    def act(self, s0):
        if random.random() < self.epsi:
            a0 = random.randrange(self.action_space_dim)
        else:
            s0 =  torch.tensor(s0, dtype=torch.float).view(1,-1).to(device)
            a0 = torch.argmax(self.eval_net(s0)).item()
        return a0
```

這邊有兩個 function 來完成 epsilon-greedy，一個是 update_iter，每次 iteration 完後就會 call 這個 function，來更新目前 iteration 次數，以及 epsi threshold 乘上一個 decay 值
而在實際決定動作的時候，會先 random 一個數值，如果小於 epsi threshold 的話，就會隨機挑一個 action 做，如此就可以完成 epsilon-greedy action 了

五、The mechanism of critic updating

Sample random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from R

Set $y_i = \ r_i + \gamma Q'\left(s_{t+1}, \mu'\left(s_{t+1}|\theta^{\mu'}\right)|\theta^{Q'}\right)$

Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i\left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$

由於 critic 作用是 Q 值的產生，所以我們更新也只要考慮這部份就好了
我們會先對 buffer sample 一切 tranitions 出來做更新用
第二行是在計算 target value，分別是 reward 和 target network 產生的值
第三行就是把 target value 和 evaluation network 產生的 Q 值做 Mean Square Error Loss，然後再更新就好了
要注意的是第二行在計算 target value 時，會用到 actor 的 target network 來產生 action，在實做時這部份要小心不要更新到了

六、The mechanism of actor updating

Update the actor policy using the sampled gradient:

$$\nabla_{\theta^\mu}\mu|s_i \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$$

他的 loss 會是 sampled 的 Q 對 a 的 gradient，用 code 寫成的話會變成下面樣子

```
a_loss = -self.eval_cnet(s, self.eval_anet(s)).mean()
a_loss.backward()
```

actor 負責的是 action 的選擇，所以會希望他所選的 action 能盡量讓 critic 產生的 Q 值越大越好，而換成 loss 要越小越好的話就多加一個負號就可以了

七、How to calculate the gradients

對於 actor 的 gradient，前項 Q 對 a 的 gradient 可以看做是將 actor predict 的結果丟到 critic 後，對結果最 back propagate
後項則是 actor 產生的 action 結果做 gradient
所以整體來看可以看做是一連串下來做 back propagation 就可以了

八、How the code work

- CartPole-v0

首先先設定好 environment 以及 model 參數

```python
env = gym.make('CartPole-v0')
params = {
    'gamma': 0.95,
    'epsi_high': 1,
    'epsi_low': 0.001,
    'decay': 0.995,
    'lr': 0.0005,
    'capacity': 10000,
    'batch_size': 128,
    'copy_weight': 50,
    'state_space_dim': env.observation_space.shape[0],
    'action_space_dim': env.action_space.n
}
agent = Agent(**params)
agent.load_weight()
score = []
mean = []
finish = False
```

下方是主程式，每次 environment 都會 reset，然後因為我是在遠端機器，所以就不處理 X server 的問題了，直接把 render 關掉

每次都會讓 agent 對這個 state 做動作，丟回 env 後就記錄下來
然後讓 agent 去 learn 他

後面就是一些更新次數或 epsilon 以及紀錄結果的 code，就不詳細解釋了

```python
for episode in range(100):
    s0 = env.reset()
    total_reward = 1
    while True:
        # env.render()
        a0 = agent.act(s0)
        s1, r1, done, _ = env.step(a0)

        if done:
            r1 = -1

        agent.put(s0, a0, r1, s1)

        if done:
            break

        total_reward += r1
        s0 = s1
        agent.learn()
    agent.update_iter()
    # if total_reward==200 and not finish:
    #     agent.save_weight()
    #     print('save')
    # if sum(score[-100:])/100 == 200:
    #     finish = True
    #     print('avg score reach 200')
    score.append(total_reward)
    print(f'Episode {episode}: score {total_reward}, avg {sum(score[-100:])/100}')
    mean.append(sum(score[-100:])/100)
agent.plot(score, mean)
```

右圖是 Net 的架構以及用一個 Agent class 把 model 和訓練等 function 都包在一起

```python
device = torch.device('cuda:2')
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)
    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

class Agent(object):
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)
        self.eval_net = Net(self.state_space_dim, 32, self.action_space_dim).to(device)
        self.target_net = Net(self.state_space_dim, 32, self.action_space_dim).to(device)
        self.target_net.load_state_dict(self.eval_net.state_dict())

        self.optimizer = optim.Adam(self.eval_net.parameters(), lr=self.lr)
        self.buffer = []
        self.steps = 0
        self.epsi = self.epsi_high
        self.training = True
```

再來是每次 iteration 結束後，會更新 step 次數以及 epsilon 的數值，另外次數如果到達一定程度的話就更新 target network 的參數

```python
    def update_iter(self):
        self.steps += 1
        self.epsi *= self.decay
        if self.steps % self.copy_weight == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())

    def act(self, s0):
        if random.random() < self.epsi:
            a0 = random.randrange(self.action_space_dim)
        else:
            s0 =  torch.tensor(s0, dtype=torch.float).view(1,-1).to(device)
            a0 = torch.argmax(self.eval_net(s0)).item()
        return a0

    def put(self, *transition):
        if len( self.buffer)==self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)
```

```python
    def learn(self):
        if (len(self.buffer)) < self.batch_size:
            return

        if not self.training:
            return

        samples = random.sample(self.buffer, self.batch_size)
        s0, a0, r1, s1 = zip(*samples)
        s0 = torch.tensor(s0, dtype=torch.float).to(device)
        a0 = torch.tensor(a0, dtype=torch.long).view(self.batch_size, -1).to(device)
        r1 = torch.tensor(r1, dtype=torch.float).view(self.batch_size, -1).to(device)
        s1 = torch.tensor(s1, dtype=torch.float).to(device)

        y_true = r1 + self.gamma * torch.max(self.target_net(s1).detach(), dim=1)[0].view(self.batch_size, -1)
        y_pred = self.eval_net(s0).gather(1, a0)

        loss_fn = nn.MSELoss()
        loss = loss_fn(y_pred, y_true)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

最後是 learn，首先會先檢查 buffer 夠不夠 batch_size 的數量，再來就是 sample，然後做各種前處理，再做上面有說過的更新就好了

- Pendulum-v0

主程式如右，一樣先設定好 environment 環境，然後就開始跑 iteration
每次 iteration 都限制最多只有 200 回合，然後也是跑和存到 buffer，如果 buffer 是滿的話就更新參數

```python
def main():
    env = gym.make('Pendulum-v0')
    env.seed(args.seed)

    agent = Agent()

    training_records = []
    running_reward, running_q = -1000, 0
    for i_ep in range(10000):
        score = 0
        state = env.reset()

        for t in range(200):
            action = agent.select_action(state)
            state_, reward, done, _ = env.step(action)
            score += reward
            if args.render:
                env.render()
            agent.store_transition(Transition(state, action, (reward + 8) / 8, state_))
            state = state_
            if agent.memory.isfull:
                q = agent.update()
                running_q = 0.99 * running_q + 0.01 * q

        running_reward = running_reward * 0.9 + score * 0.1
        training_records.append(TrainingRecord(i_ep, running_reward))
```

```python
class Agent():
    max_grad_norm = 0.5

    def __init__(self):
        self.training_step = 0
        self.var = 1.
        self.eval_cnet, self.target_cnet = CriticNet().float().to(device), CriticNet().float().to(device)
        self.eval_anet, self.target_anet = ActorNet().float().to(device), ActorNet().float().to(device)
        self.memory = Memory(10000)
        self.optimizer_c = optim.Adam(self.eval_cnet.parameters(), lr=0.001)
        self.optimizer_a = optim.Adam(self.eval_anet.parameters(), lr=0.0001)
        self.Loss = nn.MSELoss()

    def select_action(self, state):
        state = torch.from_numpy(state).float().to(device).unsqueeze(0)
        mu = self.eval_anet(state)
        dist = Normal(mu, torch.tensor(self.var, dtype=torch.float).to(device))
        action = dist.sample()
        action.clamp(-2.0, 2.0)
        return (action.item(),)

    def save_param(self):
        torch.save(self.eval_anet.state_dict(), 'ddpg_anet_params.pkl')
        torch.save(self.eval_cnet.state_dict(), 'ddpg_cnet_params.pkl')

    def store_transition(self, transition):
        self.memory.update(transition)
```

上圖是 Agent，將訓練的東西都包在一起
對 Actor 和 Critic 上面的問題已經有看過了，這邊就直接省略說明
對於 Actor 和 Critic 一樣都會有 Evaluation net 和 Target net
Memory 則是用來存 episode history 的類別

selection action 裡面，為了讓 agent 能有一點變化，會是讓 actor 出來一個 mu，然後對此做一個 normal distribution 的抽樣，而其 variance 會逐步減小，最後為了不讓他衝過頭，會在對兩邊做 crop，超過或低於都直接變成最大最小值

```python
    def update(self):
        self.training_step += 1

        transitions = self.memory.sample(32)
        s = torch.tensor([t.s for t in transitions], dtype=torch.float).to(device)
        a = torch.tensor([t.a for t in transitions], dtype=torch.float).view(-1, 1).to(device)
        r = torch.tensor([t.r for t in transitions], dtype=torch.float).view(-1, 1).to(device)
        s_ = torch.tensor([t.s_ for t in transitions], dtype=torch.float).to(device)

        with torch.no_grad():
            q_target = r + args.gamma * self.target_cnet(s_, self.target_anet(s_))
        q_eval = self.eval_cnet(s, a)

        # update critic net
        self.optimizer_c.zero_grad()
        c_loss = self.Loss(q_eval, q_target).mean()
        c_loss.backward()
        nn.utils.clip_grad_norm_(self.eval_cnet.parameters(), self.max_grad_norm)
        self.optimizer_c.step()

        # update actor net
        self.optimizer_a.zero_grad()
        a_loss = -self.eval_cnet(s, self.eval_anet(s)).mean()
        a_loss.backward()
        nn.utils.clip_grad_norm_(self.eval_anet.parameters(), self.max_grad_norm)
        self.optimizer_a.step()

        if self.training_step % 300 == 0:
            self.target_cnet.load_state_dict(self.eval_cnet.state_dict())
        if self.training_step % 301 == 0:
            self.target_anet.load_state_dict(self.eval_anet.state_dict())

        self.var = max(self.var * 0.999, 0.01)

        return q_eval.mean().item()
```

最後是整個 model 的更新，我們會從 buffer 裡 sample batch_size 大小的份數，然後做完各種前處理之後就是實做上面解釋過的，對 actor 和 critic 的更新了

而這邊為了方便，沒有讓 target network 做 soft update，而是跟 DQN 一樣到一定 episode 就覆蓋過去，結果也顯示還是不錯的

右圖則是存 history 的 class

更新方式是用一個 data_pointer 紀錄這串 list 的頭是哪裡，滿了後要更新就把他蓋過去

sample 也就直接用 numpy 的 random choice 來取出一定數量就完成了

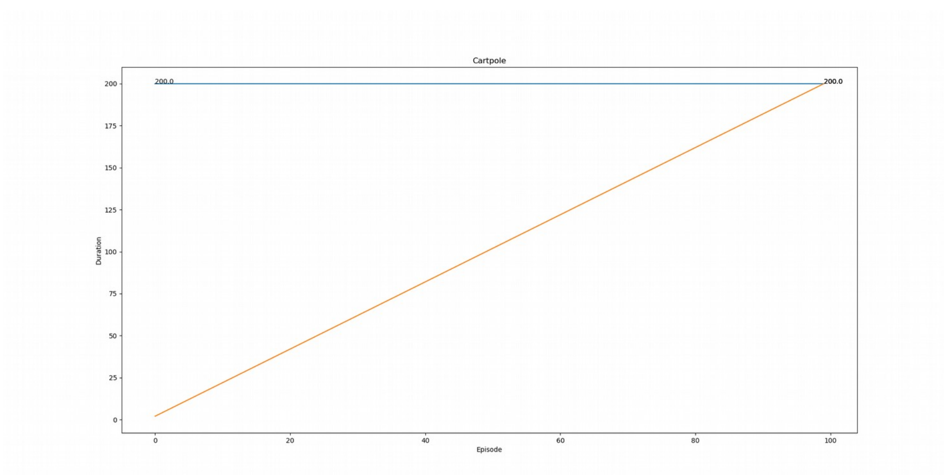
```python
class Memory():
    data_pointer = 0
    isfull = False

    def __init__(self, capacity):
        self.memory = np.empty(capacity, dtype=object)
        self.capacity = capacity

    def update(self, transition):
        self.memory[self.data_pointer] = transition
        self.data_pointer += 1
        if self.data_pointer == self.capacity:
            self.data_pointer = 0
            self.isfull = True

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

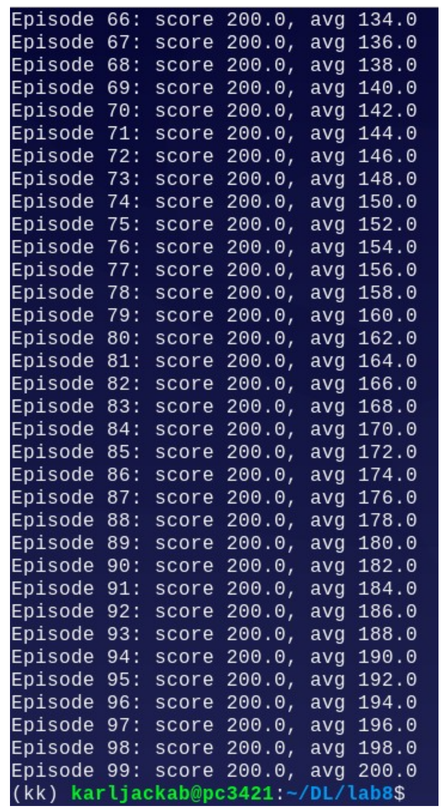
九、Other study or improvement for the project

其實老師在上這邊時非常的快速，我是自己額外看完 youtube 上李宏毅影片後才寫這份作業的
但其實自己寫的部份也沒有說非常多，在網路上有找到別人寫好的版本，參考後修改一些地方就下去跑了
其中也發現別人包成 class 的寫法和對於 sample history data 的寫法，甚至是 model 參數的設定方式都不一樣，收穫十分良多
pendulum 會要跑好幾個小時，但幸好沒太多問題，很快就 train 起來了
cartpole 則是 train 到很好的時候再多 train 就會不小心壞掉，應該是跟那個 epsilon 有關係，所以我在 training 階段，只要連續到 200 一定次數就不再 store weight，結果就可以像下面一樣都到 200 了

十、Performance

- CartPole: 200





```
Episode 66: score 200.0, avg 134.0
Episode 67: score 200.0, avg 136.0
Episode 68: score 200.0, avg 138.0
Episode 69: score 200.0, avg 140.0
Episode 70: score 200.0, avg 142.0
Episode 71: score 200.0, avg 144.0
Episode 72: score 200.0, avg 146.0
Episode 73: score 200.0, avg 148.0
Episode 74: score 200.0, avg 150.0
Episode 75: score 200.0, avg 152.0
Episode 76: score 200.0, avg 154.0
Episode 77: score 200.0, avg 156.0
Episode 78: score 200.0, avg 158.0
Episode 79: score 200.0, avg 160.0
Episode 80: score 200.0, avg 162.0
Episode 81: score 200.0, avg 164.0
Episode 82: score 200.0, avg 166.0
Episode 83: score 200.0, avg 168.0
Episode 84: score 200.0, avg 170.0
Episode 85: score 200.0, avg 172.0
Episode 86: score 200.0, avg 174.0
Episode 87: score 200.0, avg 176.0
Episode 88: score 200.0, avg 178.0
Episode 89: score 200.0, avg 180.0
Episode 90: score 200.0, avg 182.0
Episode 91: score 200.0, avg 184.0
Episode 92: score 200.0, avg 186.0
Episode 93: score 200.0, avg 188.0
Episode 94: score 200.0, avg 190.0
Episode 95: score 200.0, avg 192.0
Episode 96: score 200.0, avg 194.0
Episode 97: score 200.0, avg 196.0
Episode 98: score 200.0, avg 198.0
Episode 99: score 200.0, avg 200.0
(kk) karljackab@pc3421:~/DL/lab8$
```

- Pendulum：-152.091138