第一部分

采用 Q-learning 算法实现与作业 2 相同的任务。

源代码

```
1.      def qLearning(self,s0,initialQ,nEpisodes,nSteps,epsilon=0,temperature=0):
2.              '''
3.              qLearning 算法，需要将 Epsilon exploration 和 Boltzmann exploration 相结合。
4.              以 epsilon 的概率随机取一个动作，否则采用 Boltzmann exploration 取动作。
5.              当 epsilon 和 temperature 都为 0 时，将不进行探索。
6.
7.              Inputs:
8.              s0 -- 初始状态
9.              initialQ -- 初始化 Q 函数 (|A|x|S| array)
10.             nEpisodes -- 回合（episodes）的数
    量 (one episode consists of a trajectory of nSteps that starts in s0
11.             nSteps -- 每个回合的步数(steps)
12.             epsilon -- 随机选取一个动作的概率
13.             temperature -- 调节 Boltzmann exploration 的参数
```

```
14.
15.         Outputs:
16.         Q -- 最终的 Q 函数 (|A|x|S| array)
17.         policy -- 最终的策略
18.         rewardList -- 每个 episode 的累计奖励（|nEpisodes| array）
19.         '''
20.         nStates=self.mdp.nStates
21.         nActions=self.mdp.nActions
22.         Q=initialQ
23.         rewardList=np.zeros(nEpisodes)
24.         n=np.zeros([nActions,nStates])
25.         for episode in range(nEpisodes):
26.             total_reword=0
27.             s=s0 # 回合初始的地方
28.             for step in range(nSteps):
29.                 prob=np.random.uniform()
30.                 if(prob<epsilon): # 以一定的概率随机探索动作 exploration
31.                     action=np.random.choice(nActions)
32.                 else: # exploitation
33.                     if(temperature!=0): # Boltzmann
34.                         action=np.argmax(np.exp(Q[:,s]/temperature))
35.                     else: # greedy
36.                         action=np.argmax(Q[:,s])
37.                 reward,next_state=self.sampleRewardAndNextState(s,action) # 根据当前的状态和所选择的动作所采样得到的奖励和下一状态
38.                 n[action][s]+=1
39.                 alpha=1/n[action][s] # learning rate
40.                 Q[action][s]=Q[action][s]+alpha*(reward+self.mdp.discount*np.max(Q[:,next_state])-Q[action][s])
41.                 s=next_state
42.                 total_reword+=reward
43.             rewardList[episode]=total_reword
44.         policy=np.argmax(Q,axis=0)
45.         return [Q,policy,rewardList]
```

代码实现思路

Q-learning 是一个基于值的强化学习算法，根据 Q 函数评估应该选取哪个动作，这个函数决定了处于某一个特定状态（s）以及在该状态下采取特定动作（a）的奖励期望值，所以我们的目的就是最大化 Q 函数。Q-table 用于帮助我们找到每个状态中的最佳动作，通过选择当前状态下所有动作中的最佳动作来使期望奖励达到最大，最后根据 Q-table 中每一个状态下 Q 值最大策略的提取出 policy。在我们探索环境之前，Q-table 初始化为 0，但是随着对环境的持续探索，Q 给出了越来越好的近似，此时，我们就认为提取到了一个较好的 policy。

在 Q-learning 算法中，会经历若干次回合，在每一个回合中，每一步都会随机生成一个概率，若概率小于我们设定的 epsilon（探索速率），则采用随机探索的方式去选择一个动作，反之，我们将会进行 exploitation，选择当前 Q-table 中在当前状态下使得 Q 达到最大的动作。然后根据当前状态和所选取的动作进行立即回报和下一状态的采样，最后利用 Bellman 方程去迭代地更新 Q-learning 中的 Q 函数：$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$，其中$\alpha$为学习率（注意，在本代码中学习率会随着回合和步数的增加逐渐减小），直到所有回合结束后，提取 Q-table 中的 policy。其中，会记录每一个回合中的总回报。

TestRL.py 实现截图

```
Q-learning results
[[17.03924644 20.15767277 27.07746035 30.29101872]
 [13.86413982 24.08332942 29.55374975 39.69950003]]
[0 1 1 1]
```

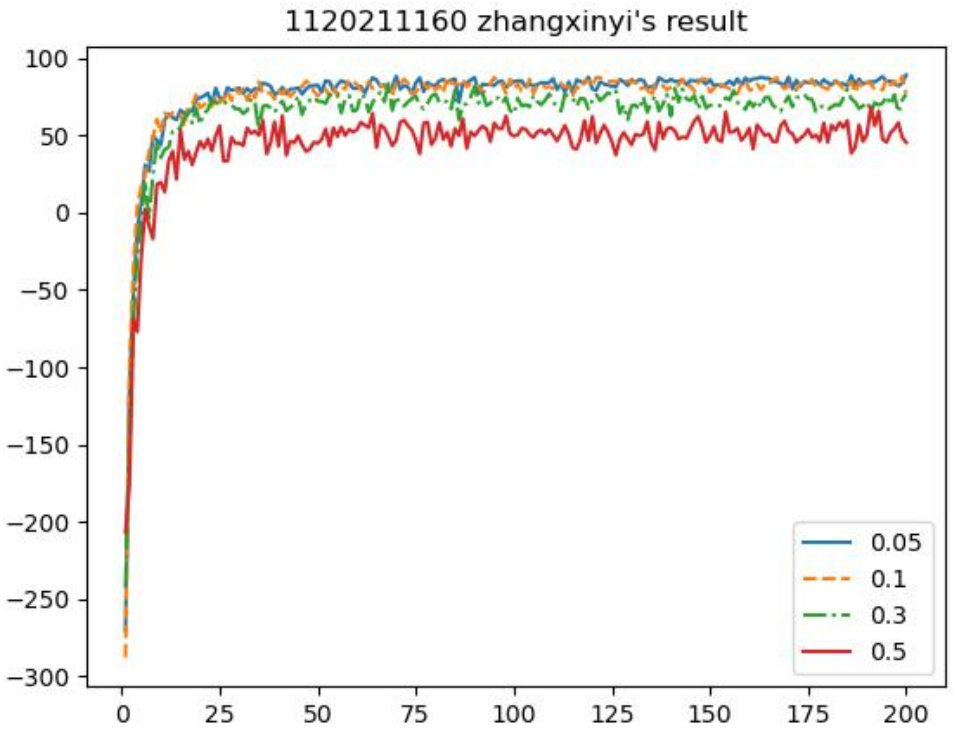
TestMaze.py 实现截图

```
epsilon=0.05
  rewardList_avg [-270.05758885 -114.34114536  -52.54338539  -11.55593416    6.63479275
   30.81330668   26.9713347    49.62748103   46.94084048   43.89657927
   60.11157588   63.99608027   61.54981815   60.14274798   66.46089136
   63.8433145    66.58050893   65.59105334   73.10127601   74.33278011
   75.57112467   77.37150449   69.94176511   80.60739991   74.18135628
   73.42889541   80.93895414   79.10141559   79.78845887   80.5744798
   77.17732105   80.322455     78.05027463   80.37335343   79.71900446
   83.83787699   83.18308427   81.83070157   76.1950797    77.93765816
   81.08870839   76.77781968   79.94789802   79.99432358   80.79994951
   76.70547957   79.95952784   82.04273225   82.15994389   82.58228902
   77.99163077   85.37522484   84.80626746   78.93357521   82.6307269
   85.37135417   84.22754728   83.67969335   80.52416552   80.4241021
   81.72712317   77.3396834    82.91283306   87.34110092   85.74455337
   81.76355004   82.61676913   79.09938629   81.49887175   88.31845595
   79.80357978   82.23661275   81.35451687   76.91483819   84.15051368
```

```
epsilon=0.1
  rewardList_avg [-287.6555379  -100.8436519   -35.50218638    4.40488578   16.79050074
   24.70410013   39.57345589   49.07815259   60.37714511   51.65277195
   64.56495955   62.08165345   64.6401362    64.4885636    64.03248779
   55.77774953   68.74766326   69.52820756   76.24545099   67.91234303
   69.73014963   72.25550392   68.72893499   75.49707758   71.24357873
   73.39037114   78.66899541   72.47706186   79.5134334    77.24781791
   80.39162622   72.73010239   78.50707805   74.2277321    84.72351509
```

```
epsilon=0.3
  rewardList_avg [-242.61624603 -151.14817468  -72.50543018  -27.91949581   -6.9256843
   20.1919465    -3.47622954   24.82058287   46.32011751   35.64827137
   40.98170351   42.38059262   52.67223422   52.53450121   55.37644898
   60.56327249   64.90909569   57.96846252   67.55973956   57.04483382
   64.21602958   66.75074538   72.89406598   71.07344281   70.79605684
   72.32902292   69.87764444   69.71468636   67.28993639   67.1737974
   64.110794     66.10289525   73.88412573   71.12582137   53.29652732
```

```
epsilon=0.5
  rewardList_avg [-206.67560328 -176.04142702  -68.31009826  -77.13032895  -29.4347833
    2.25792274   -8.52639338  -16.90537156   18.28503585   19.63131859
   13.39914472   33.25720242   39.79344209   21.70145184   53.72371233
   34.4453823    39.74285939   31.01375078   38.92664911   46.18966267
   41.90166148   47.65828421   39.91586834   49.73883397   56.29312853
   33.45942045   33.40386967   48.79220286   44.87856253   45.09726515
   43.33742702   54.7352017    51.3654263    52.7925263    50.04647424
   59.7264801    38.03078545   45.51245374   58.87640008   43.05930627
   62.72569539   37.13296049   46.32044683   45.15936944   49.67782374
```



分析生成的 Q-learning.jpg 的含义和曲线的走势

Q-learning.jpg 中横轴表示回合（Episode）次数，纵轴表示对总 reward 进行整体缩小后的数值，其中不同颜色的线分别表示在不同 epsilon（探索速率）下随着回合增加总 reward 的变化。

根据 Q-learning.jpg 可知，随着 epsilon 的减小，算法收敛速度缓缓加快，红线（epsilon=0.5）收敛明显慢于其他另外三条线。同时，随着 epsilon 的减小总 reward 明显提升，蓝线（epsilon=0.05）和黄线（epsilon=0.1）比较接近，表明总 reward 将不会随着 epsilon 的继续减小而提升，即 epsilon 在 0.05 到 0.1 之间为较优的探索速率。

Epsilon 越小，表明更加倾向于 exploitation，利用 Q-table 中的已有的 Q 值函数进行下一动作的选择。

第二部分

Q-learning 与 sarsa 的实现思路和算法分析

在 Q-learning 中 Q 函数的 bellman 迭代方程为 $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$；在 sarsa 中 Q 函数的 bellman 迭代方程为 $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$，以及还需要在获得新的状态信息后选择出新的动作，其他的与 Q-learning 一样。

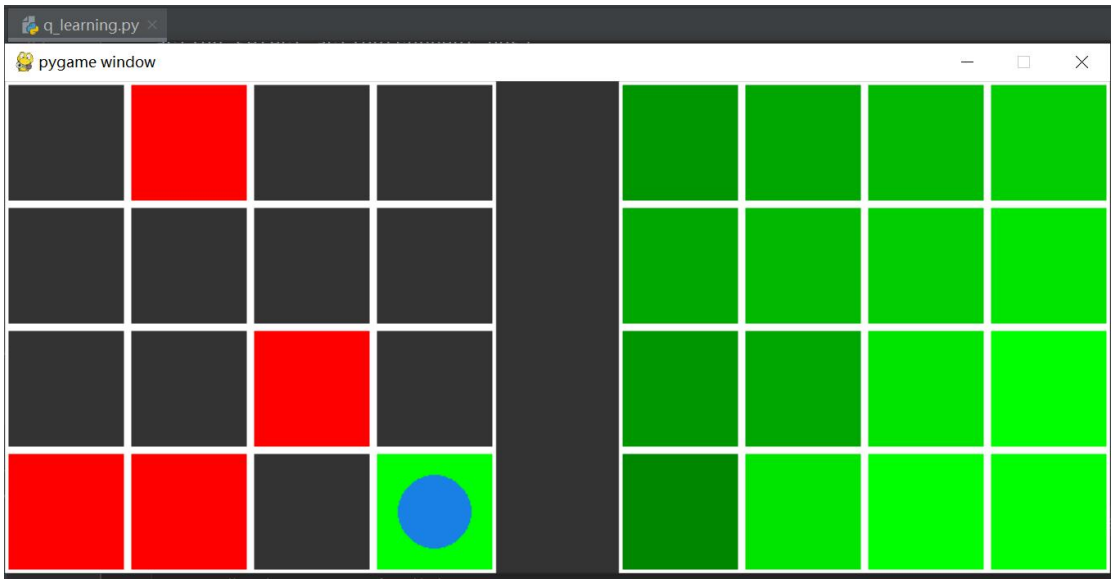
二者的区别在于更新 Q-table 的时候选择的策略不同，Q-learning 是选择下一状态中最优动作来进行 Q 值的更新，sarsa 的选择策略是 Q 值直接与上一个策略一样。

Q-learning 补全代码截图

```
    global current_pos, epsilon_delta, epsilon
    #此处需要进行动作选取,给action赋值
    action=select_action(current_pos)
```

```
    if new_pos not in terminals:
        #此处实现Q-learning算法
        Q[current_pos][action]+= alpha*(reward[new_pos]+gamma*np.max(Q[new_pos])-Q[current_pos][action])
    else:
        # 此处实现Q-learning算法
        Q[current_pos][action]+=alpha*(reward[new_pos]-Q[current_pos][action])
```

Q-learning 运行截图

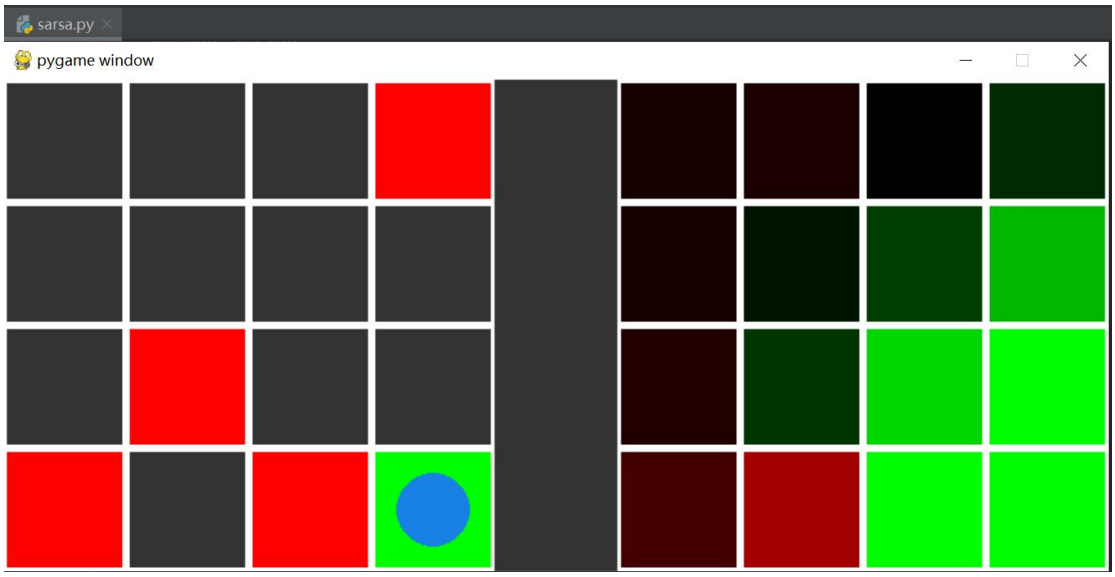


Sarsa 补全代码截图

```
#此处需要实现动作选取，注意此处action和q-learning的区别
action=select_action(current_pos)

new_pos = list(current_pos)
if action==0 and new_pos[0]>0: #move up
    new_pos[0] -= 1
elif action==1 and new_pos[0]<n-1: #move down
    new_pos[0] += 1
elif action==2 and new_pos[1]>0: #move left
    new_pos[1] -= 1
elif action==3 and new_pos[1]<n-1: #move right
    new_pos[1] += 1
new_pos = tuple(new_pos)
new_action=select_action(new_pos)
if new_pos not in terminals:
    #此处需要实现sarsa算法
    Q[current_pos][action]+=alpha*(reward[new_pos]+gamma*Q[new_pos][new_action]-Q[current_pos][action])
else:
    #此处需要实现sarsa算法
    Q[current_pos][action]+=alpha*(reward[new_pos]-Q[current_pos][action])
```

Sarsa 运行截图



第三部分

本次实验选择运行 pytorch 版本的 flappy bird 代码。

算法分析

在原始的 DQN 算法中，使用一个神经网络来逼近 Q 函数，该网络接受 state 作为输入，并生成每个动作的 Q-value 作为输出。然而，在每次更新 Q 网络参数时，它的目标值是由该网络自身生成的，这导致了在每次参数更新时，Q 网络的目标值也会随之改变。这会导致**训练过程中目标值的剧烈波动**，使得网络训练变得不稳定，造成了 **DQN 的高估**；同时由于目标值是由同一个网络生成的，这意味着目标值与实际值之间存在高度相关性，这会导致网络在训练过程中**很难收敛到准确的 Q 值估计**。Target Network 是一个与 Q 网络结构相同的目标网络，它的参数在训练过程中保持不变，不受梯度更新的影响。**通过引入 Target Network，DQN 算法可以更加稳定地进行训练，并且能够更好地收敛到最优的 Q 值函数估计，从而提高算法的性能和效果。**

实现思路

根据以上的算法分析可知，，Target Network 的实现思路是定期从主网络复制参数，使其保持与主网络一致，并用于计算目标 Q 值，从而提高训练的稳定性和收敛性。所以在代码中设置了一个 iteration_target，当网络的训练次数达到其整数倍时，将主网络的参数复制到 Target Network 中。（如 train 函数中 17 行所示）

具体代码

添加 Target Network 的 DQN 代码如下所示（仅修改 train 函数，并增添了“pre”训练部分。其中“pre”可以用于继续训练以前保存的模型，使得模型可以在训练途中暂停下来。本部分代码只展示 dqn.py 的修改部分，完整代码可查看源代码文件）：

Main 函数中的 train 和 pre 功能的实现

```
    elif mode == 'train':
            if not os.path.exists('pretrained_model/'):
                os.mkdir('pretrained_model/')

            model = NeuralNetwork()
            model_target = NeuralNetwork()

            if cuda_is_available:  # put on GPU if CUDA is available
                model,model_target = model.cuda(), model_target.cuda()

            model.apply(init_weights)
            start = time.time()

            train(model, model_target, start)

        elif mode == 'pre':
            iteration_start = 10
            print("train from iter:", iteration_start)
            model = torch.load("./pretrained_model/current_model_"+str(iteration_start)+".pth")
            model_target = NeuralNetwork()
            if cuda_is_available:  # put on GPU if CUDA is available
                model, model_target = model.cuda(), model_target.cuda()
            start = time.time()
            train(model,model_target,start,iteration_start,iteration_target=100)
```

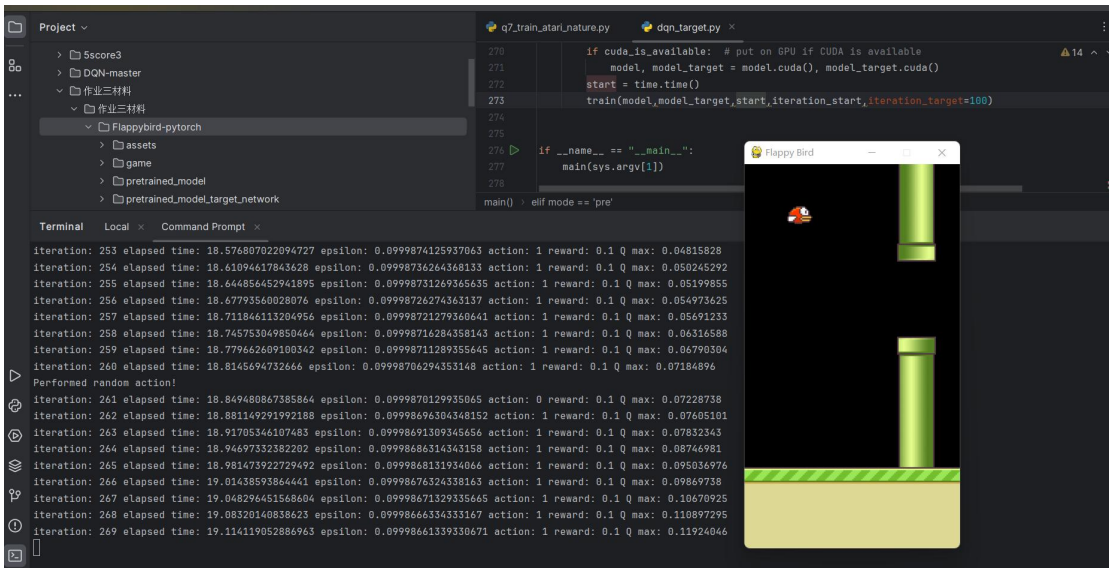
train 函数部分，其中修改部分为 13、16、17、18、53 行，已用红色标识出来

```
    def train(model, model_target, start, iteration_start=0,iteration_target = 100):
        optimizer = optim.Adam(model.parameters(), lr=1e-6)
        criterion = nn.MSELoss()
        game_state = GameState()
        replay_memory = []
        action = torch.zeros([model.number_of_actions], dtype=torch.float32)
        action[0] = 1
        image_data, reward, terminal = game_state.frame_step(action)
        image_data = resize_and_bgr2gray(image_data)
        image_data = image_to_tensor(image_data)
        state = torch.cat((image_data, image_data, image_data, image_data)).unsqueeze(0)
```

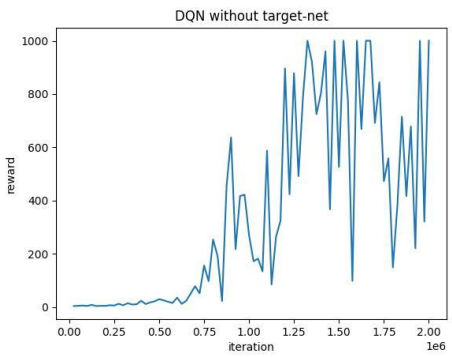
```
        epsilon = model.initial_epsilon
        iteration = iteration_start
        epsilon_decrements = np.linspace(model.initial_epsilon, model.final_epsilon, model.number_of_iterations)
        while iteration < model.number_of_iterations:
            if iteration % iteration_target == 0:
                model_target.load_state_dict(model.state_dict())
                print("replace target_net's params from origin_net")
            output = model(state)[0]
            action = torch.zeros([model.number_of_actions], dtype=torch.float32)
            if torch.cuda.is_available():  # put on GPU if CUDA is available
                action = action.cuda()
            random_action = random.random() <= epsilon
            if random_action:
                print("Performed random action!")
            action_index = [torch.randint(model.number_of_actions, torch.Size([]), dtype=torch.int)
                            if random_action
                            else torch.argmax(output)][0]
            if torch.cuda.is_available():  # put on GPU if CUDA is available
                action_index = action_index.cuda()
            action[action_index] = 1
            image_data_1, reward, terminal = game_state.frame_step(action)
            image_data_1 = resize_and_bgr2gray(image_data_1)
            image_data_1 = image_to_tensor(image_data_1)
            state_1 = torch.cat((state.squeeze(0)[1:, :, :], image_data_1)).unsqueeze(0)
            action = action.unsqueeze(0)
            reward = torch.from_numpy(np.array([reward], dtype=np.float32)).unsqueeze(0)
            replay_memory.append((state, action, reward, state_1, terminal))
            if len(replay_memory) > model.replay_memory_size:
                replay_memory.pop(0)
            epsilon = epsilon_decrements[iteration]
            minibatch = random.sample(replay_memory, min(len(replay_memory), model.minibatch_size))
            # unpack minibatch
            state_batch = torch.cat(tuple(d[0] for d in minibatch))
            action_batch = torch.cat(tuple(d[1] for d in minibatch))
            reward_batch = torch.cat(tuple(d[2] for d in minibatch))
            state_1_batch = torch.cat(tuple(d[3] for d in minibatch))
            if torch.cuda.is_available():  # put on GPU if CUDA is available
                state_batch = state_batch.cuda()
                action_batch = action_batch.cuda()
                reward_batch = reward_batch.cuda()
                state_1_batch = state_1_batch.cuda()
            output_1_batch = model_target(state_1_batch)
            y_batch = torch.cat(tuple(reward_batch[i] if minibatch[i][4]
```

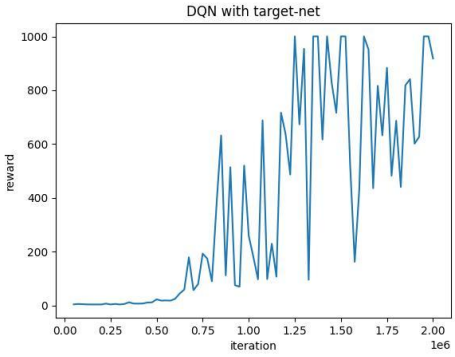
```
                                    else reward_batch[i] + model.gamma * torch.max(output_1_batch[i]
    )
                                    for i in range(len(minibatch))))
            q_value = torch.sum(model(state_batch) * action_batch, dim=1)
            optimizer.zero_grad()
            y_batch = y_batch.detach()
            loss = criterion(q_value, y_batch)
            loss.backward()
            optimizer.step()
            state = state_1
            iteration += 1
            if iteration % 25000 == 0:
                torch.save(model, "pretrained_model/current_model_" + str(iteration) + ".pth")
            print("iteration:", iteration, "elapsed time:", time.time() - start, "epsilon:", epsilon,
    "action:",
                  action_index.cpu().detach().numpy(), "reward:", reward.numpy()[0][0], "Q max:",
                  np.max(output.cpu().detach().numpy()))
```

代码运行截图



训练过程中奖励的变化图（DQN without target-net and DQN with target-net）：





可以看出，在加入 targetnetwork 后，迭代至 80w 次时比没有 Target Network 的稳定，迭代至 175w 次时比没有 Target Network 的稳定，但是两者都没有在迭代至 200w 次时收敛，猜测是训练轮次过少。

第四部分

本部分从 DQN 原始论文中获取 DQN 网络的相关参数，

第一个隐层，卷积 32 个 8*8 的滤波器，stride=4，使用 ReLU 作为激活函数。第二个隐层卷积 64 个 4*4 的滤波器，stride=2，使用 ReLU 作为激活函数。第三个隐层，卷积 64 个 3*3 的滤波器，stride=1，使用 ReLU 作为激活函数。最后一个隐层是全连接的，该层由 512 个 ReLU 组成。输出层是一个全连接的线性层，只有一个输出。我们在游戏中有效动作只考虑 4~18 之间的个数。

代码截图及实现思路

根据以上提到的参数信息，在 initialize_models 函数中建立了以下 Q-Network 和 TargetNetwork：

```
# 创建Q网络
self.q_network = nn.Sequential(
    nn.Conv2d(in_channels=n_channels * self.config.state_history, out_channels=32, kernel_size=8, stride=4,
              padding=((4 - 1) * img_height - 4 + 8) // 2),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2,
              padding=((2 - 1) * img_height - 2 + 4) // 2),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1,
              padding=((1 - 1) * img_height - 1 + 3) // 2),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * img_height * img_width,  out_features: 512),
    nn.ReLU(),
    nn.Linear(in_features=512, out_features=num_actions)
)
```

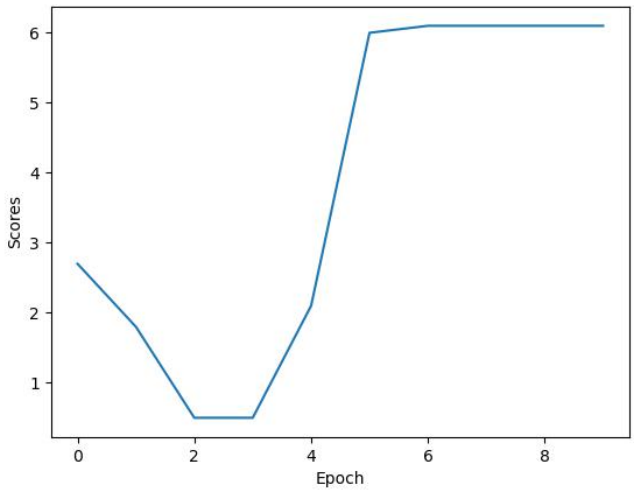
```
# 创建Target网络，与Q网络具有相同的配置但从头开始初始化
self.target_network = nn.Sequential(
    nn.Conv2d(in_channels=n_channels * self.config.state_history, out_channels=32, kernel_size=8, stride=4,
              padding=((4 - 1) * img_height - 4 + 8) // 2),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2,
              padding=((2 - 1) * img_height - 2 + 4) // 2),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1,
              padding=((1 - 1) * img_height - 1 + 3) // 2),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(64 * img_height * img_width, out_features: 512),
    nn.ReLU(),
    nn.Linear(in_features=512, out_features=num_actions)
)
# 模型的输入大小为img_height * img_width的图像，通道数为n_channels * self.config.state_history
input_size = (img_height, img_width, n_channels * self.config.state_history)

return input_size
############################################################
######################## END YOUR CODE ######################
```

根据代码中提到的 Q 值函数$Q_{sample}(s) = r + \gamma * max_{a'}Q_{target}(s', a')$和 loss 函数（$loss = (Q_{sample}(s) - Q(s,a))^2$）的计算公式，建立了以下 calc_loss 函数：

```
################################################################
###################### YOUR CODE HERE - 3-5 lines #############
q_samp = rewards + gamma * torch.max(target_q_values, dim=1)[0] * (~done_mask)
actions = actions.to(torch.int64)
q_values = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)
loss = torch.nn.functional.mse_loss(q_samp, q_values)
return loss
################################################################
######################### END YOUR CODE ######################
```

Q6 运行结果图



在 q7 中，本次实验调用了三个不同的模型参数 5score.part1.rar/ 5score.part2.rar/ 5score.part3.rar，每一个模型下分别得到三个视频，比分分别为（冒号右侧为我们训练的 agent）：

20:19；17:20；14:20

6:20；18:20,；19:20

20:20；8:20；9:20

根据比值和得分，认为 6:20 是最好的一次对战。