[第一部分 1](#_Toc22713)

[源代码 1](#_Toc7172)

[代码实现思路 2](#_Toc26876)

[TestRL.py实现截图 3](#_Toc9592)

[TestMaze.py实现截图 3](#_Toc3591)

[分析生成的Q-learning.jpg的含义和曲线的走势 5](#_Toc8641)

[第二部分 5](#_Toc18448)

[Q-learning与sarsa的实现思路和算法分析 5](#_Toc5024)

[Q-learning补全代码截图 5](#_Toc3242)

[Q-learning运行截图 6](#_Toc2249)

[Sarsa补全代码截图 6](#_Toc12136)

[Sarsa运行截图 7](#_Toc14475)

[第三部分 7](#_Toc31825)

[算法分析 7](#_Toc20701)

[实现思路 7](#_Toc31567)

[具体代码 8](#_Toc11432)

[代码运行截图 10](#_Toc19392)

[训练过程中奖励的变化图 10](#_Toc17794)

[第四部分 11](#_Toc16840)

[代码截图及实现思路 11](#_Toc1090)

[Q6运行结果图 12](#_Toc31523)

# 第一部分

采用Q-learning算法实现与作业2相同的任务。

## 源代码

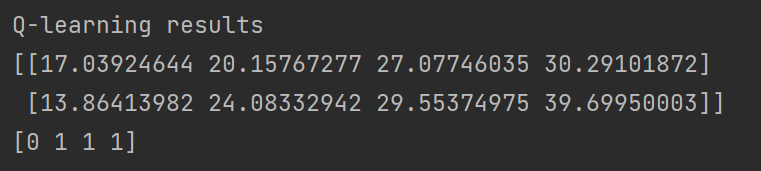
1. def qLearning(self,s0,initialQ,nEpisodes,nSteps,epsilon=0,temperature=0):
2. '''
3. qLearning算法，需要将Epsilon exploration和 Boltzmann exploration 相结合。
4. 以epsilon的概率随机取一个动作，否则采用 Boltzmann exploration取动作。
5. 当epsilon和temperature都为0时，将不进行探索。
6. Inputs:
7. s0 -- 初始状态
8. initialQ -- 初始化Q函数 (|A|x|S| array)
9. nEpisodes -- 回合（episodes）的数量 (one episode consists of a trajectory of nSteps that starts in s0
10. nSteps -- 每个回合的步数(steps)
11. epsilon -- 随机选取一个动作的概率
12. temperature -- 调节 Boltzmann exploration 的参数
13. Outputs:
14. Q -- 最终的 Q函数 (|A|x|S| array)
15. policy -- 最终的策略
16. rewardList -- 每个episode的累计奖励（|nEpisodes| array）
17. '''
18. nStates=self.mdp.nStates
19. nActions=self.mdp.nActions
20. Q=initialQ
21. rewardList=np.zeros(nEpisodes)
22. n=np.zeros([nActions,nStates])
23. for episode in range(nEpisodes):
24. total\_reword=0
25. s=s0 *# 回合初始的地方*
26. for step in range(nSteps):
27. prob=np.random.uniform()
28. if(prob<epsilon): *# 以一定的概率随机探索动作 exploration*
29. action=np.random.choice(nActions)
30. else: *# exploitation*
31. if(temperature!=0): *# Boltzmann*
32. action=np.argmax(np.exp(Q[:,s]/temperature))
33. else: *# greedy*
34. action=np.argmax(Q[:,s])
35. 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*
36. n[action][s]+=1
37. alpha=1/n[action][s] *# learning rate*
38. Q[action][s]=Q[action][s]+alpha\*(reward+self.mdp.discount\*np.max(Q[:,next\_state])-Q[action][s])
39. s=next\_state
40. total\_reword+=reward
41. rewardList[episode]=total\_reword
42. policy=np.argmax(Q,axis=0)
43. return [Q,policy,rewardList]

## 代码实现思路

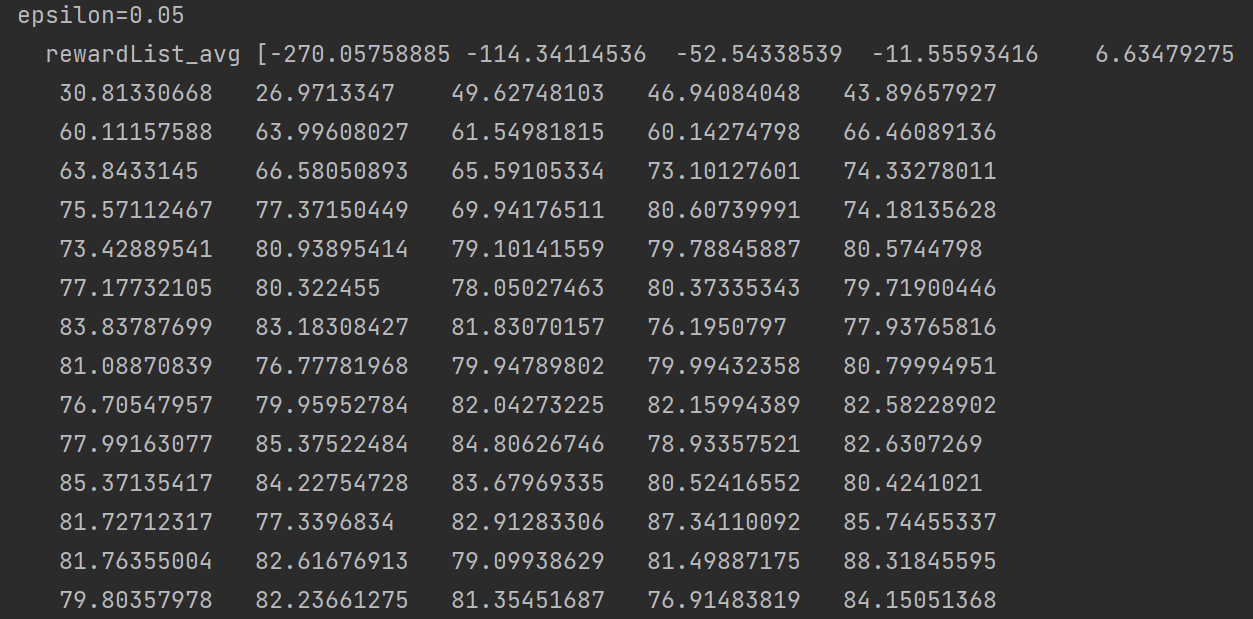
1. learning是一个基于值的强化学习算法，根据Q函数评估应该选取哪个动作，这个函数决定了处于某一个特定状态（s）以及在该状态下采取特定动作（a）的奖励期望值，所以我们的目的就是最大化Q函数。Q-table用于帮助我们找到每个状态中的最佳动作，通过选择当前状态下所有动作中的最佳动作来使期望奖励达到最大，最后根据Q-table中每一个状态下Q值最大策略的提取出policy。在我们探索环境之前，Q-table初始化为0，但是随着对环境的持续探索，Q给出了越来越好的近似，此时，我们就认为提取到了一个较好的policy。

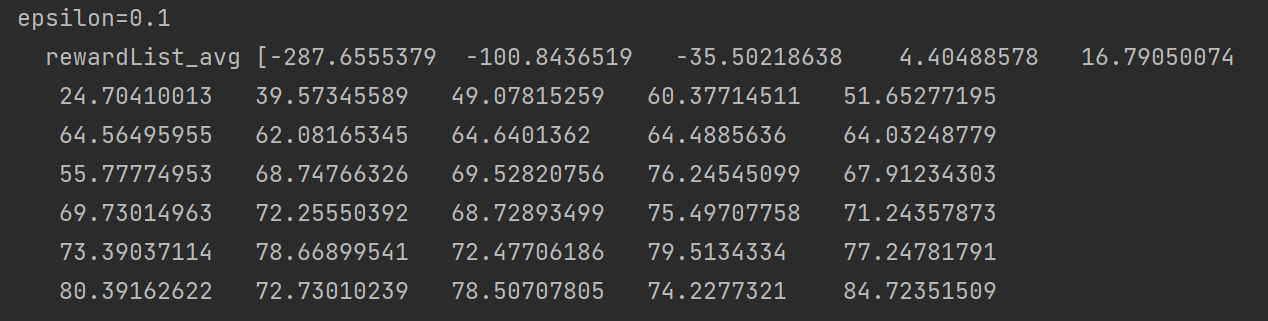
在Q-learning算法中，会经历若干次回合，在每一个回合中，每一步都会随机生成一个概率，若概率小于我们设定的epsilon（探索速率），则采用随机探索的方式去选择一个动作，反之，我们将会进行exploitation，选择当前Q-table中在当前状态下使得Q达到最大的动作。然后根据当前状态和所选取的动作进行立即回报和下一状态的采样，最后利用Bellman方程去迭代地更新Q-learning中的Q函数：，其中为学习率（注意，在本代码中学习率会随着回合和步数的增加逐渐减小），直到所有回合结束后，提取Q-table中的policy。其中，会记录每一个回合中的总回报。

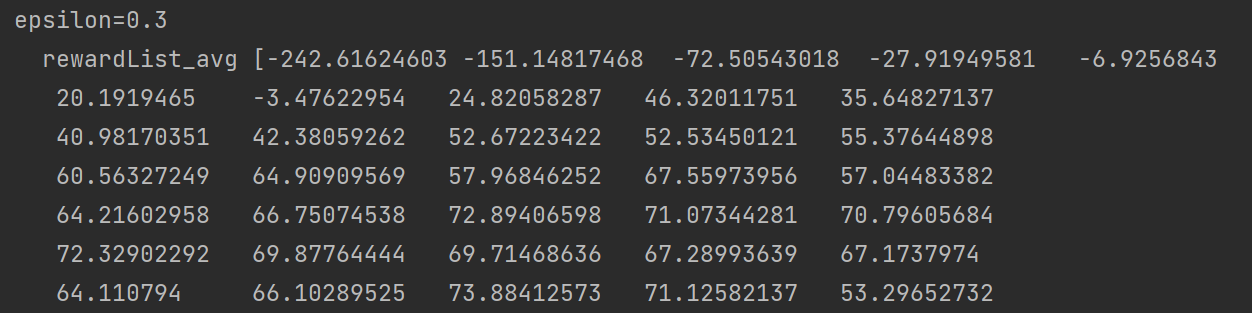
## TestRL.py实现截图

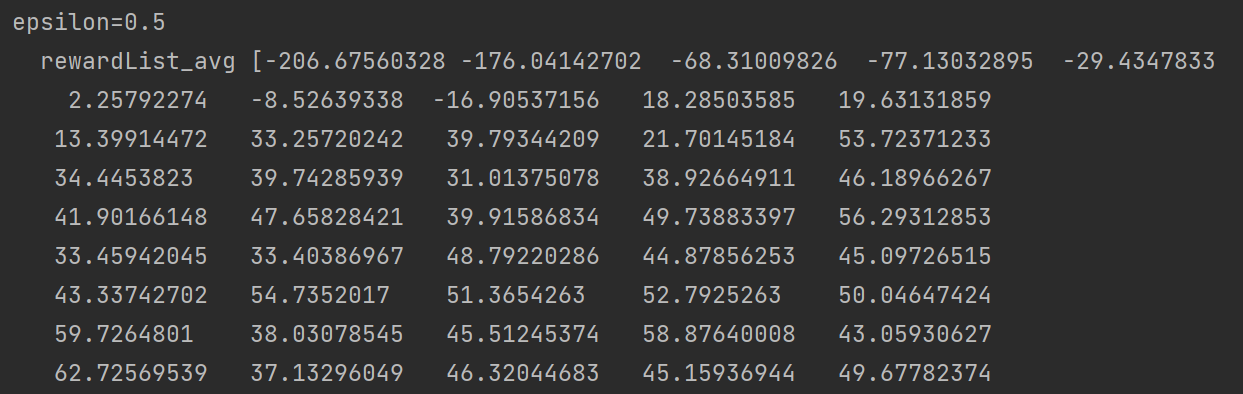


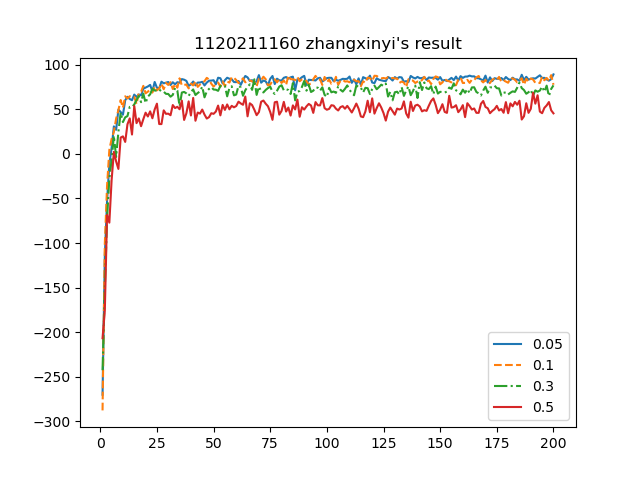
## TestMaze.py实现截图











## 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。

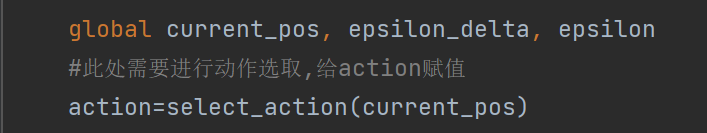
# 第二部分

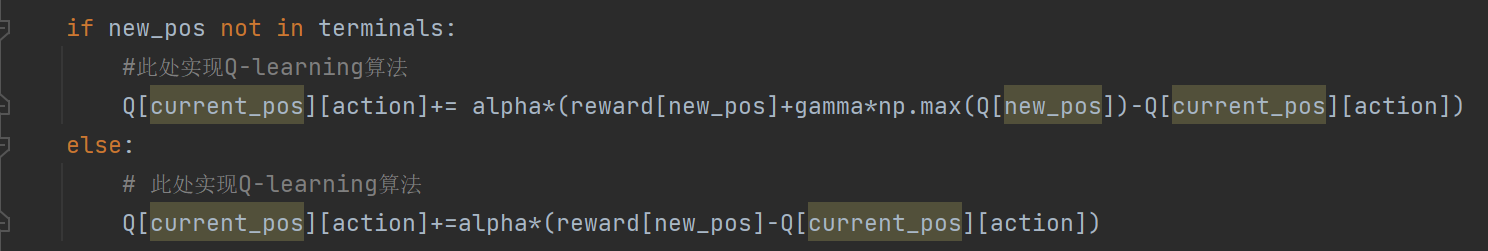
## Q-learning与sarsa的实现思路和算法分析

在Q-learning中Q函数的bellman迭代方程为；在sarsa中Q函数的bellman迭代方程为，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。

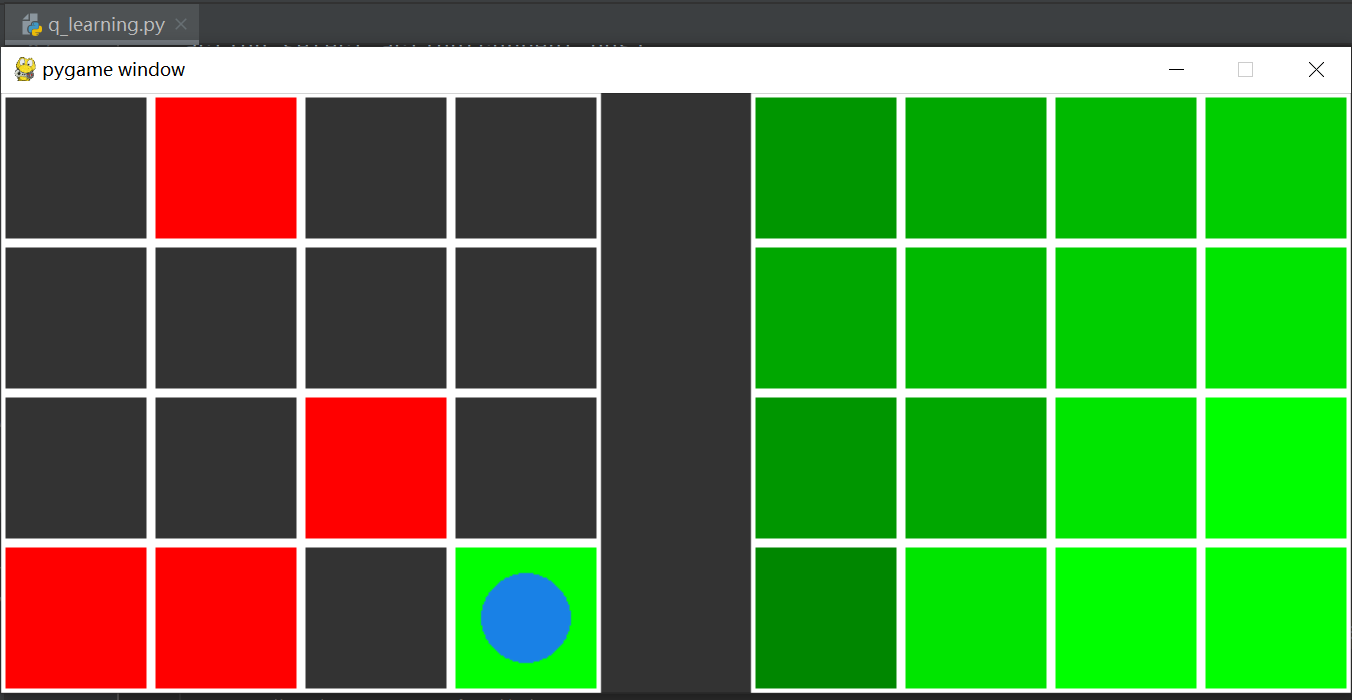
二者的区别在于更新Q-table的时候选择的策略不同，Q-learning是选择下一状态中最优动作来进行Q值的更新，sarsa的选择策略是Q值直接与上一个策略一样。

## Q-learning补全代码截图

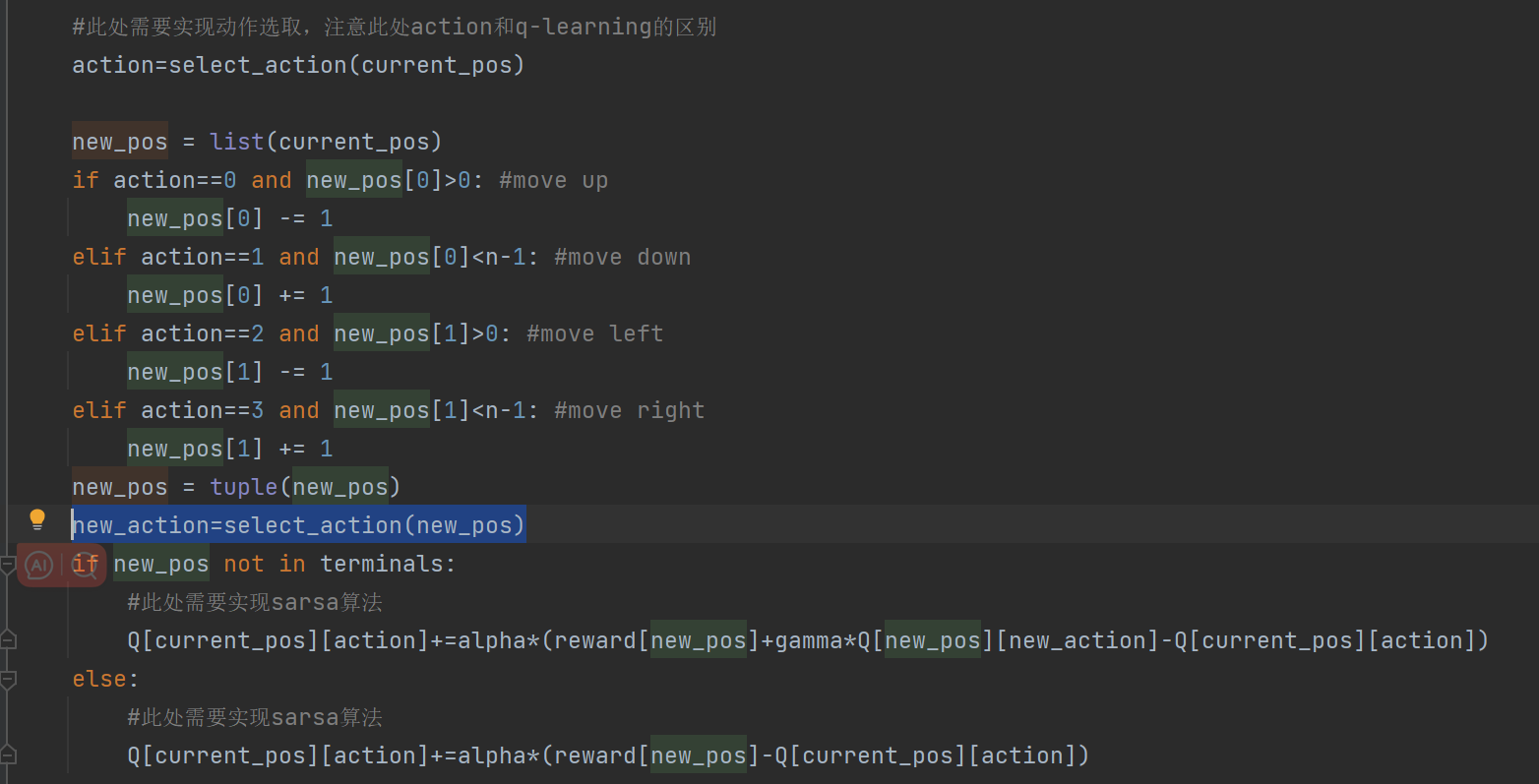




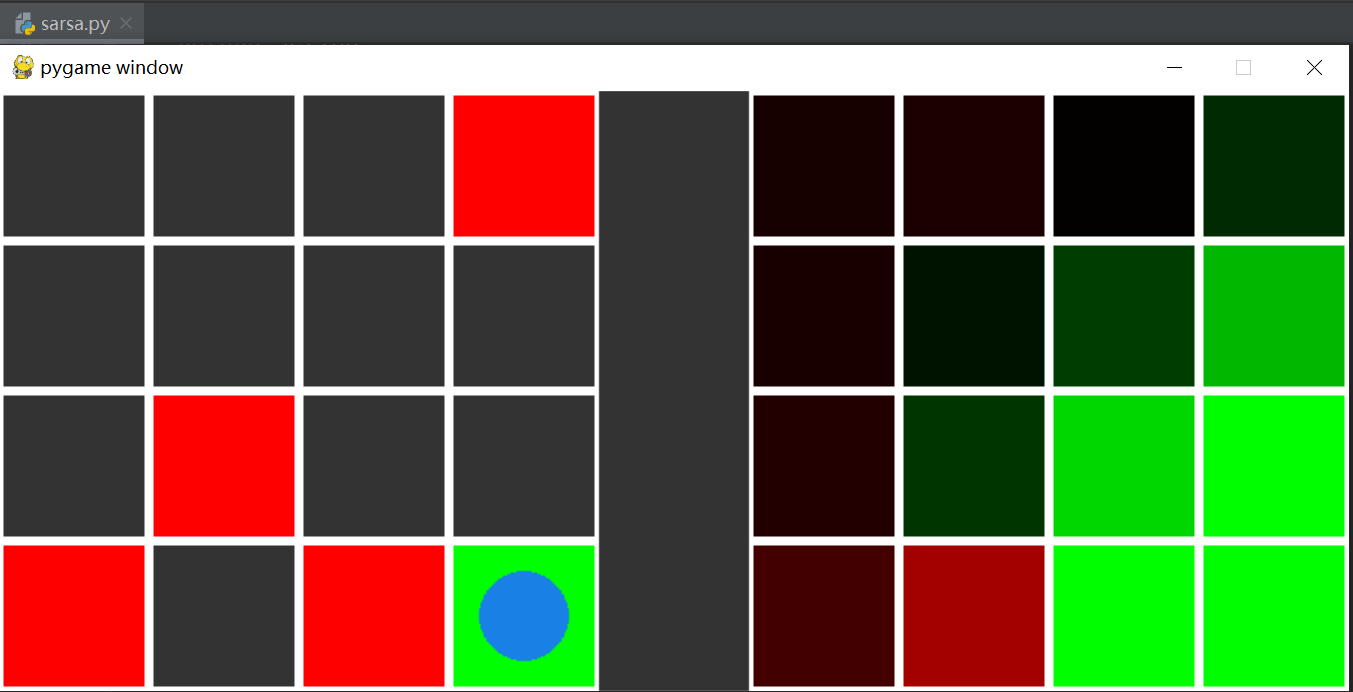
## Q-learning运行截图



## Sarsa补全代码截图



## 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）：

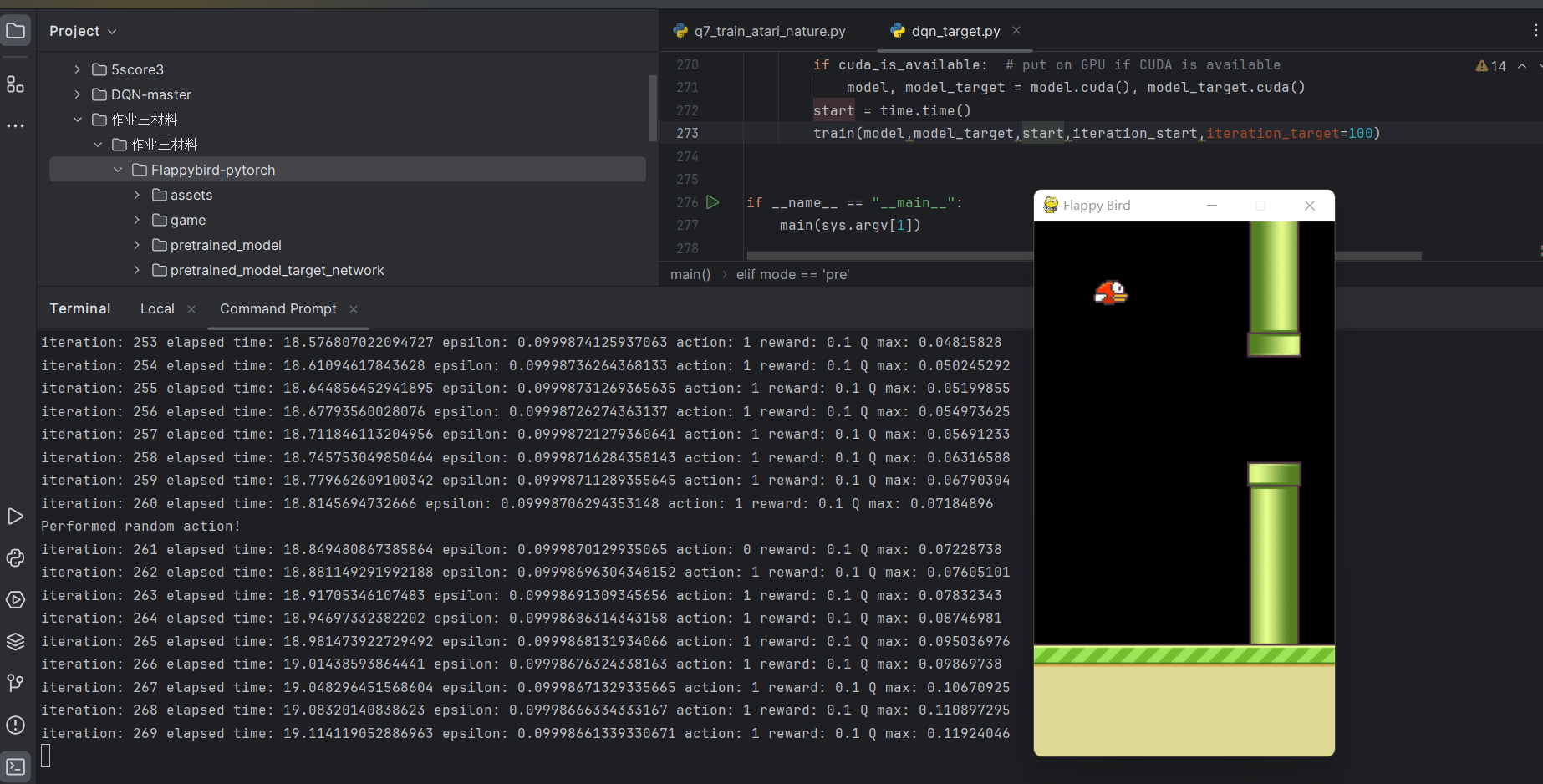
**Main函数中的train和pre功能的实现**

1. elif mode == 'train':
2. if not os.path.exists('pretrained\_model/'):
3. os.mkdir('pretrained\_model/')
4. model = NeuralNetwork()
5. model\_target = NeuralNetwork()
6. if cuda\_is\_available:  *# put on GPU if CUDA is available*
7. model,model\_target = model.cuda(), model\_target.cuda()
8. model.apply(init\_weights)
9. start = time.time()
10. train(model, model\_target, start)
12. elif mode == 'pre':
13. iteration\_start = 10
14. print("train from iter:", iteration\_start)
15. model = torch.load("./pretrained\_model/current\_model\_"+str(iteration\_start)+".pth")
16. model\_target = NeuralNetwork()
17. if cuda\_is\_available:  *# put on GPU if CUDA is available*
18. model, model\_target = model.cuda(), model\_target.cuda()
19. start = time.time()
20. train(model,model\_target,start,iteration\_start,iteration\_target=100)

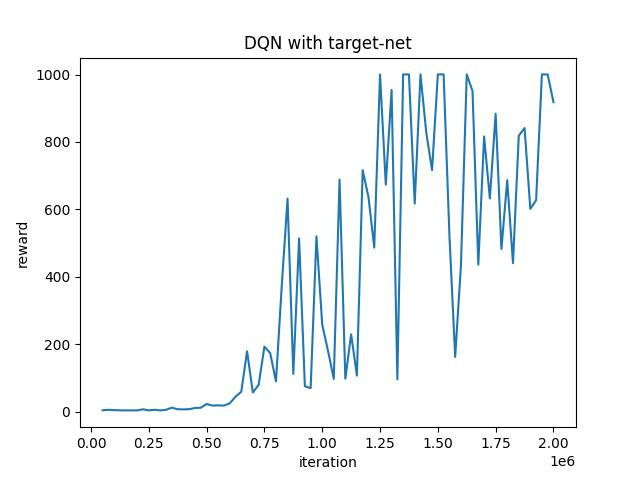
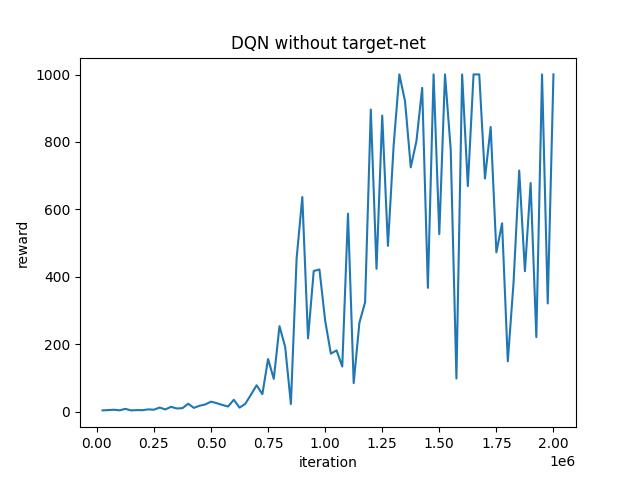
**train函数部分，其中修改部分为13、16、17、18、53行，已用红色标识出来**

1. def train(model, model\_target, start, iteration\_start=0,iteration\_target = 100):
2. optimizer = optim.Adam(model.parameters(), lr=1e-6)
3. criterion = nn.MSELoss()
4. game\_state = GameState()
5. replay\_memory = []
6. action = torch.zeros([model.number\_of\_actions], dtype=torch.float32)
7. action[0] = 1
8. image\_data, reward, terminal = game\_state.frame\_step(action)
9. image\_data = resize\_and\_bgr2gray(image\_data)
10. image\_data = image\_to\_tensor(image\_data)
11. state = torch.cat((image\_data, image\_data, image\_data, image\_data)).unsqueeze(0)
12. epsilon = model.initial\_epsilon
13. iteration = iteration\_start
14. epsilon\_decrements = np.linspace(model.initial\_epsilon, model.final\_epsilon, model.number\_of\_iterations)
15. while iteration < model.number\_of\_iterations:
16. if iteration % iteration\_target == 0:
17. model\_target.load\_state\_dict(model.state\_dict())
18. print("replace target\_net's params from origin\_net")
19. output = model(state)[0]
20. action = torch.zeros([model.number\_of\_actions], dtype=torch.float32)
21. if torch.cuda.is\_available():  *# put on GPU if CUDA is available*
22. action = action.cuda()
23. random\_action = random.random() <= epsilon
24. if random\_action:
25. print("Performed random action!")
26. action\_index = [torch.randint(model.number\_of\_actions, torch.Size([]), dtype=torch.int)
27. if random\_action
28. else torch.argmax(output)][0]
29. if torch.cuda.is\_available():  *# put on GPU if CUDA is available*
30. action\_index = action\_index.cuda()
31. action[action\_index] = 1
32. image\_data\_1, reward, terminal = game\_state.frame\_step(action)
33. image\_data\_1 = resize\_and\_bgr2gray(image\_data\_1)
34. image\_data\_1 = image\_to\_tensor(image\_data\_1)
35. state\_1 = torch.cat((state.squeeze(0)[1:, :, :], image\_data\_1)).unsqueeze(0)
36. action = action.unsqueeze(0)
37. reward = torch.from\_numpy(np.array([reward], dtype=np.float32)).unsqueeze(0)
38. replay\_memory.append((state, action, reward, state\_1, terminal))
39. if len(replay\_memory) > model.replay\_memory\_size:
40. replay\_memory.pop(0)
41. epsilon = epsilon\_decrements[iteration]
42. minibatch = random.sample(replay\_memory, min(len(replay\_memory), model.minibatch\_size))
43. *# unpack minibatch*
44. state\_batch = torch.cat(tuple(d[0] for d in minibatch))
45. action\_batch = torch.cat(tuple(d[1] for d in minibatch))
46. reward\_batch = torch.cat(tuple(d[2] for d in minibatch))
47. state\_1\_batch = torch.cat(tuple(d[3] for d in minibatch))
48. if torch.cuda.is\_available():  *# put on GPU if CUDA is available*
49. state\_batch = state\_batch.cuda()
50. action\_batch = action\_batch.cuda()
51. reward\_batch = reward\_batch.cuda()
52. state\_1\_batch = state\_1\_batch.cuda()
53. output\_1\_batch = model\_target(state\_1\_batch)
54. y\_batch = torch.cat(tuple(reward\_batch[i] if minibatch[i][4]
55. else reward\_batch[i] + model.gamma \* torch.max(output\_1\_batch[i])
56. for i in range(len(minibatch))))
57. q\_value = torch.sum(model(state\_batch) \* action\_batch, dim=1)
58. optimizer.zero\_grad()
59. y\_batch = y\_batch.detach()
60. loss = criterion(q\_value, y\_batch)
61. loss.backward()
62. optimizer.step()
63. state = state\_1
64. iteration += 1
65. if iteration % 25000 == 0:
66. torch.save(model, "pretrained\_model/current\_model\_" + str(iteration) + ".pth")
67. print("iteration:", iteration, "elapsed time:", time.time() - start, "epsilon:", epsilon, "action:",
68. action\_index.cpu().detach().numpy(), "reward:", reward.numpy()[0][0], "Q max:",
69. 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。

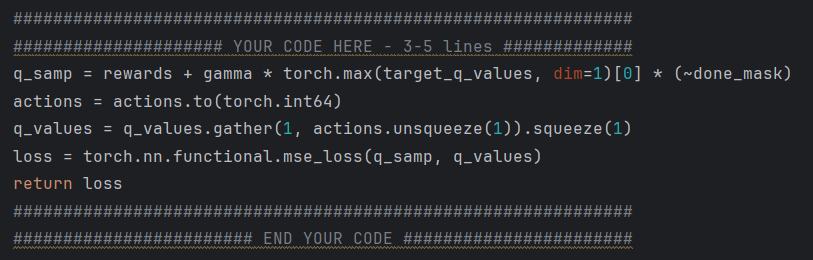
## 代码截图及实现思路

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：

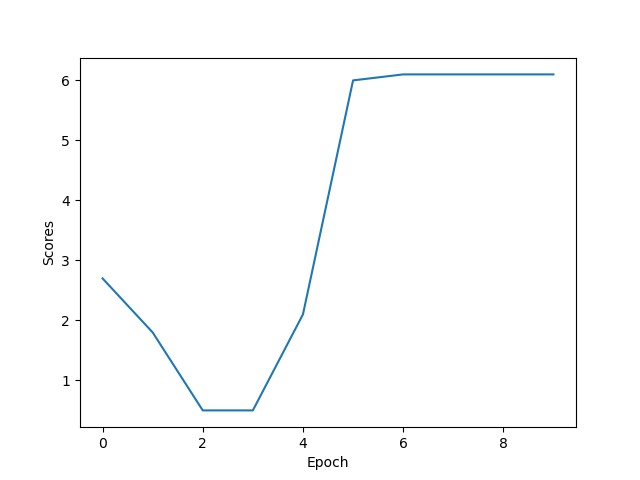




根据代码中提到的Q值函数和loss函数（）的计算公式，建立了以下calc\_loss函数：



## 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。