# 中山大学计算机学院

# 人工智能

# 本科生实验报告

**（2022 学年春季学期）**

课程名称：Artificial Intelligence

| 教学班级 | 计科 2 班 | 专业（方向） | 计算机科学与技术 |
|---|---|---|---|
| 学号 | 20337263 | 姓名 | 俞泽斌 |

## 一、 实验题目

**以 Cart Pole 为环境， 实现 DQN 和 PG 算法， 要求进行可视化(reward, loss, entropy 等）**

## 二、 实验内容

### 1. 算法原理

DQN：

Q-learning 算法采用一个 Q-tabel 来记录每个状态下的动作值并进行比较来算出最佳策略，但在当状态空间或动作空间较大时，需要的存储空间也会较大 DQN 是采取用一个人工神经网络来进行深度强化学习的算法，主要的思想是每一次都判断一下所有下一个状态和动作的价值，然后根据值的大小来选择动作策略。同时为了保证训练的收敛性及稳定性，一般都采用一个经验池来保存已经做完的动作并学习更新人工神经网络，来得到更佳的结果。

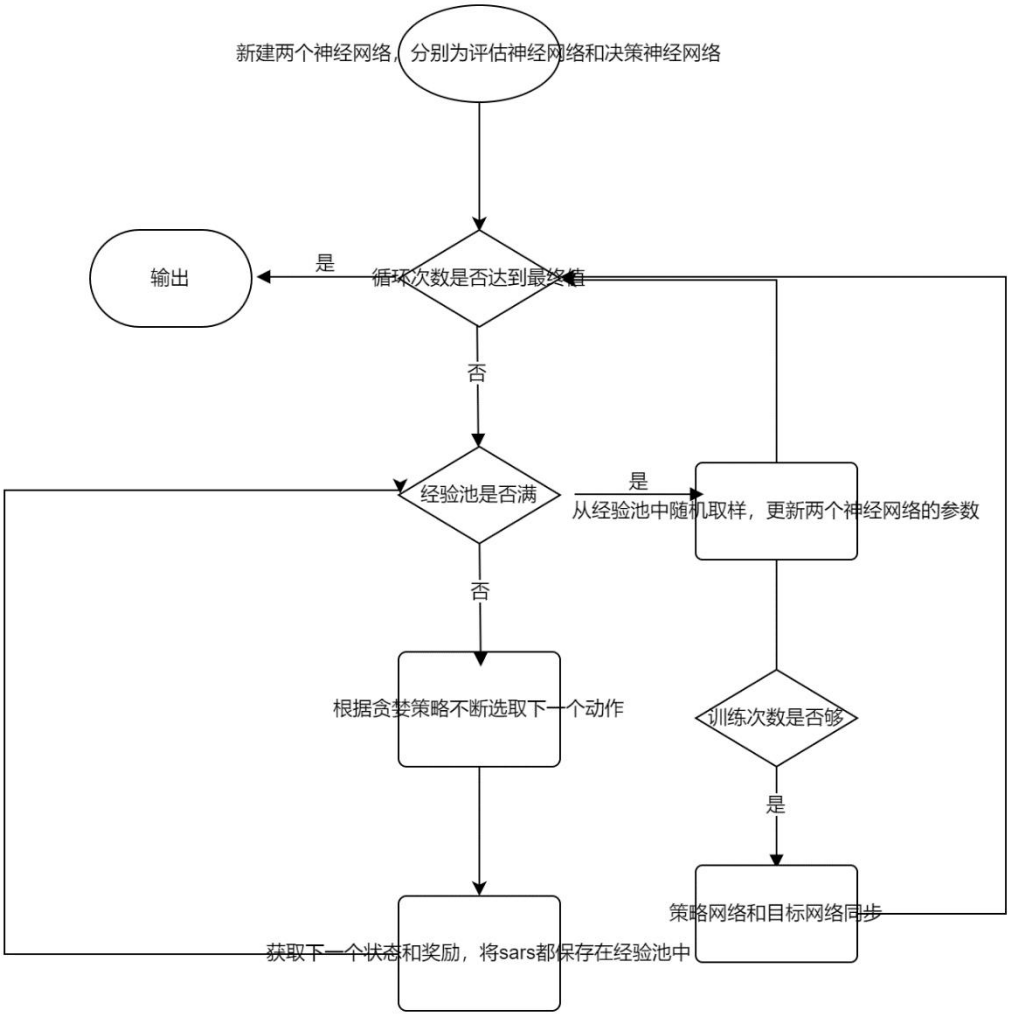
我这里使用了两个神经网络，其中一个是用来估计的，另一个是实际的决策操作，先动作和状态通过估计的神经网络来得到策略，有一个 update_target 参数来判断是否到了取样的结束阶段，如果是就将估计神经网络的参数赋值给决策神经网络，进行下一步的策略是两种，有概率性的随机进行或者是通过评估的神经网络来得到奖励最大的那一个动作，然后每一次进行下一步的时候都会计算并保存损失，差距等参数，然后保存其中的奖励，等到奖励值到一定限度，也就是经验池满了的时候，就在经验池中采样来更新两个神经网络。

PG:

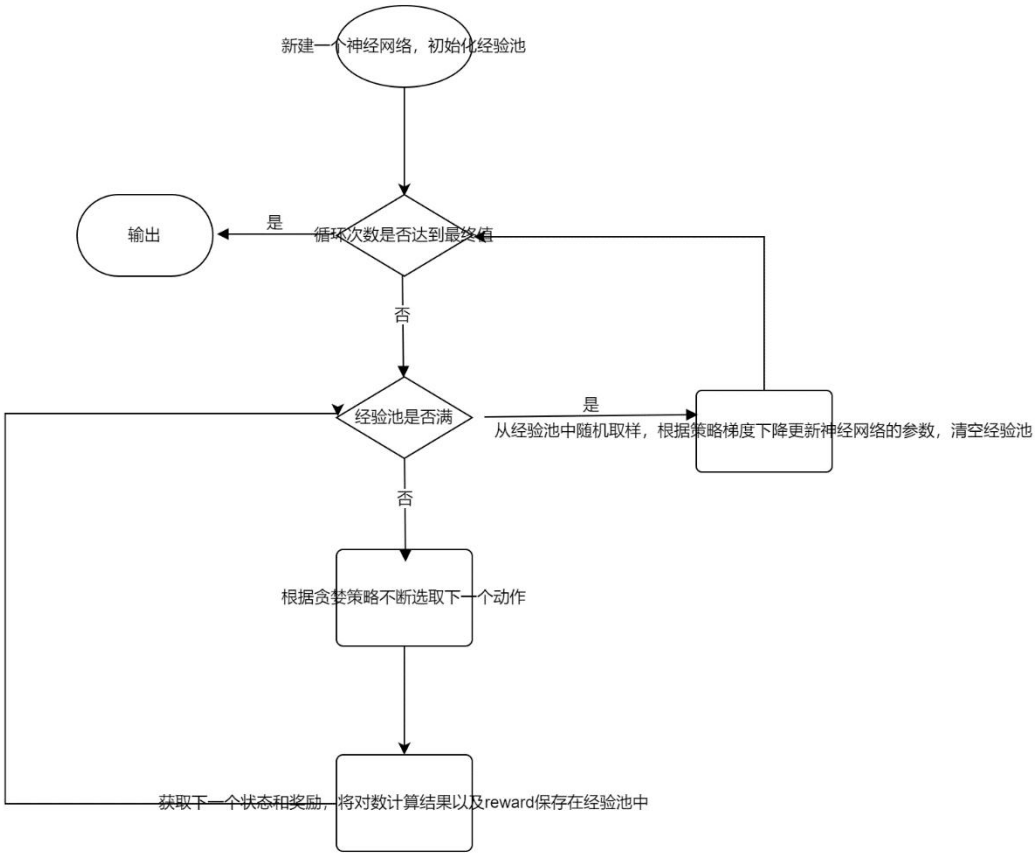
PG 算法直接计算每个状态对应的动作或者动作的概率，使得它可以在一个连续区间内选取动作，他对策略函数进行建模，然后用梯度下降更新网络的参数。因为所需要的是最大化累计奖励的期望值，所以可以选择一个 loss 函数来设成-函数的奖励值与动作策略概率的乘积的对数值，可以方便相关运算，同时在 pytorch 中也有相对应的库函数来调用，如果一个动作的奖励值较大，则下次选取该动作的可能性增加的幅度也大，反之选取该动作的可能性增加的幅度小。

### 2. 伪代码

DQN


新建两个神经网络，分别为评估神经网络和决策神经网络
循环次数是否达到最终值
是
输出
否
经验池是否满
是
从经验池中随机取样，更新两个神经网络的参数
否
根据贪婪策略不断选取下一个动作
训练次数是否够
是
策略网络和目标网络同步
获取下一个状态和奖励，将sars都保存在经验池中

PG：




### 3. 关键代码展示（带注释）

```
import gym
import torch
import torch.nn.functional as F
import numpy as np
from torch import nn, optim
import matplotlib.pyplot as plt
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

首先是一些库函数的引入以及为了提高矩阵计算的速度而使用 GPU 来进行计算，这里使用的是 cuda，具体教程都是参考网上的代码和实现，安装完 cuda 的包后可以在命令行中输入 nvidia-sml 得到对应的 cuda 版本号来代表安装成功

```
C:\Users\Aholic^y>nvidia-smi
Sun Jun 19 21:23:09 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 512.15       Driver Version: 512.15       CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ... WDDM  | 00000000:01:00.0  On |                  N/A |
| N/A   40C    P8     6W /  N/A |    431MiB /  6144MiB |      1%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
```

这样就可以使用 cuda 来调用 pytorch 从而使用 gpu 进行有关的计算和操作，但后来发现 gpu 计算的加速一般都在于参数比较大的情况，而本次实验中参数较为小的时候也可以得到基本收敛的结果，所以还是大部分使用 cpu 来进行计算

```
env = gym.make("CartPole-v0")
lr = 1e-3
n_episodes = 500
alpha = 0.99  # 学习速率
capacity = 5000
eps = 1.0
eps_min = 0.05
hidden = 128
sample_size = 64  # 取样学习大小
eps_decay = 0.99
update_target = 100
lossf = nn.MSELoss()
allloss = []
allreward = []
```

一些基本的参数，学习率，采样次数，采样的大小，随机选择进行下一步的概率等等。

```python
class Net(nn.Module):  # 神经网络
    def __init__(self, input_size, output_size):
        # 初始化神经网络
        super(Net, self).__init__()
        self.func1 = nn.Linear(input_size, hidden)
        self.func2 = nn.Linear(hidden, output_size)

    def forward(self, x):
        x = torch.Tensor(x)
        hide = F.relu(self.func1(x))
        out = self.func2(hide)
        return out
```

基本的神经网络的类，这也是本次实验中唯一封装的一个类，因为在 DQN 算法当中，我们所需要的是两个人工神经网络，一个是估计的神经网络，另一个是实际做决策的神经网络。

```python
def sample(memory, n):  # 取样
    index = np.random.choice(len(memory), n)
    sample_set = []
    for i in index:
        sample_set.append(memory[i])
    return zip(*sample_set)
```

取样函数，因为 DQN 中关于估计神经网络的更新是通过从经验池中取样然后学习的过程来实现的，所以需要一个随机的选取 memory 其中的几个参数的函数，并且将他们封装好发出来。

```python
def next_action(obs, eval_net):
    global eps
    if np.random.uniform() > eps:
        value = eval_net(obs)
        action = torch.max(value, dim=-1)[1].numpy()
    else:
        action = np.random.randint(0, env.action_space.n)
    return int(action)
```

进行下一步的函数，主要操作是通过一个 eps 概率，有概率性的随机操作，或者是通过评估神经网络来获得一个奖励值最大的动作来进行，在 main 函数中会保存这个动作对应的一些信息到经验池中。

```
def push(memory, *transition):
    if len(memory) == capacity:
        memory.pop(0)
    memory.append(transition)
```

这是往经验池中添加元素的操作，因为所添加的东西是很多信息的一个集合，所以不能简单的使用一个 append 函数，然后需要通过传递地址来，同时使用一个 capacity 来判断经验池是否溢出，如果满了就排除最早的那个经验，因为最不重要。

```
def learn(learn_step, target_net, eval_net, memory, optimer):
    global eps
    if learn_step % update_target == 0:
        target_net.load_state_dict(eval_net.state_dict())  # 将评估网络复制到目标网络中
    learn_step += 1
    if eps > eps_min:
        eps *= eps_decay
    # 取样
    obs, actions, rewards, next_obs, dones = sample(memory,sample_size)
    actions = torch.LongTensor(actions)
    dones = torch.IntTensor(dones)
    rewards = torch.FloatTensor(rewards)
    # 更新两个神经网络
    q_eval = eval_net(obs).gather(-1, actions.unsqueeze(-1)).squeeze(-1)
    q_next = target_net(next_obs).detach()
    q_target = rewards + alpha * (1 - dones) * torch.max(q_next, dim=-1)[0]  # Q_tar
    loss = lossf(q_eval, q_target)
    allloss.append(loss.item())
    optimer.zero_grad()
    loss.backward()
    optimer.step()
    return learn_step, target_net, eval_net, eps, optimer, memory
```

这是 DQN 算法里的核心内容了，也就是关于两个网络的更新方面的，前面提到，关于下一步动作，我们一般大概率都是采用的估计网络来输出，当经验池中的步骤和奖励足够多的时候，我们就要开始学习了，从经验池中取样，然后取出里面奖励最大的那个值来更新两个网络的参数，同时如果学习的次数到达了 update_target，就将评估网络复制到目标网络中。

```python
def DQN():
    # env.reset()
    memory=[]
    o_dim = env.observation_space.shape[0]
    a_dim = env.action_space.n
    eval_net = Net(o_dim, a_dim)  # 评估网络
    target_net = Net(o_dim, a_dim)  # 目标网络
    optimer = optim.Adam(eval_net.parameters(), lr=lr)  # 优化器
    learn_step = 0
    for i_episode in range(n_episodes):
        # env.render()
        obs = env.reset()
        episode_reward = 0
        done = False
        while not done:
            action = next_action(obs, eval_net)
            next_obs, reward, done, info = env.step(action)
            # 保存进行状态
            push(memory, obs, action, reward, next_obs, done)
            episode_reward += reward
            obs = next_obs
            if len(memory) >= capacity:
                learn_step, target_net, eval_net, eps, optimer, memory = learn(learn_step, target_net, eval_net, memory,
                                                                               optimer)
        print(f"Episode: {i_episode}, Reward: {episode_reward}")
        allreward.append(episode_reward)
    printfig(allloss, "loss")
    printfig(allreward, "reward")
```

DQN() › for i_episode in range(n_episod...

剩下的方面就是 DQN 函数的处理，其实大部分也就是拼接操作了，首先初始化两个网络为目标和评估网络，然后调用 pytorch 库的那个的 optim 的优化器，在规定的循环次数内，不断进行下一步的操作，并保存相关的经验参数，如果经验的数量超过了经验池的容量，就通过学习来更新两个网络。
最后输出两个图表来展示实验过程中的 reward 和 loss

PG:

```python
import gym
import numpy as np
import torch
import torch.nn.functional as F
from torch import nn, optim, Tensor
from torch.distributions import Categorical
import matplotlib.pyplot as plt
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
lr = 1e-3
hidden = 128
n_episodes = 1200
alpha = 0.99
capacity = 6000
allrewards = []
allloss = []
```

首先依旧是一些基本的库函数的引入以及基本参数的确定。

```
class QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNet, self).__init__()
        self.func1 = nn.Linear(input_size, hidden_size)
        self.func2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = Tensor(x)
        hide = F.relu(self.func1(x))
        out = F.softmax(self.func2(hide), dim=-1)
        return out
```

这是 PG 算法中的神经网络的确定，与 DQN 算法中的神经网络不同的是，这里的隐层到输出层中的操作用的是 softmax 函数，其优点主要是预测的概率为非负数；并且各种预测结果概率之和等于 1，方便之后的策略梯度下降。

```
def sample(memory):
    return zip(*memory)
```

这里的取样只需要取其中一个，就写一个简单的取样函数。

```
def next_action(obs, net):
    prob = net(obs)
    m = Categorical(prob)  # 探索机制同时按照动作概率进行采样
    action = m.sample()  # 根据采样选择动作
    log_prob = m.log_prob(action)  # 策略梯度函数更新时的log项，之后只需要乘以alpha*reward即可
    return int(action), log_prob
```

下一步的操作，我们这里所需要保存的参数有两个，一个是 reward，另一个是我们策略梯度下降的时候有一个对数项（-函数的奖励值与动作策略概率的乘积的对数值）要每次先把他计算出来保存，方便更新的时候可以直接取出来然后进行计算。

```python
def learn(memory, optimer):
    log_probs, rewards = sample(memory)
    log_probs = torch.stack(log_probs)
    discounts = [alpha ** i for i in range(len(rewards) + 1)]
    Ret = sum([a * b for a, b in zip(discounts, rewards)])
    loss = -Ret * log_probs
    allloss.append(loss.sum().item())
    optimer.zero_grad()
    loss.sum().backward()
    optimer.step()
```

学习函数，主要也是在经验池满的时候开始进行操作，从我们上述的下一步函数可以看出，我们每进行下一步，就会计算一次 reward 和对数值并保存到经验池中，现在我们就取样并进行学习，主要操作是计算每一次的折扣函数和 reward 的乘积从而得到对数值，然后计算出 loss，并且通过 loss 来返回修改神经网络中的几个参数。

```python
def main():
    memory = []
    env = gym.make("CartPole-v0")
    o_dim = env.observation_space.shape[0]
    a_dim = env.action_space.n
    net = QNet(o_dim, hidden, a_dim)
    optimer = optim.Adam(net.parameters(), lr=lr)
    for i_episode in range(n_episodes):
        obs = env.reset()
        episode_reward = 0
        done = False
        while not done:
            action, log_prob = next_action(obs, net)
            next_obs, reward, done, info = env.step(action)
            push(memory,log_prob, reward)
            '''
            if len(memory) == capacity:
                memory.pop(0)
            memory.append(log_prob)
            memory.append(reward)
            '''
            obs = next_obs
            episode_reward += reward
        learn(memory, optimer)
        memory.clear()
        allrewards.append(episode_reward)
        print(f"Episode: {i_episode}, Reward: {episode_reward}")
    printfig(allloss, "loss")
    printfig(allrewards, "reward")
```

这是主体的代码，主要的操作也就是和 DQN 一样的拼接了，首先初始化神经网络，然后调用 pytorch 库的那个的 optim 的优化器，在规定的循环次数内，不断进行下一步的操作，并保存相关的 reward 和对数参数，如果经验的数量超过了经验池的容量，就通过学习来计算策略梯度并更新网络。

**4. 创新点&优化（如果有）**

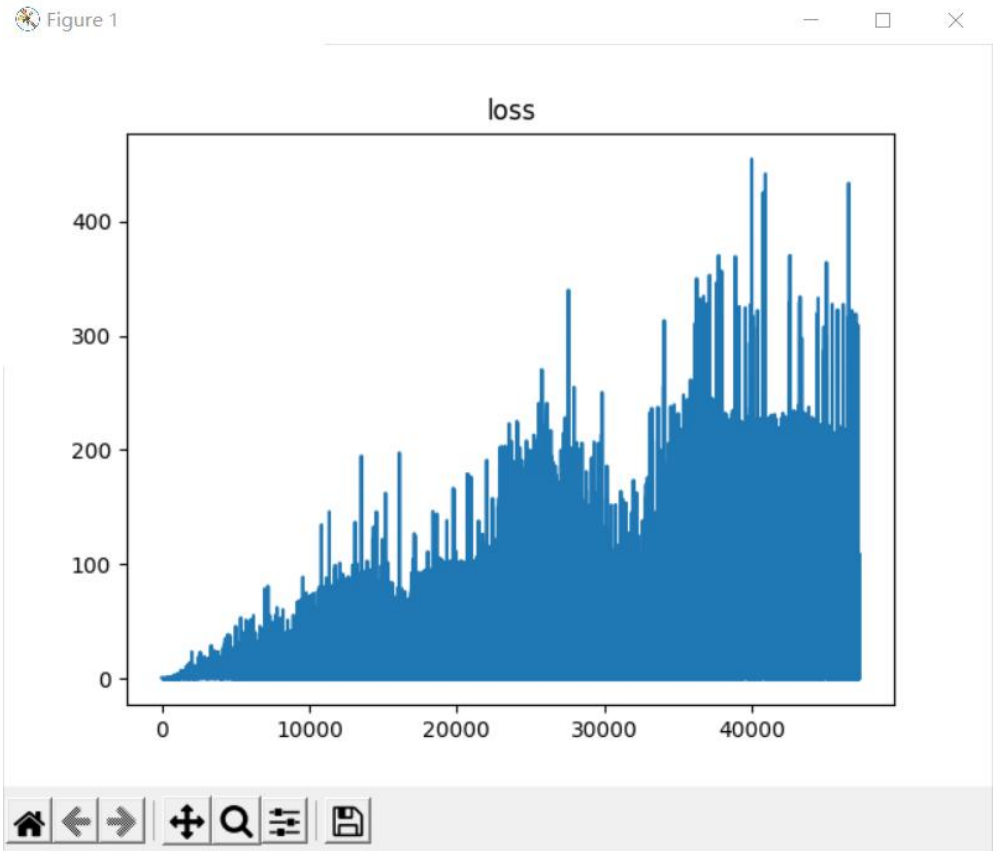
```
torch.manual_seed(0)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

尝试采用了 cuda 来调用 pytorch 从而使用 gpu 进行有关的计算和操作，但后来发现 gpu 计算的加速一般都在于参数比较大的情况，而本次实验中参数较为小的时候也可以得到基本收敛的结果，所以还是大部分使用 cpu 来进行计算

# 三、 实验结果及分析

**1. 实验结果展示示例（可图可表可文字，尽量可视化）**

## DQN



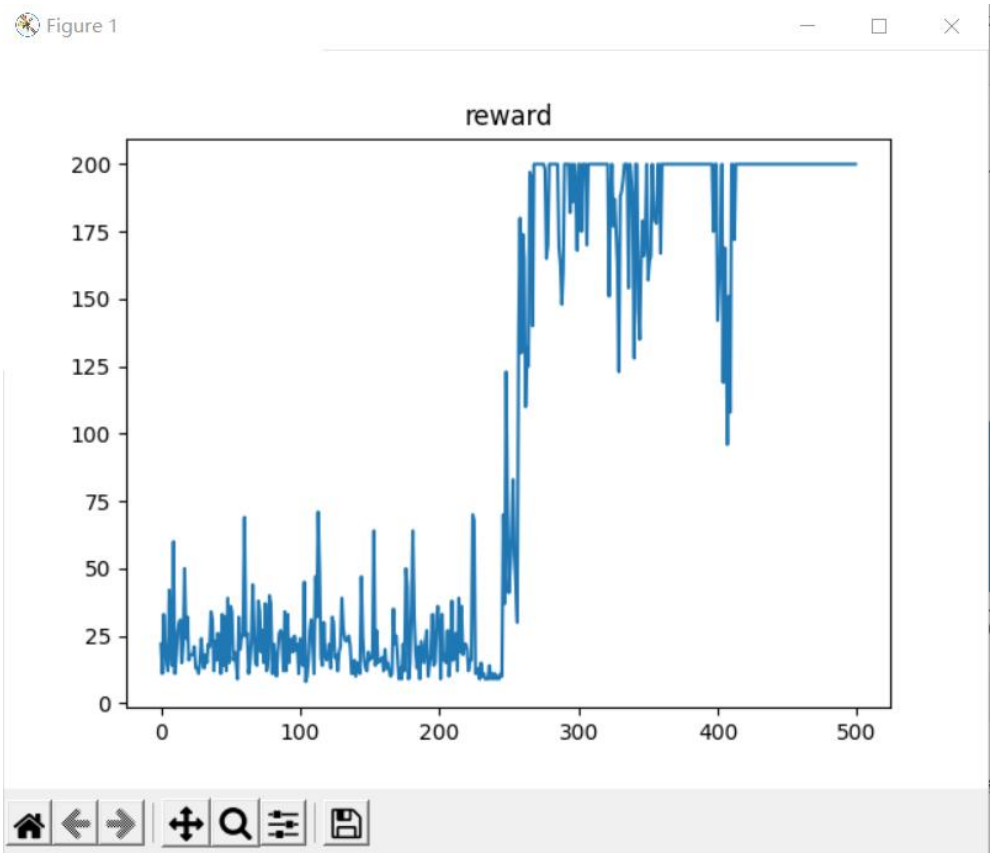


Figure 1
reward
200
175
150
125
100
75
50
25
0
0
100
200
300
400
500

PG

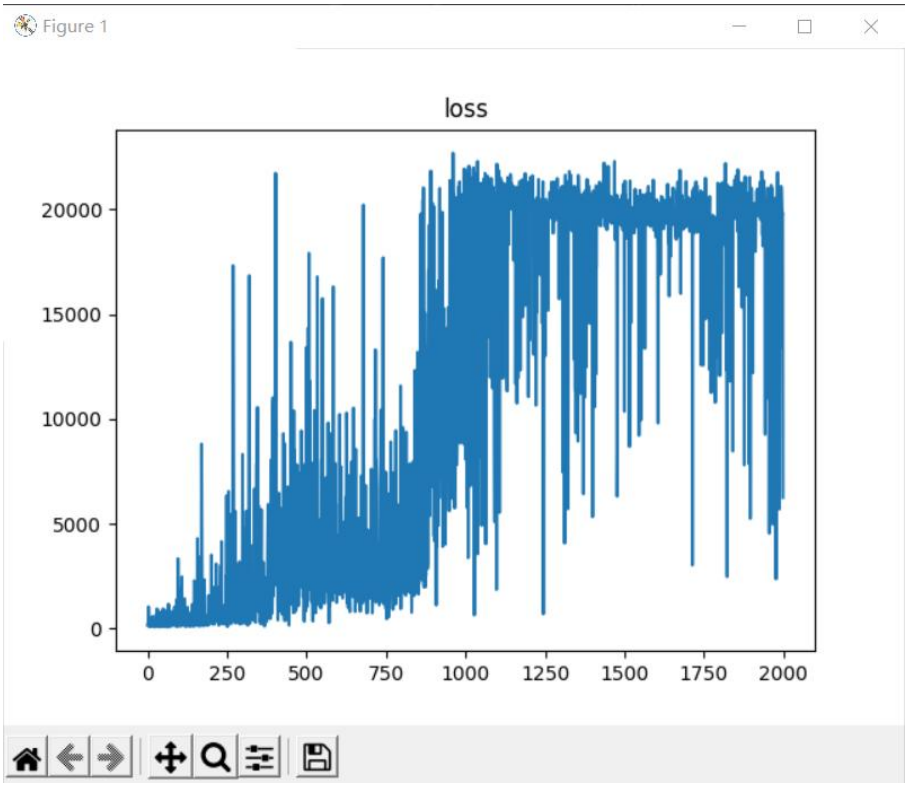
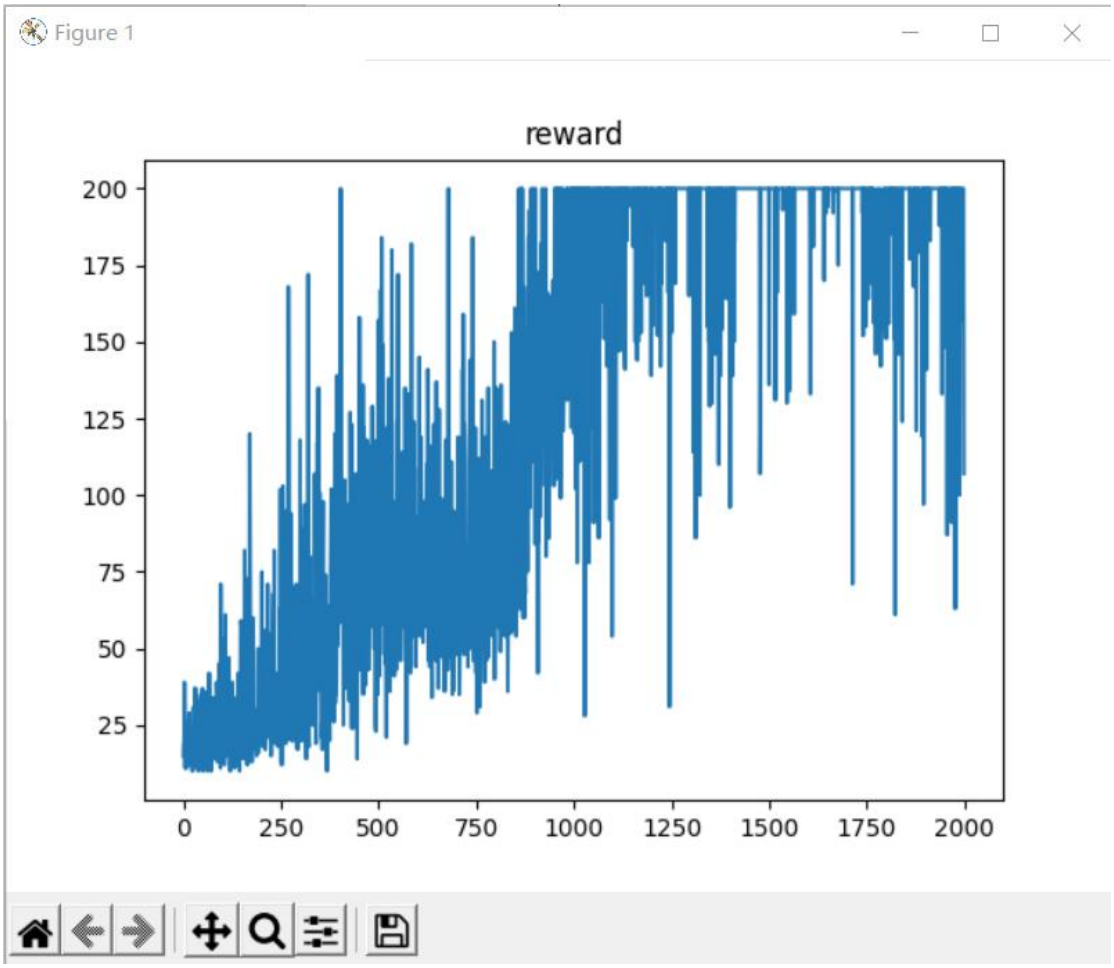


Figure 1
loss
20000
15000
10000
5000
0
0
250
500
750
1000
1250
1500
1750
2000




**2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）**

可以看到，本次实验中 DQN 算法比起 PG 算法来说，DQN 算法更快收敛，并且收敛后的波动较小，应该是因为更新的是两个网络中的参数，同时取样的时候是取了很多的经验池样本来学习，比起 PG 算法的一个网络以及学习时值抽取少量的样本，DQN 算法更容易收敛。但同时是因为上述的两个原因，DQN 算法所需要取样的以及计算的步骤就特别多，导致 DQN 算法的速度可以说是慢了许多，特别是运行到后面更甚，PG 算法计算操作少，运行速度更快。

DQN：循环次数 500 次

```
耗时:460.34375s
```

PG：循环次数 2000 次


**|----------如有优化，请重复 1，2，分析优化后的算法结果----------|**

# 四、 参考资料

**https://blog.csdn.net/lz_peter/article/details/84574716?ops_request_misc=%257B%2522request%255Fid%2522%253A%25221655683252167814354653 80%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165568325216781435465380&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-2-84574716-null-null.142^v17^control,157^v15^new_3&utm_term=softmax&spm=1018.2226.3001.4187**

**https://blog.csdn.net/zuzhiang/article/details/103180919?ops_request_misc=%257B%2522request%255Fid%2522%253A%25221655644615167814837208 88%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165564461516781483720888&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~baidu_landing_v2~default-2-103180919-null-null.142^v17^control,157^v15^new_3&utm_term=pg%E7%AE%97%E6%B3%95%E5%8E%9F%E7%90%86&spm=1018.2226.3001.4187**

**https://github.com/joenghl/BaseRL/tree/da2b68e4bc30d18c0791c5c71a33a083ff5e3b1f**

PS：可以自己设计报告模板，但是内容必须包括上述的几个部分，不需要写实验感想