# 策略梯度实验报告

1120210526 马煜

## 参考文献：

Reinforce:

<https://goodboychan.github.io/python/reinforcement_learning/pytorch/udacity/2021/05/12/REINFORCE-CartPole.html>

<https://hrl.boyuai.com/chapter/2/%E7%AD%96%E7%95%A5%E6%A2%AF%E5%BA%A6%E7%AE%97%E6%B3%95>

<https://datawhalechina.github.io/easy-rl/#/chapter4/chapter4>

PPO：

<https://spinningup.openai.com/en/latest/algorithms/ppo.html>

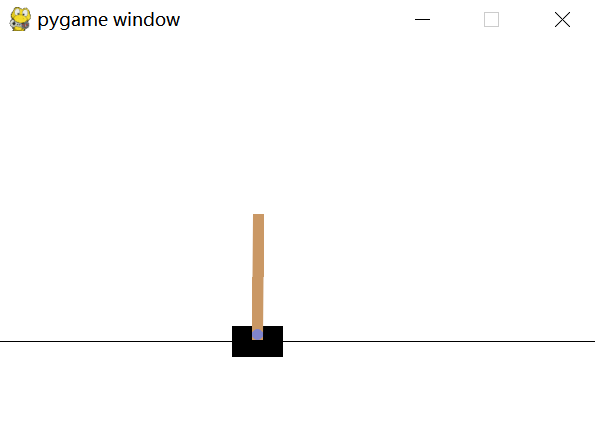
<https://datawhalechina.github.io/easy-rl/#/chapter5/chapter5>

## reinforce

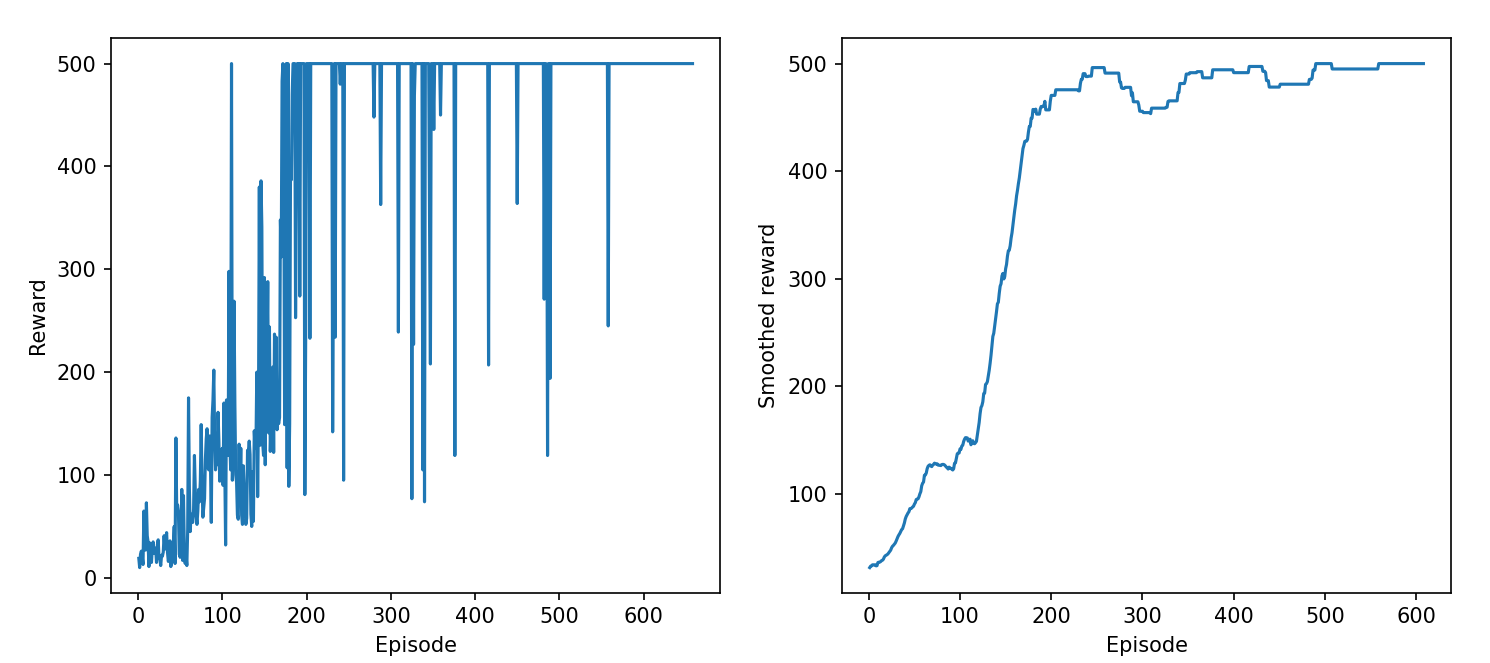
代码见reinforce.py

###### 实验结果展示

命令行输出见**RE.txt**



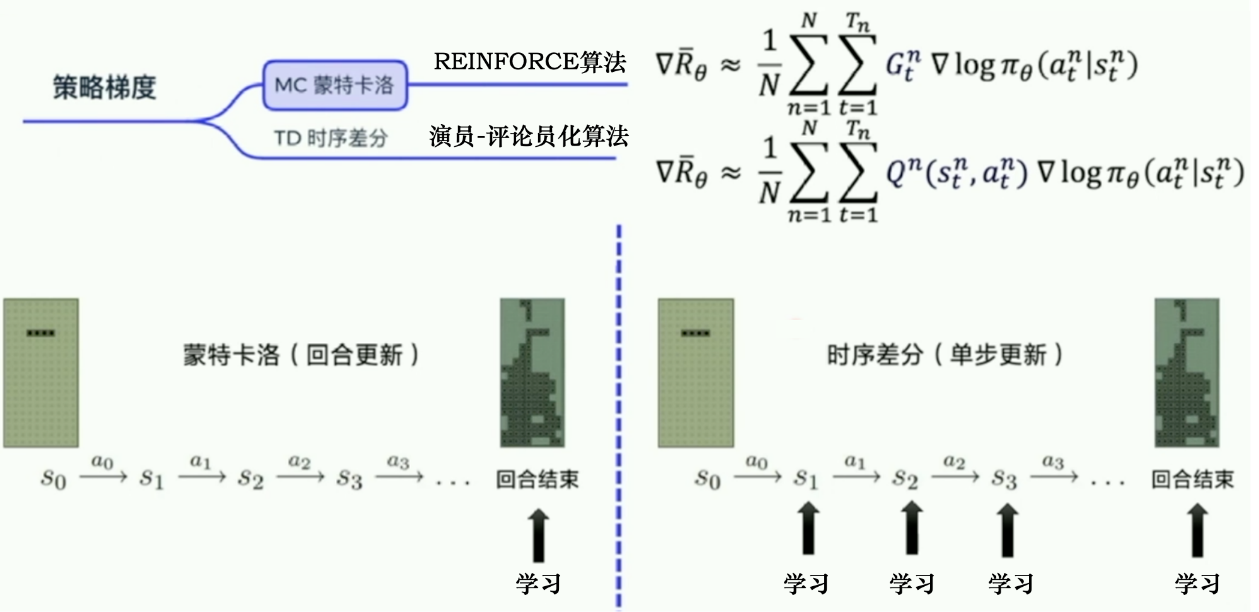
最终稳定效果



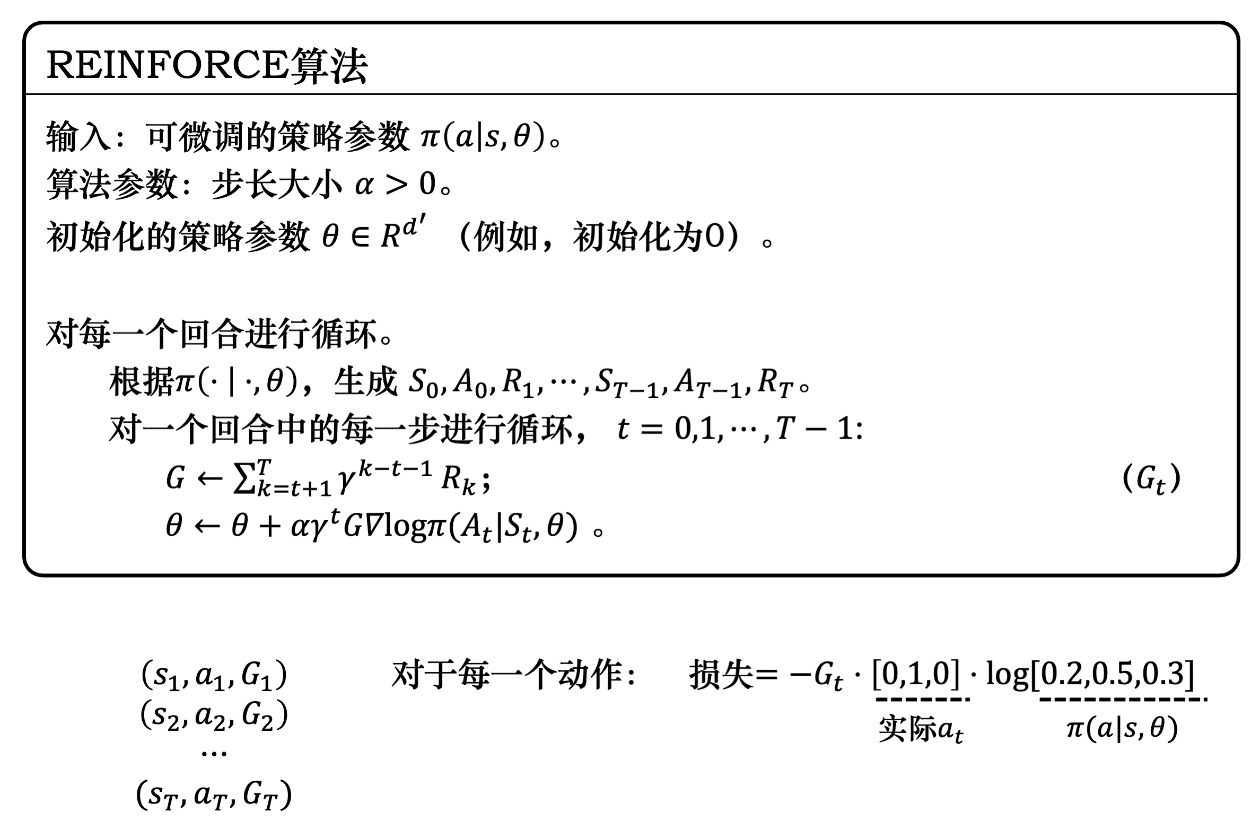
回报图像

可以看到结果最终收敛的很好而且很稳定，在训练初期稳步上升，伴随微小波动

###### 实验过程

REINFORCE 用的是回合更新的方式，它在代码上的处理上是先获取每个步骤的奖励，然后计算每个步骤的未来总奖励

算法伪代码如下：



下面我来解释代码块的功能：

Policy类包装了神经网络结构，采用了两层全连接，隐藏层大小为16。层间使用relu进行激活，最后输出使用softmax得到动作的概率分布。

Act方法当前策略网络输出根据概率分布采样的动作和动作对应的对数概率（这个对数概率相当于伪代码中对数概率点乘独热向量的结果）

Pre方法选择概率最高的动作，作为最终演示输出的策略。

class Policy(nn.Module):

    def **\_\_init\_\_**(*self*, *s\_size*=4, *h\_size*=16, *a\_size*=2):

        super(Policy, *self*).**\_\_init\_\_**()

*self*.fc1 = nn.Linear(*s\_size*, *h\_size*)  *# 全连接层，将输入状态映射到隐藏层*

*# self.fc2 = nn.Linear(h\_size, h\_size)  # 第二个隐藏层，没用到*

*self*.fc3 = nn.Linear(*h\_size*, *a\_size*)  *# 将隐藏层的输出映射到输出层*

    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 F.**softmax**(*x*, *dim*=1)  *# 对分数进行 Softmax 操作，得到动作的概率分布*

    def **act**(*self*, *state*):

*state* = torch.**from\_numpy**(*state*).**float**().**unsqueeze**(

            0).**to**(device)  *# 将环境传递的状态转换为 PyTorch 张量*

        probs = *self*.**forward**(*state*).**cpu**()  *# 使用策略网络的前向传播计算动作的概率分布*

*# Categorical 分布是离散概率分布的一种，用于描述随机变量的可能取值集合是有限的情况。*

        m = Categorical(probs)

*# 在强化学习中，通常用来表示在一个离散动作空间中选择动作的概率分布。*

        action = m.**sample**()  *# 从概率分布中采样一个动作（考虑每个动作的概率）*

*# 返回选定的动作及其动作的对数概率，对数概率将在策略梯度算法中用于梯度更新。*

        return action.**item**(), m.**log\_prob**(action)

    def **pre**(*self*, *state*):

*state* = torch.**from\_numpy**(*state*).**float**().**unsqueeze**(

            0).**to**(device)  *# 将环境传递的状态转换为 PyTorch 张量*

        probs = *self*.**forward**(*state*).**cpu**()  *# 使用策略网络的前向传播计算动作的概率分布*

*# Categorical 分布是离散概率分布的一种，用于描述随机变量的可能取值集合是有限的情况。*

        m = Categorical(probs)

*# 选择概率最高的一个动作*

        action = torch.**argmax**(probs)

        return action.**item**(), m.**log\_prob**(action)

使用reinforce算法进行训练，**重点讨论一下反向传播部分（这里也只展示了这部分代码）。**

可以看到反向传播部分有注释掉的和实际使用的。

在注释中的方法中，首先计算了累积奖励 R，然后将每个时间步的策略损失乘以相同的 R。这种方法是将所有时间步的策略损失一起考虑，然后一次性进行梯度计算和参数更新。

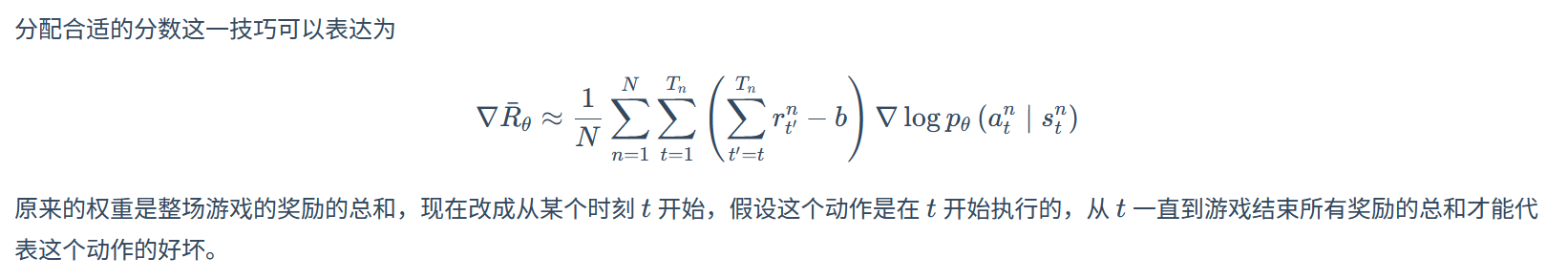
在实际的方法中，在每个时间步上分别计算策略损失，并逐步累积梯度：

实际方法使用一个循环来遍历每个时间步。在每个时间步上，计算了当前时间步的奖励 reward 和动作对数概率 log\_prob。

累积奖励 G 是在循环中逐步更新的，通过 G = gamma \* G + reward 计算。这个 G 表示从当前时间步起的累积奖励。

在每个时间步上，计算了当前时间步的策略损失 loss，并将其乘以 G。这是一种逐步计算策略损失的方式。

使用 loss.backward() 来计算当前时间步的梯度，然后在每个时间步上进行参数更新。

实际使用的方法对应蘑菇书技巧2分配合适分数中，如下图所示：

def **reinforce**(*n\_episode*=10000, *max\_t*=1000, *gamma*=0.99, *print\_every*=10):

*……*

*# discounts = [gamma\*\*i for i in range(len(rewards)+1)]#计算折扣因子数组 discounts*

*# R = sum([a\*b for a, b in zip(discounts, rewards)])# 将 discounts 和 rewards 列表逐一组合成一个元组的迭代器，也就是在每个时间步上，它会生成一个包含折扣因子和奖励的元组。例如，如果 discounts 为 [0.1, 0.01, 0.001]，rewards 为 [1, 2, 3]，那么 zip 会生成迭代器 (0.1, 1), (0.01, 2), (0.001, 3)*

*# policy\_loss = []*

*# for log\_prob in saved\_log\_porbs:*

*#     policy\_loss.append(-log\_prob \* R)#这个对数概率在计算的时候已经考虑了独热向量，所以这里不需要再乘以独热向量，直接乘以 R 即可*

*# policy\_loss = torch.cat(policy\_loss)*

*# policy\_loss = policy\_loss.sum()#计算策略损失*

*# optimizer.zero\_grad()#清零优化器梯度缓冲区，PyTorch 默认情况下会保留之前的梯度信息*

*# policy\_loss.backward()#自动计算策略损失相对于网络参数的梯度*

*# optimizer.step()#更新网络的参数*

        G = 0

        optimizer.**zero\_grad**()

        for i in reversed(range(**len**(rewards))):

*# d= np.mean(rewards[:])*

            reward = rewards[i]

            log\_prob = saved\_log\_porbs[i]

            G = *gamma* \* G + reward  *# - d*

            loss = -log\_prob \* G

            loss.backward()

        optimizer.**step**()

……

以下代码块固定了种子，保证结果在不同设备上的可复现性，在两个实验中都添加了这个代码

seed = 520

    env.seed(20020917)

    np.random.seed(seed)

    torch.**manual\_seed**(0)  *# cpu*

    torch.cuda.**manual\_seed**(0)  *# GPU*

*# 设置 Python 的哈希种子。*

*# 在某些情况下，Python 的哈希值可以影响一些数据结构的顺序，通过设置这个种子，可以确保在不同运行中哈希值的计算是一致的。*

    os.environ['PYTHONHASHSEED'] = str(seed)

*# 控制 PyTorch 中的 cuDNN 库的行为*

*# 此选项确保 cuDNN 使用确定性算法执行操作。这意味着相同的操作在不同运行中将生成相同的结果。*

*# 这对于实现训练的可重现性非常重要，因为在深度学习中，有些优化可能涉及到随机性操作，例如权重初始化*

    torch.backends.cudnn.deterministic = True

*# cuDNN 尝试自动寻找最适合你的硬件的实现以获得最佳性能。它会根据硬件性能特征选择不同的算法，以获得最佳的速度。*

*# 如果你的输入大小或类型在运行时改变，那么它可能会选择不同的算法，这会导致性能波动。*

    torch.backends.cudnn.benchmark = False

*# 禁用 cuDNN 可能会降低性能，但有时它对于特定任务和硬件配置是必要的，特别是在需要更高可重现性的情况下。*

    torch.backends.cudnn.enabled = False

在主函数中创建了实验环境

定义了策略网络和优化器（使用adam优化器）

调用训练算法训练

env = gym.make('CartPole-v1')  *# 创建环境*

policy = Policy().**to**(device)

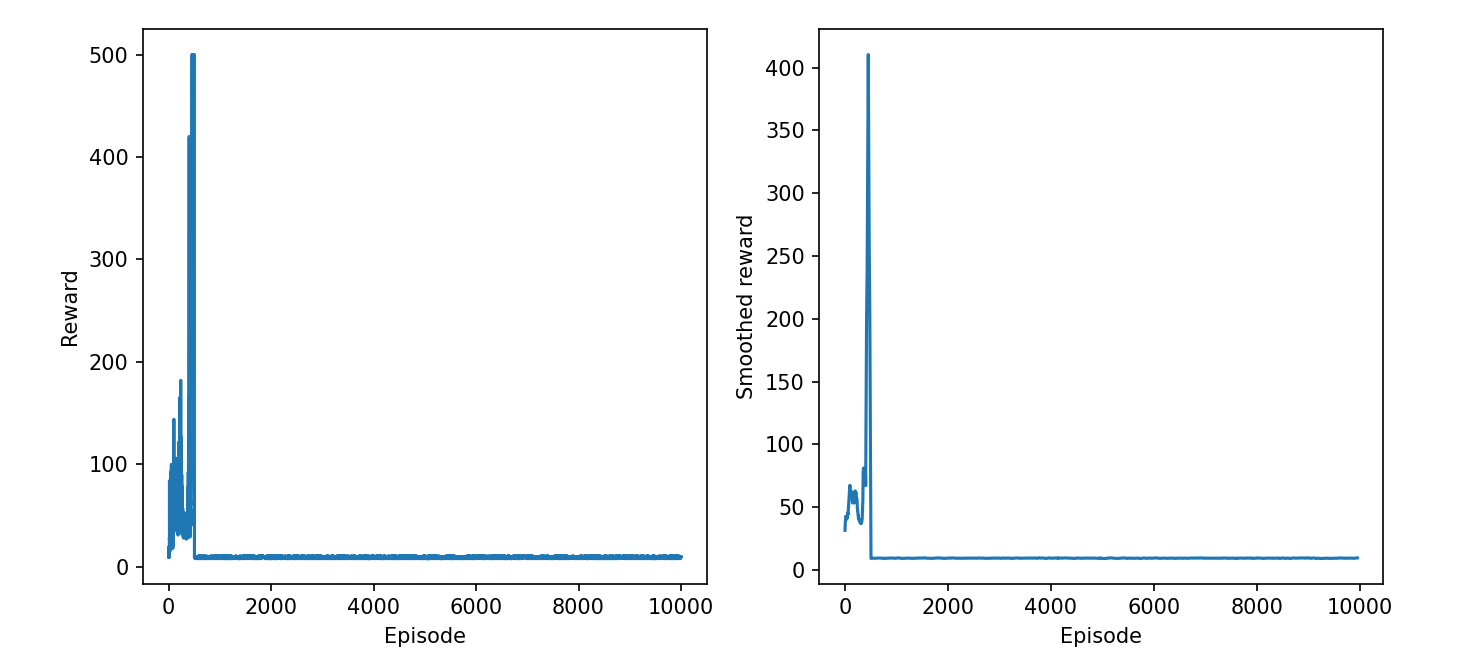
*# policy.parameters()返回策略网络 policy 中的所有可学习参数 ；lr 学习率*

    optimizer = optim.Adam(policy.**parameters**(), *lr*=1e-2, *betas*=[0.9, 0.999])

    scores = **reinforce**()  *# 使用reinforce算法进行训练*

###### 问题讨论

在实验的过程中，我发现网络过大（如将隐藏层设置为256，结果见下图），无助于网络的训练，可能出现了梯度消失，导致算法越学越傻。



在反向传播部分，一开始我使用的是整个回合同一个R，效果很不好，收敛很慢，改为逐步计算损失然后累加梯度，最后再更新网络参数

## PPO：

代码见PPO\_clip\_no\_critic.py

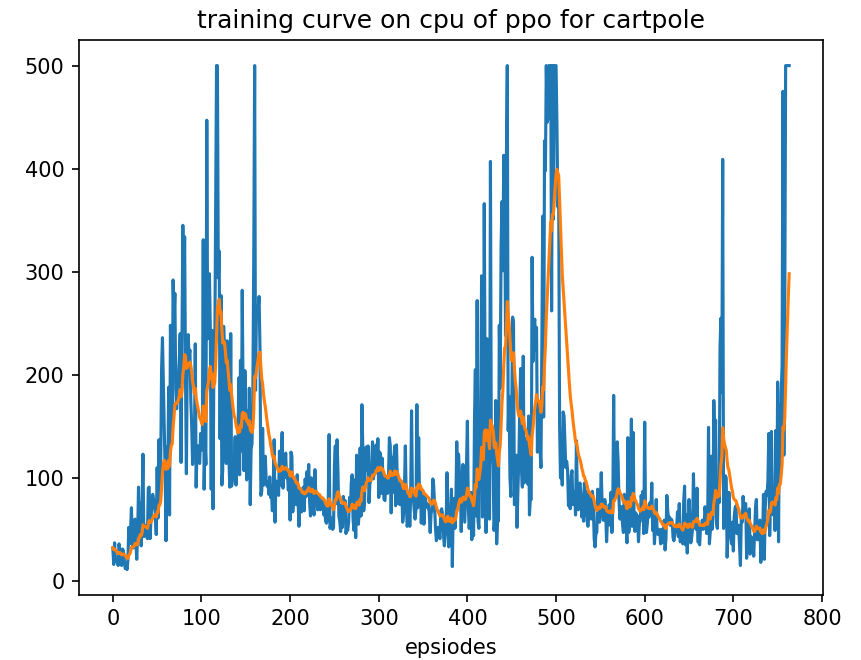
###### 说明

本实验没有使用网上的PPO的actor-critic形式，而使用的蘑菇书近端策略优化的方法，也就是用两个agent(在代码中是用同一个agent来代替，agent先采样很多步，放在回放队列中（记录此时的对数概率），然后同一个agent再学习这些采样，计算新的策略和对数概率……)

同时，使用近端策略优化裁剪PPO2来避免两个模型差别过大

###### 实验结果

输出见PPO.txt



因为参数调节不是很合理，PPO算法训练的波动很大，但最终也可以获得可以完美运行的策略

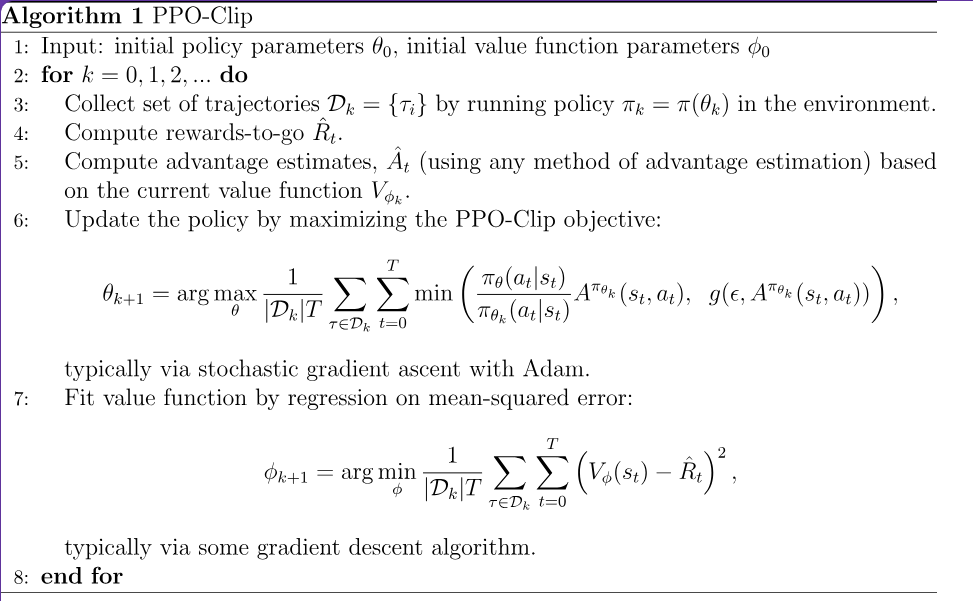
###### 实验过程

**PPO-Clip** 在目标中没有 KL 背离项，并且根本没有约束。相反，依靠目标函数中的专门剪裁来消除新政策远离旧政策的激励。同时，openAI的PPO也是用的这个方法。

以下是伪代码，在计算A的时候我使用的方法是将旧状态的价值平均作为基线：

values = returns.**mean**() *#计算旧状态的价值平均作为基线*

advantage = returns - values



PPO的网络部分和RE很像，这里我的隐藏层比较深，来配合PPO更强劲的性能

class Actor(nn.Module):

    def **\_\_init\_\_**(*self*, *input\_dim*=4, *hidden\_dim*=256, *output\_dim*=2):

        super(Actor, *self*).**\_\_init\_\_**()

*self*.fc1 = nn.Linear(*input\_dim*, *hidden\_dim*)

*self*.fc2 = nn.Linear(*hidden\_dim*, *hidden\_dim*)

*self*.fc3 = nn.Linear(*hidden\_dim*, *output\_dim*)

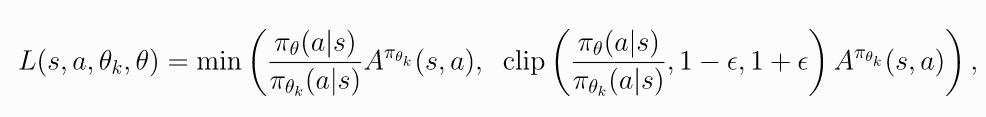
    def **forward**(*self*, *x*):

*x* = F.**relu**(*self*.fc1(*x*))

*x* = F.**relu**(*self*.fc2(*x*))

        probs = F.**softmax**(*self*.fc3(*x*), *dim*=1)

        return probs

在更新部分，根据公式得到目标函数，反向传播

for \_ in range(*self*.K\_epochs):

*#迭代k次更新*

*# 使用PPO-clip*

            values = returns.**mean**() *#计算旧状态的价值作为基线*

            advantage = returns - values

*# get action probabilities*

            probs = *self*.actor(old\_states)*# 使用 Actor 模型计算旧状态的动作概率*

            dist = Categorical(probs) *#创建一个 Categorical 分布对象，用于计算动作的概率*

*# get new action probabilities*

            new\_probs = dist.**log\_prob**(old\_actions)

*# compute ratio (pi\_theta / pi\_theta\_\_old):*

            ratio = torch.**exp**(new\_probs - old\_log\_probs) *# 计算策略比例（pi\_theta / pi\_theta\_old），这是新策略和旧策略的比值。old\_log\_probs 必须是已分离的，以避免梯度传播到旧策略。*

*# compute surrogate loss*

            surr1 = ratio \* advantage

            surr2 = torch.**clamp**(ratio, 1 - *self*.eps\_clip, 1 + *self*.eps\_clip) \* advantage

            loss = -torch.**min**(surr1, surr2).**mean**()

*self*.actor\_optimizer.**zero\_grad**()

            loss.**backward**()

*self*.actor\_optimizer.**step**()

训练中，每次采样增加计数器

*self*.sample\_count += 1

当采样数大于*update\_freq才会执行更新*

def **update**(*self*):

*#执行策略更新的方法，在update\_freq次后执行*

        if *self*.sample\_count % *self*.update\_freq != 0:

            return*# 没到update\_freq次时不执行*

采样过程和RE很像，只不过采样完不需要立刻算损失，而是将采样结果保存在回放队列中

for \_ in range(max\_steps):

            ep\_steps += 1

            action, log\_probs = *agent*.sample\_action(state) *#采样动作*

            next\_state, reward, done, \_ = *env*.step(action) *#执行动作，更新环境*

*agent*.memory.push([state, action, log\_probs, reward, done]) *#保存transition*

            state = next\_state *#state更新到下一个状态*

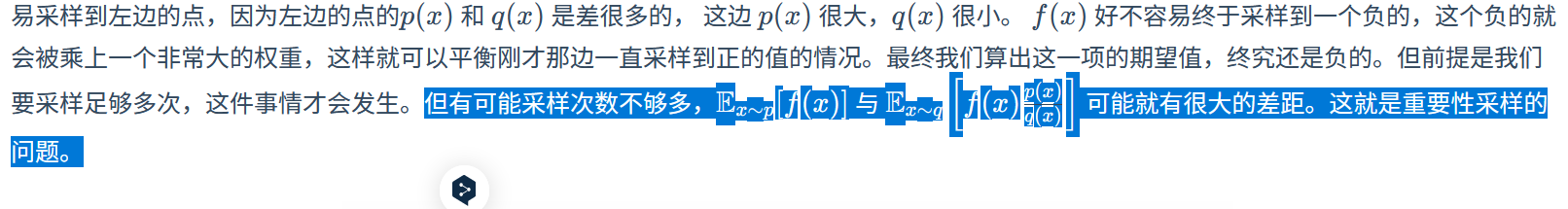
*agent*.update() *#更新智能体(每update\_freq次采样后更新)*

            ep\_reward += reward *#累加回合奖励*

            if done:

                break *#回合结束，跳出循环*

###### 问题讨论

在实验过程中，我发现，如果将update\_freq设置的很小时，网络不会学习到任何知识，可能的原因如下：**