基于**CartPole-v0**环境的**PolicyGradient**代码实现

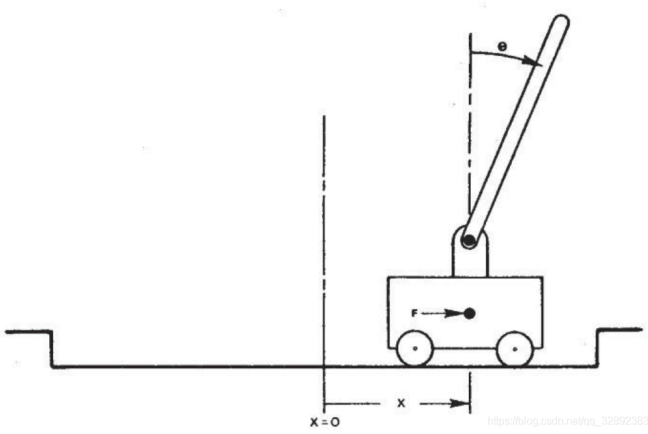
作者：LDJ

CartPole-v0环境介绍

具体的大背景我就不再重复，也就是在**gym**第三方库，**python3.7.6**，**torch1.4**。

Cart Pole即车杆游戏，游戏模型如下图所示。游戏里面有一个小车，上有竖着一根杆子，每次重置后的初始状态会有所不同。小车需要左右移动来保持杆子竖直，为了保证游戏继续进行需要满足以下两个条件：

- 杆子倾斜的角度θ不能大于15°
- 小车移动的位置x需保持在一定范围（中间到两边各2.4个单位长度）



动作（**action**）：

- 左移（0）
- 右移（1）

状态变量（**state variables**）：

- x：小车在轨道上的位置（position of the cart on the track）
- θ：杆子与竖直方向的夹角（angle of the pole with the vertical）
- $\dot{x}$：小车速度（cart velocity）
- $\dot{\theta}$：角度变化率（rate of change of the angle）

游戏奖励（**reward**）：

在*gym*的*Cart Pole*环境（*env*）里面，左移或者右移小车的*action*之后，*env*会返回一个+1的*reward*。其中*CartPole-v0*中到达200个*reward*之后，游戏也会结束。

以上便是该环境的介绍，我们的目标就是从0开始训练这个环境使得满足上面的两个条件以达到完成目标的操作。

策略梯度

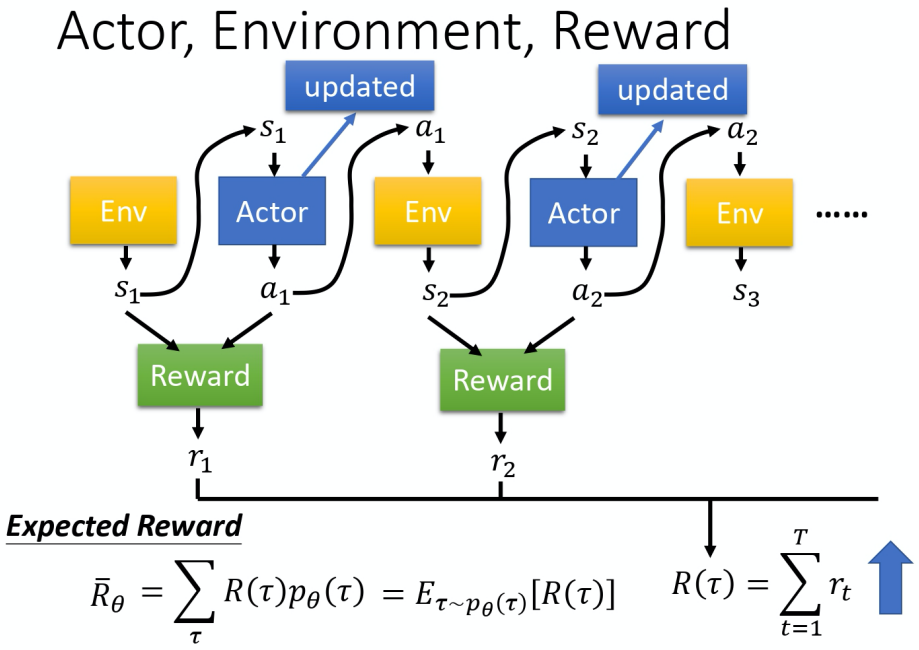
在强化学习里面，环境跟奖励函数不是我们可以控制的，环境跟奖励函数是在开始学习之前，就已经事先给定的。我们唯一能做的事情是调整$actor$里面的策略，使得$actor$可以得到最大的奖励。$actor$里面会有一个策略，这个策略决定了$actor$的动作。策略就是给一个外界的输入，它会输出$actor$现在应该要执行的动作。

策略一般写成 π。假设我们是用深度学习的技术来做强化学习的话，策略就是一个网络。这里的环境下，我们会使用最简单的多层感知机的神经网络进行训练。

这里需要注意的是：环境的动作是指环境的函数内部的参数或内部的规则长什么样子，环境这一项通常你是无法控制它的，因为这个是人家写好的，你不能控制它。放到这个地方，也就是说我们找了gym设定好的一个十分简单的环境，然后我这就是要在这个环境下进行一个抉择。智能体的动作是指你能控制，给定一个 s_t，$actor$要采取什么样的 a_t 会取决于$actor$的参数 θ，所以这部分是$actor$可以自己控制的。随着$actor$的动作不同，每个同样的轨迹，它就会有不同的出现的概率。

上面能控制的部分也就是我们操作的代码的核心，我们主要思路就是通过**MLP**多层感知机进行一个策略的概率输出，然后根据概率进行一个$action$的选择，由于$action$是**{0,1}**的，所以为了达到随机性我们可以用伯努利函数进行将**MPL**输出的概率转换成**{0,1}**的$action$。进而进行采样，这一部分会在$agent.py$的**PolicyGradient**中实现。

那么这里关键问题是我们如何判断好坏呢？这里奖励的重要性就不言而喻了。根据奖励，我们就可以得到目标函数，也就是有方向了。R (回报)是一个随机变量。我们能够计算的是 R 的期望值。



上图中，我们就可以发现其实对于回报是一个随机变量，很自然的我们就会想到求期望进行表示。

$$\bar{R}_\theta = \sum_\tau R(\tau) p_\theta(\tau) = E_{\tau \sim p_\theta(\tau)}[R(\tau)] \tag{1}$$

怎么最大化期望奖励呢？我们用的是 梯度上升(gradient ascent)，因为要让它越大越好，所以是梯度上升。梯度上升在更新参数的时候要加。要进行梯度上升，我们先要计算期望的奖励(expected reward) \bar{R}- 的梯度。我们对 $\bar{R}$取一个梯度，这里面只有$p_\theta(\tau)$是跟 θ有关，所以梯度就放在 $p_\theta(\tau)$这个地方。$R(\tau)$这个奖励函数不需要是可微分的(differentiable)，这个不影响我们解接下来的问题。举例来说，如果是在 GAN 里面，$R(\tau)$其实是一个 discriminator，它就算是没有办法微分，也无所谓，你还是可以做接下来的运算。

Policy Gradient

$$\bar{R}_\theta = \sum_\tau R(\tau)p_\theta(\tau) \qquad \nabla\bar{R}_\theta = ?$$

$$\nabla\bar{R}_\theta = \sum_\tau R(\tau)\nabla p_\theta(\tau) = \sum_\tau R(\tau)p_\theta(\tau)\frac{\nabla p_\theta(\tau)}{p_\theta(\tau)}$$

$R(\tau)$ do not have to be differentiable

It can even be a black box.

$$\nabla f(x) = f(x)\nabla log f(x)$$

$$= \sum_\tau R(\tau)p_\theta(\tau)\nabla log p_\theta(\tau)$$

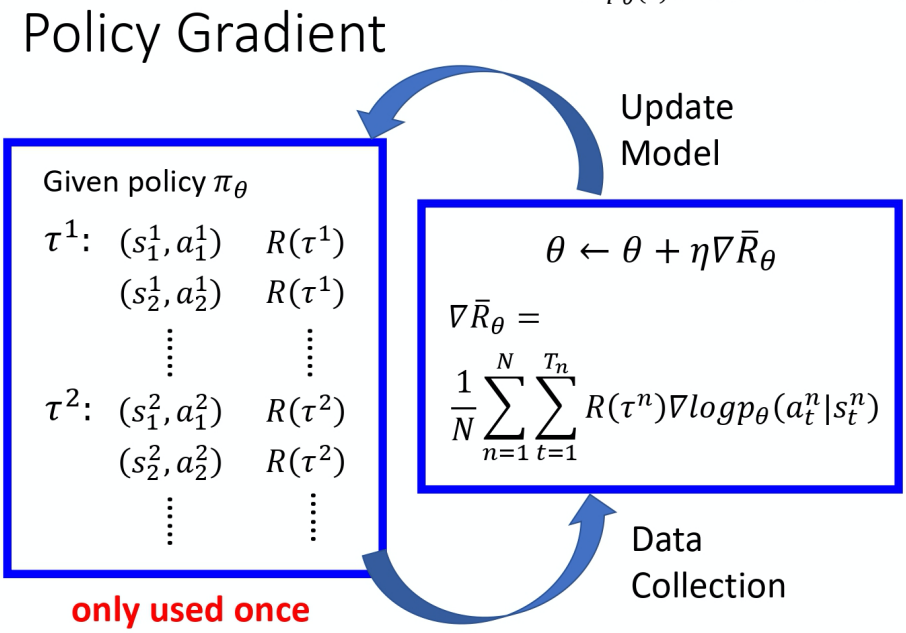
$$= E_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla log p_\theta(\tau)] \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla log p_\theta(\tau^n)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla log p_\theta(a_t^n|s_t^n)$$

我们可以直观地来理解上面这个式子，也就是在你采样到的数据里面， 你采样到在某一个状态s_t要执行某一个动作 a_t， 这个s_t跟 a_t它是在整个轨迹τ的里面的某一个状态和动作的对。

- 假设你在 s_t执行a_t，最后发现 τ的奖励是正的， 那你就要增加这一项的概率，你就要增加在s_t执行a_t的概率。
- 反之，在s_t 执行 a_t会导致 τ的奖励变成负的，你就要减少这一项的概率。

$$\nabla\bar{R}_\theta = E_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla log p_\theta(\tau)]$$



以上这张图很好的表达了，我们如何去实现这个流程。

玩游戏的时候是有随机性的，所以 agent 本身是有随机性的，在同样状态s_1，不是每次都会采取a_1，所以你要记录下来。在状态s_1^1采取a_1^1，在状态s_2^1采取a_2^1。整场游戏结束以后，得到的分数是 $R(\tau^1)$。你会采样到另外一笔数据，也就是另外一场游戏。在另外一场游戏里面，你在状态s_1^2 采取a_1^2，在状态s_2^2采取 a_2^2，然后你采样到的就是 τ^2，得到的奖励是 $R(\tau^2)$。

你就可以把采样到的东西代到这个梯度的式子里面，把梯度算出来。也就是把这边的每一个 s 跟 a 的对拿进来，算一下它的对数概率(log probability)。你计算一下在某一个状态采取某一个动作的对数概率，然后对它取梯度，然后这个梯度前面会乘一个权重，权重就是这场游戏的奖励。 有了这些以后，你就会去更新你的模型。

更新完你的模型以后。你要重新去收集数据，再更新模型。注意，一般 `policy gradient(PG)` 采样的数据就只会用一次。你把这些数据采样起来，然后拿去更新参数，这些数据就丢掉了。在等一下的代码中的话，其实是做的是batch_size 的操作，但是数据也是仅仅只用了一次，但是有好多个batch_size,又由于环境的简单性，所以这里的话就没有过于体现出采集信息上面的麻烦。

$$\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} log p_\theta(a_t^n|s_t^n) \xrightarrow[\text{TF, pyTorch ...}]{} \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} \nabla log p_\theta(a_t^n|s_t^n)$$

$$\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n) log p_\theta(a_t^n|s_t^n) \Rightarrow \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n) \nabla log p_\theta(a_t^n|s_t^n)$$

我们可以把它想成一个分类的问题，在分类里面就是输入一个图像，然后输出决定说是 10 个类里面的哪一个。在做分类时，我们要收集一堆训练数据，要有输入跟输出的对。

在实现的时候，你就把状态当作是分类器的输入。 你就当在做图像分类的问题，只是现在的类不是说图像里面有什么东西，而是说看到这张图像我们要采取什么样的行为，每一个行为就是一个类。比如说第一个类叫做向左，第二个类叫做向右，第三个类叫做开火。

在做分类的问题时，要有输入和正确的输出，要有训练数据。而这些训练数据是从采样的过程来的。假设在采样的过程里面，在某一个状态，你采样到你要采取动作 a， 你就把这个动作 a 当作是你的 ground truth。你在这个状态，你采样到要向左。 本来向左这件事概率不一定是最高， 因为你是采样，它不一定概率最高。假设你采样到向左，在训练的时候，你告诉机器说，调整网络的参数， 如果看到这个状态，你就向左。在一般的分类问题里面，其实你在实现分类的时候，你的目标函数都会写成最小化交叉熵(cross entropy)，其实最小化交叉熵就是最大化对数似然(log likelihood)。

缺陷：因为大多数的时候我们的奖励是正的，负的一般性不会有，但是我们如何让整个机器去学习到最大的那个，其实我们代码中的思路是将其标准化，也就是说减去一个基准，再除以一个方差，这个的目的就是重新给回报一个量纲，让其有更好的学习能力的意思。等一下在代码中会有体现，当然还有别的方法。

代码实现

model.py

```
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    '''
    多层感知机
    输入：state维度
    输出：概率
    '''
    def __init__(self,state_dim,hidden_dim=36):
        super(MLP,self).__init__()
        #24和36为隐藏层的层数，可根据state_dim,action_dim的情况来改变
        self.fc1=nn.Linear(state_dim,hidden_dim)
        self.fc2=nn.Linear(hidden_dim,hidden_dim)
        self.fc3=nn.Linear(hidden_dim,1)

    def forward(self,x):
        x=F.relu(self.fc1(x))
        x=F.relu(self.fc2(x))
        x=F.sigmoid(self.fc3(x))
        return x
    '''
    以上是MLP也就是多层感知机的实现，其实是最简单的神经网络，只是用了几层全连接层进行一个训练，其中的激活函数是relu,最后通过sigmoid函数返回一个类概率，也就是概率。接下来就是根据概率去做动作选择，再进行一个采样的过程
    '''
```

agent.py

```
import torch
from torch.distributions import Bernoulli
```

```
from torch.autograd import Variable
import numpy as np
from model import MLP


class PolicyGradient(object):

    def __init__(self,state_dim,cfg):
        self.gamma=cfg.gamma
        self.policy_net=MLP(state_dim,hidden_dim=cfg.hidden_dim)

 self.optimizer=torch.optim.RMSprop(self.policy_net.parameters(),lr=cf
g.lr)
        self.batch_size=cfg.batch_size

    def choose_action(self,state):

        state=torch.from_numpy(state).float()
        state=Variable(state)
        probs=self.policy_net(state)
        m=Bernoulli(probs) #伯努利分布,使用伯努利的原因是在该环境下的取值只有0或
者1
        action=m.sample()
        print(action)
        action=action.data.numpy().astype(int)[0] #转为标量
        return action

    def update(self,reward_pool,state_pool,action_pool):
        #将回报折现
        running_add=0
        for i in reversed(range(len(reward_pool))):
            if reward_pool[i]==0:
                running_add=0
            else:
                running_add=running_add*self.gamma+reward_pool[i]
                reward_pool[i]=running_add

        #标准化回报，这一步的处理就是加上基准的意思，使得整个回报处在有正有负的环境下
        reward_mean=np.mean(reward_pool)
        reward_std=np.std(reward_pool)
        for i in range(len(reward_pool)):
```

```
            reward_pool[i]=(reward_pool[i]-reward_mean)/reward_std

        # 梯度上升
        self.optimizer.zero_grad()

        for i in range(len(reward_pool)):
            state=state_pool[i]
            action=Variable(torch.FloatTensor([action_pool[i]]))
            reward=reward_pool[i]

            state=Variable(torch.from_numpy(state).float())
            probs=self.policy_net(state)
            m=Bernoulli(probs)
            loss=-m.log_prob(action)*reward #Negative score function x
reward
            loss.backward()
        self.optimizer.step()
    def save(self,path):

 torch.save(self.policy_net.state_dict(),path+"pg_checkpoint.pt")
    def load(self,path):

 self.policy_net.load_state_dict(torch.load(path+"pg_checkpoint.pt"))

        '''
        以上便是整个策略梯度的核心所在，具体在录视频中进行讲解
        '''
```

work.py

```
import sys, os
import gym
import torch
import datetime
from itertools import count
from agent import PolicyGradient
from plot import plot_rewards_cn
from utils import save_results,make_dir

curr_path=os.path.dirname(os.path.abspath(__file__))
```

```
curr_time=datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

class PGConfig(object):
    def __init__(self):
        self.algo="PolicyGradient" #项目名字
        self.env="CartPole-v0" #环境名称

 self.result_path=curr_path+"/outputs/"+self.env+"/"+curr_time+"/resul
ts/" #结果保存路径

 self.model_path=curr_path+"/outputs/"+self.env+"/"+curr_time+"/models
/" #模型保存的路径
        self.train_eps=300 #训练的episode数目
        self.eval_eps=50
        self.batch_size=8
        self.lr=0.01 #学习率
        self.gamma=0.99
        self.hidden_dim=36
        self.device=torch.device("cuda")


def env_agent_config(cfg,seed):
    env=gym.make(cfg.env)
    env.seed(seed)
    state_dim=env.observation_space.shape[0]
    agent=PolicyGradient(state_dim,cfg)
    return env,agent

def train(cfg,env,agent):
    print("测试开始！")
    print(f"环境：{cfg.env},算法：{cfg.algo},设备：{cfg.device}")
    state_pool=[] #存放每batch_size个episode的state序列
    action_pool=[]#存放每batch_size个episode的action序列
    reward_pool=[]#存放每batch_size个episode的reward序列
    rewards=[]
    ma_rewards=[]
    for i_episode in range(cfg.train_eps):
        state=env.reset()
        ep_reward=0
        for _ in count():#作用等同于while True
```

```
            action=agent.choose_action(state) #根据当前环境state选择
action
            next_state,reward,done,_=env.step(action)
            ep_reward+=reward
            if done:
                reward=0
            state_pool.append(state)
            action_pool.append(float(action))
            reward_pool.append(reward)
            state=next_state

            if done:
                print("回合:",i_episode,"回报：",ep_reward)
                break
        print(len(action_pool))
        if i_episode>0 and i_episode % cfg.batch_size==0:
            agent.update(reward_pool,state_pool,action_pool)
            state_pool=[] #每个回合的state
            action_pool=[] #每个回合的action
            reward_pool=[] #每个回合的reward
        print(len(action_pool))
        rewards.append(ep_reward)
        if ma_rewards:
            ma_rewards.append(0.9*ma_rewards[-1]+0.1*ep_reward)
        else:
            ma_rewards.append(ep_reward)
    print("训练完毕！")
    return rewards,ma_rewards


def eval(cfg,env,agent):
    print("测试开始！")
    print(f"环境：{cfg.env},算法：{cfg.algo},设备：{cfg.device}")
    rewards=[]
    ma_rewards=[]
    for i_episode in range(cfg.eval_eps):
        state=env.reset()
        ep_reward=0
        for _ in count():
            env.render()
            action=agent.choose_action(state)
```

```
            next_state,reward,done,_=env.step(action)
            ep_reward+=reward
            if done:
                reward=0
            state=next_state
            if done:
                print("回合:",i_episode,"回报：",ep_reward)
                break
        rewards.append(ep_reward)
        if ma_rewards:
            ma_rewards.append(0.9*ma_rewards[-1]+0.1*ep_reward)
        else:
            ma_rewards.append(ep_reward)
        env.close()
    print("测试完毕！")
    return rewards,ma_rewards



cfg=PGConfig()

#训练
env,agent=env_agent_config(cfg,seed=10)
rewards,ma_rewards=train(cfg,env,agent)
make_dir(cfg.result_path,cfg.model_path)
agent.save(path=cfg.model_path)
save_results(rewards,ma_rewards,tag="train",path=cfg.result_path)
plot_rewards_cn(rewards,ma_rewards,tag="训
练",algo=cfg.algo,path=cfg.result_path)

#测试
env,agent=env_agent_config(cfg,seed=10)
agent.load(path=cfg.model_path)
rewards,ma_rewards=eval(cfg,env,agent)
save_results(rewards,ma_rewards,tag="eval",path=cfg.result_path)
plot_rewards_cn(rewards,ma_rewards,tag="测
试",env=cfg.env,algo=cfg.algo,path=cfg.result_path)
```

最后附上github上的完整代码，点击github即可