# 强化学习实验报告八——Pendulum的深度确定性策略梯度算法(DDPG)

**学号:2013365　姓名:颜铭**

## 1　实验导言

从上次实验的代码分析中，我们确定了滑块倒立摆模型(cartpole)关于动作激励的状态响应是离散的(角度和角速度在计算时采用了差分算法和区间近似)，并重写了关于数组特征形状的分布采样函数，使其适应离散的状态-动作形式。充分考虑值函数的梯度解析学习算法特性，我们也可以将离散的策略梯度算法推广到连续问题上。

pendulum环境就是角度角速度在有限值域上连续且奖励函数为关于角度，角速度以及结合力矩模型确定的待定动作值的解析函数。一般来说。我们要研究误差模型最为常用的连续分布为高斯分布。我们考虑嵌入高斯分布随机采样连续动作进行模式迁移，同时依据连续状态$(\theta,\dot{\theta})$使用全连接层的权值计算实现特征拟合以得到参数分析，最后加入到对数策略梯度损失中进行连续策略梯度学习。

具体地，pendulum环境中的状态数有三个，摆针在竖直方向上的顺时针旋转角为$\theta$,$\theta$的取值范围在$[-\pi,\pi]$之间，而相应的状态为$[\cos\theta,\sin\theta,\dot{\theta}]$之间。这三个物理量表示的是角度和角速度以及旋转参数。对于动作$a$取值，我们确定为一个取值为$a\backsim[-2,2]$之间的力矩，其是一个连续量，这样就导致了策略应该是奖励的解析函数，不能用于离散算法如深度Q学习网络解决。pendulum环境的奖励确定为:

$$r=-(\theta^2+0.1\times\hat{\theta}^2+0.001\times a^2) \tag{1}$$

一般来说每回合最大的步数为200步，其每一步的最低奖励为-16.27最高奖励为0

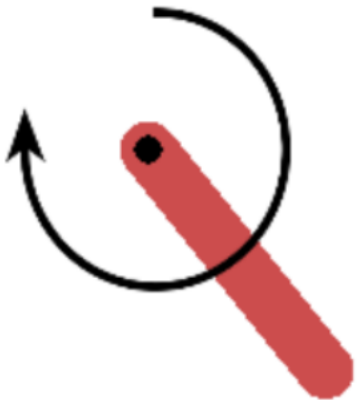


图1.Pendulum倒立摆模型渲染图

此外，我们可以借鉴`tensorflowv1`的代码实现。使用`pytorch`框架代码进行相关复现

## 2　实验分析

回顾基于分幕式序列的蒙特卡洛控制视角下策略梯度算法的伪代码,我们在此基础上通过结合代表深度学习思想的线性网络层进行基于梯度损失函数的训练，再结合演员评论员算法的控制学习，即可得到有限可数采样状态下的概率参数优化组合成的**不确定性策略梯度算法**

**REINFORCE：$\pi_\theta$ 的蒙特卡洛策略梯度的控制算法 (分幕式)**

输入：一个可导的参数化策略 $\pi(a|s,\boldsymbol{\theta})$

算法参数：步长 $\alpha > 0$

初始化策略参数 $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (如初始化为 $\mathbf{0}$)

无限循环 (对于每一幕)：

根据 $\pi(\cdot|\cdot,\boldsymbol{\theta})$，生成一幕序列 $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

对于幕的每一步循环，$t = 0, 1, \ldots, T-1$：

$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

我们进一步给出对于连续动作的深度确定性策略梯度算法，也即学术界主流研究成果之一的DDPG算法:

|初始化评论员网络(policy_train)$Q(s,a|\theta_Q)$和演员网络(policy_test)$\mu(s|\theta_\mu)$,其权重分别为$\theta_Q$和$\theta_u$

初始化目标$Q'$和$\mu'$，并复制权重为$\theta_{Q'} \leftarrow \theta_Q$,$\theta_{\mu'} \leftarrow \theta_\mu$

初始化回放缓冲区(replay)D(Sample可选)

执行M个回合循环，对每个回合

初始化动作探索的随机过程(多为高斯过程)，即噪声$\mathcal{N}$

初始化状态$s_t$

循环$T$个时间步长，对于每个时间步长$t$

根据当前的策略和噪声选择动作$a_t = \mu(s_t|\theta_t)$

环境根据$a_t$反馈奖励$r_t$和下一个状态$s_{t+1}$

存储经验即$(s_t, a_t, r_t, s_{t+1})$到Sample中

更新策略执行:

从Sample中随机采样$N$个小批量的转移

计算实际的Q值$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta_\mu)|\theta_{Q'})$

对损失函数$L$(MSELoss or Entropy)进行关于参数$\theta$的随机

梯度下降更新评论员网络

$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s,a|\theta_Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta_\mu)|_{s_i}$

软更新目标网络(重点)

$\theta^{Q'} \leftarrow \tau\theta_Q + (1-\tau)\theta_{Q'}, \theta_{\mu'} \leftarrow \tau\theta_\mu + (1-\tau)\theta_{\mu'}$

针对深度确定性策略梯度连续和离散的策略梯度算法在深度学习网络实现上的区别在于策略概率的表示的具体形式$\pi(A_t|S_t,\theta)$在动作奖励函数中的引入保证可微操作的重参数化以及连续动作网络使用更普遍的高斯误差模型。而基于$\gamma$的离轨策略的蒙特卡洛周期采样奖励和形式保持不变。此外，在值约束下的动作状态价值在一定精度范围内有多个连续值，这也就要求我们基于连续的概率分布计算权重修正的误差不确定性(使用交叉熵损失函数是合适的)

具体解释重参数化，高斯模型也即高斯分布(正态分布模型)，我们知道其解析形式为:

$$N(\mu,\sigma)=\frac{1}{2\pi\sqrt{\sigma}}e^{-\frac{(x-\mu)^2}{2\sigma}} \tag{2}$$

因此我们在一个幕(episode)中采样到的多个连续动作价值需要确定一个共性的动作特征，也即通过随机连续样本的线性观测拟合输出连续概率空间可达的估计(相较连续可以视为一种试探性开发的推广)意义的隐性表达参数对$(\mu,\sigma)$,隐函数的重合来自于值域变换都和同一个线性权重和偏置的全连接输出层相关。也即一层隐含层为固定下采样特征(值为200)感受野的两层全连接层(输出大小为1)。对于$\mu$和$\sigma$的取值特征，我们需要通过激活函数进行取值的变换:

- 对于平均值$\mu$：使用tanh激活函数使得正向激励的动作价值对称分布，均值取值为$[-1\backsim 1]$

$$tanh(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}} \tag{3}$$

- 对于方差$\sigma$：使用softplus激活函数使得方差恒为正，而且有较好的线性逼近特性和终端收敛特性(为负无穷逼近零，为正无穷则消去$\ln(1)$影响接近输出无穷大)

$$softplus(x)=\ln(1+e^x) \tag{4}$$

另外使用损失函数形式为$\text{loss}=G\nabla\pi(a_t,s_t)+0.01\,\text{Entropy}(x)$，这里交叉熵损失函数是连续分布的交叉熵模型:

$$\text{Entropy}(p,q)=-\sum_{i=1}^{n}p_i\log q_i \tag{5}$$

这里$p$ 是真实的概率分布，$q$ 是模型预测的概率分布，$n$ 是类别数

## 3 代码实现

由以上分析我们可以直接得到采样类`Sample`的代码实现与离散策略梯度相同，注意区分`observation_`维度的变换。

```
class Sample():
    def __init__(self,env,policyNet):
        self.env=env
        self.policy_net = policyNet
        self.gamma=0.95

    def sample_episodes(self,num_episode):
        batch_obs = []
        batch_actions =[]
        batch_rwd =[]
        for i in range(num_episode):
            observation = self.env.reset()

            reward_episode = []

            while True:
                state = np.reshape(observation,[1,3])
                action = self.policy_net.choose_action(state)
                # if device == torch.device('cuda'):
                #     action = action.cpu().numpy()
```

```
                # else :
                #     action = action.numpy()
                observation_ , reward,done,_ = self.env.step(action)


                batch_obs.append(np.reshape(observation,[1,3])[0,:])
                batch_actions.append(action)
                reward_episode.append((reward+8)/8)

                if done:
                    reward_sum =0
                    discounted_sum_reward = np.zeros_like(reward_episode)
                    for t in reversed(range(len(reward_episode))):
                        reward_sum = reward_sum*self.gamma + reward_episode[t]
                        discounted_sum_reward[t] = reward_sum
                    for t in range(len(reward_episode)) :
                        batch_rwd.append(discounted_sum_reward[t])

                    break

                observation = observation_

        batch_obs=np.reshape(batch_obs,[len(batch_obs),self.policy_net.n_features])
        batch_actions = np.reshape(batch_actions,[len(batch_actions),1])
        batch_rwd = np.reshape(batch_rwd , [len(batch_rwd),1])
        return batch_obs,batch_actions,batch_rwd
```

同上次分析，对应算法实现公式为:

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{6}$$

此外，我们定义高斯模型的全连接层网络，由上节理论分析有:

```
class PolicyNet(torch.nn.Module):
    def __init__(self,env):
        super(PolicyNet,self).__init__()
        self.n_features = env.observation_space.shape[0]
        self.n_actions =1
        #self.action_bound =action_bound
        #self.obs = torch.tensor(0,dtype=torch.float32).expand(None,self.n_features)
        self.dist_normal = None


        self.f1 = torch.nn.Linear(in_features=self.n_features,out_features=200)
        torch.nn.init.normal_(self.f1.weight,mean=0,std=0.1)
        torch.nn.init.constant_(self.f1.bias,0.1)
        self.act1 = torch.nn.ReLU()
```

```
        self.f2 = torch.nn.Linear(200,self.n_actions)
        self.act2= torch.nn.Tanh()## tanh激活函数
        torch.nn.init.normal_(self.f2.weight,mean=0,std=0.1)
        torch.nn.init.constant_(self.f2.bias,0.1)
        self.act3 = torch.nn.Softplus()## softplus激活函数

    def forward(self,x):
        x=self.f1(x)
        x=self.act1(x)
        #new_opt =x.clone()
        x=self.f2(x)
        mu=self.act2(x)
        sigma=self.act3(x)
        return mu, sigma  ##预测均值和方差

```

实现连续策略动作网络中，要注意比对采样类中对于连续状态的预估状态价值是基于高斯模型且有一定值域约束的，所以我们使用`clamp函数裁剪`。此外在已有的动作搜寻范围内计算价值特征。

```
   def choose_action(self,state):
        #state.astype(dtype=np.float32)
        state = torch.from_numpy(state).type(torch.FloatTensor)
        state=torch.squeeze(state,dim=0)
        mu,sigma = self.actor(state)
        self.dist_normal=torch.distributions.Normal(2*mu,sigma)
        action =
torch.clamp(self.dist_normal.sample(),self.action_bound[0],self.action_bound[1])
        return action
```

定义网络在每个幕中更新使用梯度反向传播计算的规则函数:

```
  def update(self,obs_batch,action_batch,reward_batch):
        obs_batch = torch.from_numpy(obs_batch).type(torch.FloatTensor)
        action_batch=torch.from_numpy(action_batch).type(torch.FloatTensor)
        reward_batch = torch.from_numpy(reward_batch).type(torch.FloatTensor)
        mu,sigma = self.actor(obs_batch)
        self.dist_normal =torch.distributions.Normal(2*mu,sigma)
        log_prob = self.dist_normal.log_prob(action_batch)
        self.loss = - (log_prob*reward_batch+0.01*self.dist_normal.entropy()).mean()
        self.optimizer.zero_grad()
        self.loss.backward()
        self.optimizer.step()
```

优化器采用`Adam`，学习率设定为0.0001

对于策略梯度的训练和演员评估(actor)测试，我们灵活使用与离散策略梯度算法训练cartpole环境基本一致的奖励曲线演算逻辑。

```python
def policy_test(env,policy,render_mode,test_iter):

    for i in range(test_iter):
        observation = env.reset()
        reward_sum = 0
        while True:
            if render_mode:
                env.render()
            state = np.reshape(observation,[1,3])
            action = policy.choose_action(state)
            observation_,reward,done,info = env.step(action)
            reward_sum+=reward
            if done:
                break
            observation = observation_

    return reward_sum
def policy_train(env,agent:PolicyConstant_Gradient,training_iter,model_saved=None):
    reward_sum =0
    reward_sum_line =[]
    training_time=[]

    agent=agent

    env=env
    tmp =0
    for i in range(training_iter):
        tmp=i
        sampler=Sample(env,agent)
        train_obs,train_action,train_reward = sampler.sample_episodes(1)
        agent.update(train_obs,train_action,train_reward)

        if i==0:
            reward_sum = policy_test(env,agent,render_mode,1)
        else:
            reward_sum = 0.95*reward_sum + 0.05*policy_test(env,agent,render_mode,1)
        reward_sum_line.append(reward_sum)
        training_time.append(i)

        print("training episodes is {},trained reward sum is {}".format(i,reward_sum))

        if reward_sum>-500:
            break
    if model_saved:
        torch.save(agent,model_saved)
    else:
        torch.save(agent,'best_pg_pendulum_{}'.format(tmp+1))
```

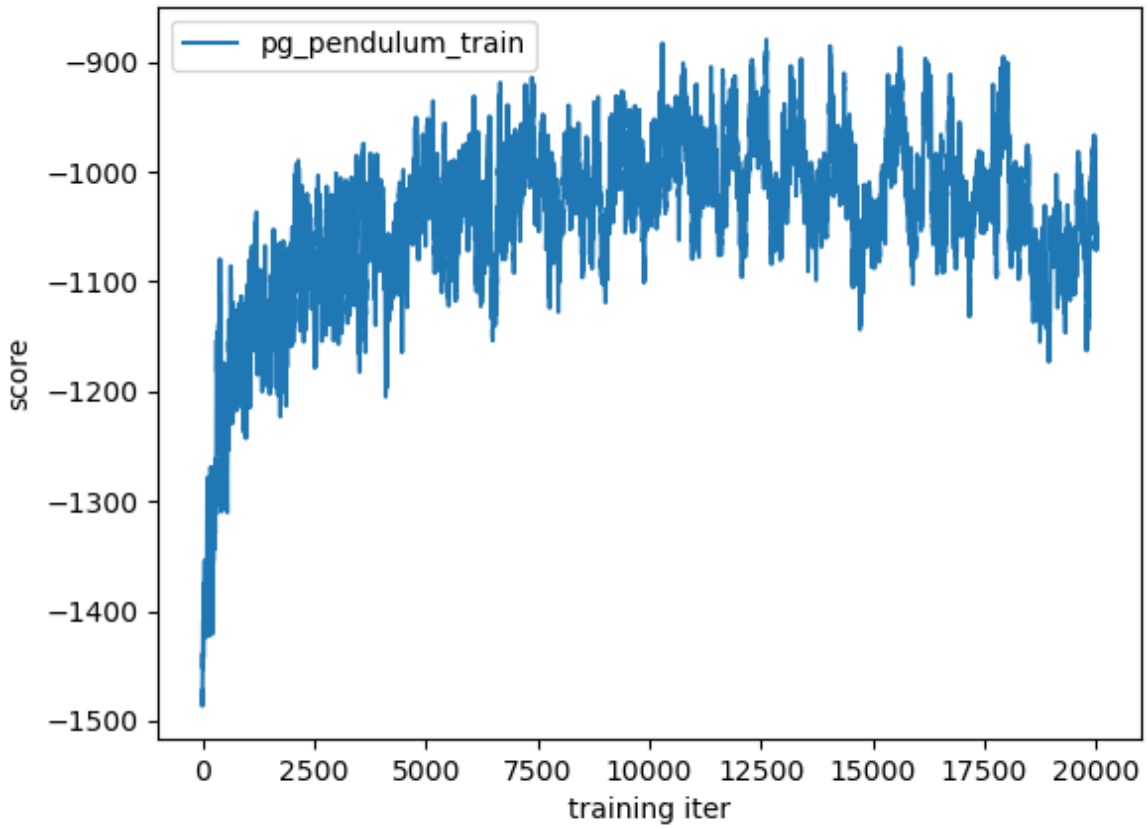
```
48
49      plt.plot(training_time,reward_sum_line,label="pg_pendulum_train")
50
51      plt.xlabel("training iter")
52      plt.ylabel("score")
53
54      plt.legend()
55      plt.show()
56
```

最后是主函数，设定分幕为20000幕，并注意初始化连续动作值的区间约束

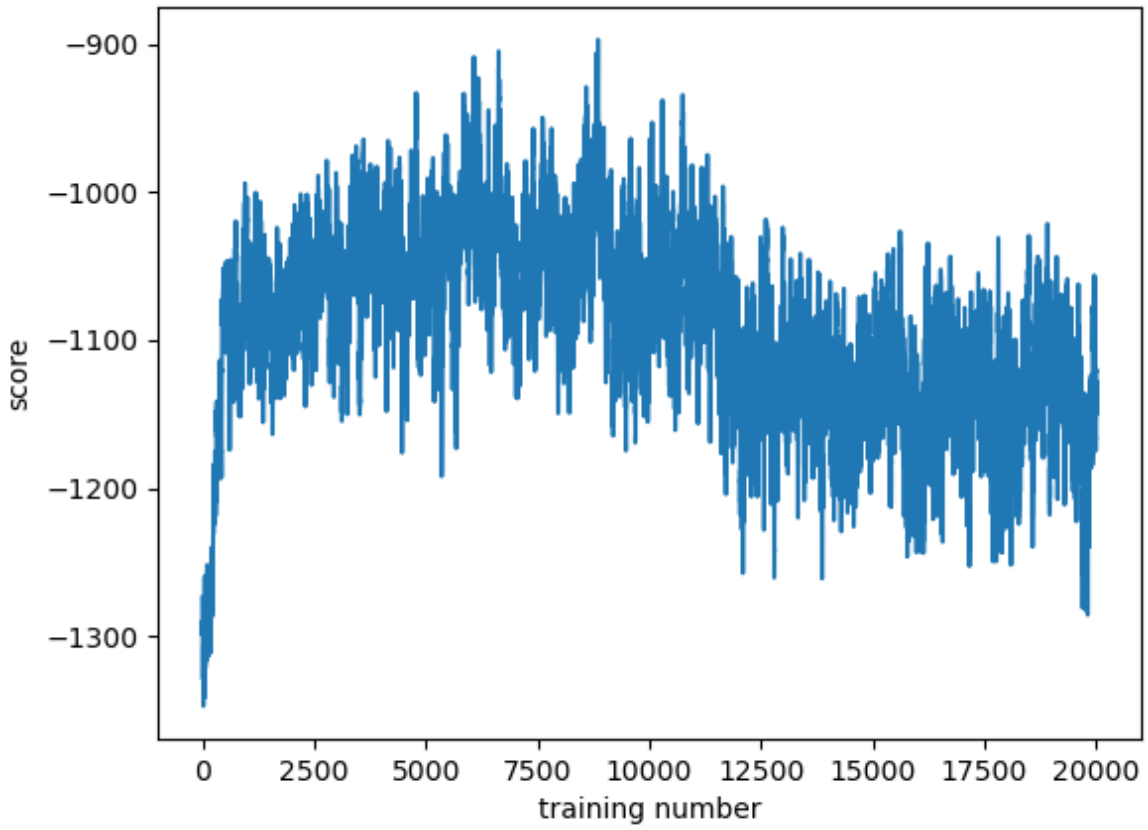
```
1   if __name__ =="__main__":
2
3       env_name = "Pendulum-v0"
4       env = gym.make(env_name)
5
6       env.unwrapped
7
8       env.seed(1)
9
10      action_bound = np.array([-env.action_space.high,env.action_space.high]).reshape(2)
11      policy_actor =PolicyNet(env)
12
13      Agent = PolicyConstant_Gradient(policy_actor,action_bound)
14
15
16      training_iter = 20000
17
18      policy_train(env,Agent,training_iter)
19
20      reward_sum = policy_test(env,Agent,True,10)
21
22      print("The Model tested Reward Sum is {}".format(reward_sum))
```

## 4 实验结果

我们通过20000次迭代，在实验一开始大概1000个episode以内有较快和容许性好(局部最优解现象存在但不主导)的梯度上升效果，其训练曲线如下:




之后即为平稳振荡，没有达到200个回合的综合奖励收敛的结束预判，为验证代码实现的准确性，也对给出的`tensorflowv1`代码进行训练，有如下奖励曲线:




可见pytorch动态图相对tf1分离占位符初始化的操作确实作为超参数的一部分提升了训练效果(网络集成的静态图资源的重参数特性的baseline效果降低是可以预见的，因为越一致的动作采样到预测的重参数影响了伪随机机制的激励)