策略梯度算法

简介

之前我们介绍的Q-learning和DQN及改进算法都是基于值函数（value-based）的方法，其中Q-learning是处理有限状态的算法，而DQN可以用来解决连续状态的问题。在强化学习中，除了基于值函数的方法，还有一支非常经典的方法，那就是基于策略（policy-based）的方法。对比两者，基于值函数的方法主要是学习值函数，然后根据值函数导出一个策略，此时并不存在一个显式的策略。而基于策略的方法则是直接显式的学习一个目标策略。策略梯度是基于策略的方法的基础，我们将从策略梯度算法说起。

策略梯度

基于策略的方法首先需要参数化策略，我们假设目标策略π_θ是一个随机性策略，并且处处可微，其中θ是对应的参数。我们可以用一个线性模型或者神经网络模型来建模这样一个策略函数，输入是某个状态，然后输出一个动作的概率分布。我们的目标是要寻找一个最优策略，来最大化这个策略在环境中的期望回报。我们将策略学习的目标函数定义为

$$J(\theta) = \mathbb{E}_{s_0}[V^{\pi_\theta}(s_0)]$$

其中s_0表示初始状态。现在有了目标函数，我们将目标函数对策略θ求导，得到导数后，我们就可以用梯度上升方法来最大化这个目标函数从而得到最优策略。

我们之前在MDP章节中学习过在策略π下的状态访问分布，我们在此用ν^π表示。然后我们对目标函数求梯度，可以得到如下式子，更详细的推导将在扩展阅读中给出。

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \sum_{s\in S}\nu^{\pi_\theta}(s)\sum_{a\in A}Q^{\pi_\theta}(s,a)\nabla_\theta\pi_\theta(a|s) &(1)\\ &= \sum_{s\in S}\nu^{\pi_\theta}(s)\sum_{a\in A}\pi_\theta(a|s)Q^{\pi_\theta}(s,a)\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_\theta(a|s)} &(2)\\ &= \mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s,a)\nabla_\theta\log\pi_\theta(a|s)] &(3)\end{aligned}$$

于是，我们就可以用这个梯度来更新策略。需要注意的是，因为上式期望的下标是π_θ，所以策略梯度算法为在线学习算法，即必须使用策略π_θ采样得到的数据来计算梯度。更一般地，我们可以把梯度写成下面这个形式：

$$g = \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{\infty}\psi_t\nabla_\theta\log\pi_\theta(a_t|s_t)]$$

其中ψ_t可以有很多种形式：

$$\begin{aligned}&1.\sum_{t'=0}^{\infty}\gamma^t r_{t'}：\text{轨迹的总回报} && 4.Q^{\pi_\theta}(s_t,a_t)：\text{动作价值函数} &(4)\\ &2.\sum_{t'=t}^{\infty}\gamma^{t'-t}r_{t'}：\text{动作}a_t\text{之后的回报} && 5.A^{\pi_\theta}(s_t,a_t)：\text{优势函数} &(5)\\ &3.\sum_{t'=t}^{\infty}r_{t'}-b(s_t)：\text{基线版本的改进} && 6.r_t+V^{\pi_\theta}(s_{t+1})-V^{\pi_\theta}(s_t)：\text{时序差分残差} &(6)\end{aligned}$$

至此，我们已经了解了策略梯度的目标与形式，那么我们今天要讲的REINFORCE算法的流程是怎样的呢？

REINFORCE 算法

其实REINFORCE就是在上文中当$\psi=\sum_{t'=t}^{\infty}\gamma^{t'-t}r_{t'}$时的策略梯度算法。他的具体算法流程如下：

- 初始化策略参数θ
- 不断进行如下循环（每个循环是一条序列）：
  - 用当前策略π_θ采样轨迹$\{s_1,a_1,r_1,s_2,a_2,r_2\dots\}$
  - 计算当前轨迹每个时刻往后的回报$\sum_{t'=t}^{\infty}\gamma^{t'-t}r_{t'}$作为$\psi_t$
  - 对θ进行更新 $\theta=\theta+\alpha\sum_t\psi_t\nabla_\theta\log\pi_\theta(a_t|s_t)$

这便是REINFORCE算法的全部了，让我们来用代码实现它看看效果如何吧！

REINFORCE代码实践

我们在Cartpole环境中进行REINFORCE算法的实验。

In [1]:

```
import gym
import torch
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import rl_utils
```

定义我们的策略网络PolicyNet，输入是状态，输出则是该状态下的动作分布。

In [2]:

```
class PolicyNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(PolicyNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return  F.softmax(self.fc2(x),dim=1)
```

再定义我们的Reinforce算法。在函数take_action中，我们通过概率对离散的动作进行采样。在更新过程中，我们按照算法，将损失函数写为$-\sum_t\psi_t\nabla_\theta\log\pi_\theta(a_t|s_t)$，对$\theta$求导后就可以通过梯度下降更新策略。

In [3]:

```
class Reinforce:
    def __init__(self, state_dim, hidden_dim, action_dim, learning_rate, gamma, device):
        self.policy_net = PolicyNet(state_dim, hidden_dim, action_dim).to(device)
        self.optimizer= torch.optim.Adam(self.policy_net.parameters(), lr=learning_rate) # 使用Adam优化器
        self.gamma = gamma # 折扣因子

    def take_action(self, state): # 根据动作概率分布随机采样
        state = torch.tensor([state], dtype=torch.float)
        probs = self.policy_net(state)
        action_dist = torch.distributions.Categorical(probs)
        action = action_dist.sample()
        return action.item()

    def update(self, transition_dict):
        reward_list = transition_dict['rewards']
        state_list = transition_dict['states']
        action_list = transition_dict['actions']

        G = 0
        self.optimizer.zero_grad()
        for i in reversed(range(len(reward_list))): # 从最后一步算起
            reward = reward_list[i]
            state = torch.tensor([state_list[i]], dtype=torch.float)
            action = torch.tensor([action_list[i]]).view(-1, 1)
            log_prob = torch.log(self.policy_net(state).gather(1, action))
            G = self.gamma * G + reward
            loss = - log_prob * G # 每一步的损失函数
            loss.backward() # 反向传播计算梯度
        self.optimizer.step() # 梯度下降
```

定义好策略，我们就可以开始实验了，看看Reinforce算法在Cartpole环境上表现如何吧！

In [4]:

```
learning_rate = 1e-3
num_episodes = 1000
hidden_dim = 128
gamma = 0.98
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

env_name = "CartPole-v0"
env = gym.make(env_name)
env.seed(0)
torch.manual_seed(0)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = Reinforce(state_dim, hidden_dim, action_dim, learning_rate, gamma, device)

return_list = []
for i in range(10):
    with tqdm(total=int(num_episodes/10), desc='Iteration %d' % i) as pbar:
        for i_episode in range(int(num_episodes/10)):
            episode_return = 0
            transition_dict = {'states': [], 'actions': [], 'next_states': [], 'rewards': [], 'dones': []}
            state = env.reset()
            done = False
            while not done:
                action = agent.take_action(state)
                next_state, reward, done, _ = env.step(action)
                transition_dict['states'].append(state)
                transition_dict['actions'].append(action)
                transition_dict['next_states'].append(next_state)
                transition_dict['rewards'].append(reward)
                transition_dict['dones'].append(done)
                state = next_state
                episode_return += reward
            return_list.append(episode_return)
            agent.update(transition_dict)
            if (i_episode+1) % 10 == 0:
                pbar.set_postfix({'episode': '%d' % (num_episodes/10 * i + i_episode+1), 'return': '%.3f' % np.mean(return_list[-10:])})
            pbar.update(1)
```

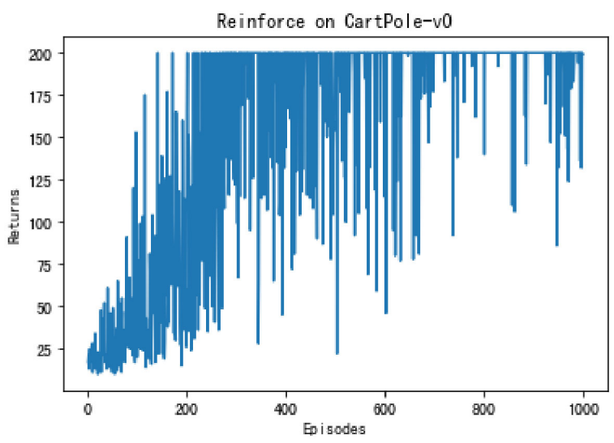
```
/opt/conda/lib/python3.7/site-packages/torch/cuda/__init__.py:52: UserWarning: CUDA initialization: Found no NVIDIA driver on your system.
  return torch._C._cuda_getDeviceCount() > 0
Iteration 0: 100%|██████████| 100/100 [00:06<00:00, 16.52it/s, episode=100, return=55.500]
Iteration 1: 100%|██████████| 100/100 [00:11<00:00,  8.62it/s, episode=200, return=75.300]
Iteration 2: 100%|██████████| 100/100 [00:22<00:00,  4.52it/s, episode=300, return=178.800]
Iteration 3: 100%|██████████| 100/100 [00:28<00:00,  3.53it/s, episode=400, return=164.600]
Iteration 4: 100%|██████████| 100/100 [00:28<00:00,  3.49it/s, episode=500, return=156.500]
Iteration 5: 100%|██████████| 100/100 [00:30<00:00,  3.30it/s, episode=600, return=187.400]
Iteration 6: 100%|██████████| 100/100 [00:30<00:00,  3.30it/s, episode=700, return=194.500]
Iteration 7: 100%|██████████| 100/100 [00:31<00:00,  3.13it/s, episode=800, return=200.000]
Iteration 8: 100%|██████████| 100/100 [00:32<00:00,  3.12it/s, episode=900, return=200.000]
Iteration 9: 100%|██████████| 100/100 [00:31<00:00,  3.15it/s, episode=1000, return=186.100]
```

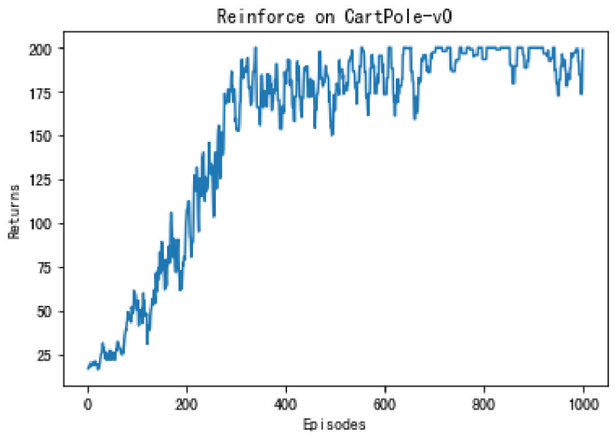
在CartPole-v0环境中，满分就是200分，我们发现Reinforce算法效果很好，可以达到200分。接下来我们画一下训练过程中每一条轨迹的回报变化图。由于抖动比较大，我们又进行了光滑处理。

In [5]:

```
episodes_list = list(range(len(return_list)))
plt.plot(episodes_list,return_list)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('Reinforce on {}'.format(env_name))
plt.show()

mv_return = rl_utils.moving_average(return_list, 9)
plt.plot(episodes_list, mv_return)
plt.xlabel('Episodes')
plt.ylabel('Returns')
plt.title('Reinforce on {}'.format(env_name))
plt.show()
```





总结

REINFORCE算法理论上是能保证局部最优的。REINFORCE算法实际上是借助于蒙特卡洛方法采样轨迹估计动作价值，这种做法的一个优点是梯度是无偏的。但是同样由于因为是蒙特卡洛方法，导致REINFORCE算法梯度估计的方差很大，从而可能会造成一定程度上的不稳定，这也是接下来将介绍的Actor-Critic算法要解决的问题。

拓展阅读：策略梯度证明

我们要证明$\nabla_\theta J(\theta) \propto \sum_{s\in S} \nu^\pi(s) \sum_{a\in A} Q^{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s)$

先从状态价值函数的推导开始：

$$\begin{aligned}
\nabla_\theta V^{\pi_\theta}(s) &= \nabla_\theta (\sum_{a\in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a)) && (7)\\
&= \sum_{a\in A} (\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \nabla_\theta Q^{\pi_\theta}(s,a)) && (8)\\
&= \sum_{a\in A} (\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \nabla_\theta \sum_{s',r} p(s',r|s,a)(r+V^{\pi_\theta}(s'))) && (9)\\
&= \sum_{a\in A} (\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \sum_{s',r} p(s',r|s,a) \nabla_\theta V^{\pi_\theta}(s')) && (10)\\
&= \sum_{a\in A} (\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \sum_{s'} p(s'|s,a) \nabla_\theta V^{\pi_\theta}(s')) && (11)
\end{aligned}$$

为了简化表示，我们让$\phi(s) = \sum_{a\in A} \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$, 定义$d^{\pi_\theta}(s \to x, k)$为策略$\pi$从状态$s$出发$k$步后到达状态$x$的概率，我们继续推导:

$$\begin{aligned}
\nabla_\theta V^{\pi_\theta}(s) &= \phi(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^{\pi_\theta}(s') && (12)\\
&= \phi(s) + \sum_a \sum_{s'} \pi_\theta(a|s) P(s'|s,a) \nabla_\theta V^{\pi_\theta}(s') && (13)\\
&= \phi(s) + \sum_{s'} d^{\pi_\theta}(s \to s', 1) \nabla_\theta V^{\pi_\theta}(s') && (14)\\
&= \phi(s) + \sum_{s'} d^{\pi_\theta}(s \to s', 1) [\phi(s') + \sum_{s''} d^{\pi_\theta}(s' \to s'', 2) \nabla_\theta V^{\pi_\theta}(s'')] && (15)\\
&= \phi(s) + \sum_{s'} d^{\pi_\theta}(s \to s', 1) \phi(s') + \sum_{s''} d^{\pi_\theta}(s \to s'', 2) \nabla_\theta V^{\pi_\theta}(s'') && (16)\\
&= \phi(s) + \sum_{s'} d^{\pi_\theta}(s \to s', 1) \phi(s') + \sum_{s''} d^{\pi_\theta}(s' \to s'', 2) \phi(s'') + \sum_{s'''} d^{\pi_\theta}(s \to s''', 3) \nabla_\theta V^{\pi_\theta}(s''') && (17)\\
&= \ldots\ldots && (18)\\
&= \sum_{x\in S} \sum_{k=0}^{\infty} d^{\pi_\theta}(s \to x, k) \phi(x) && (19)
\end{aligned}$$

我们定义$\eta(s) = \mathbb{E}_{s_0}[\sum_{k=0}^{\infty} d^{\pi_\theta}(s_0 \to s, k)]$。至此我们回到我们的目标函数：

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{s_0}[V^{\pi_\theta}(s_0)] && (20)\\
&= \sum_s \mathbb{E}_{s_0}[\sum_{k=0}^{\infty} d^{\pi_\theta}(s_0 \to s, k)] \phi(s) && (21)\\
&= \sum_s \eta(s) \phi(s) && (22)\\
&= (\sum_s \eta(s)) \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \phi(s) && (23)\\
&\propto \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \phi(s) && (24)\\
&= \sum_s \nu^\pi(s) \sum_a Q^{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a|s) && (25)
\end{aligned}$$

证明完毕!