**强化学习理论学习与代码实现**

# 深度强化学习-Policy Gradient策略梯度

#### “前言”

在前面讲到的 DQN 系列强化学习算法中，我们主要对\*\*价值函数\*\*进行了近似表示，基于价值来学习策略。这种基于价值的强化学习方法在很多领域都得到比较好的应用，但是也有很多局限性，因此在另一些场景下需要借助其他的方法。

从本篇博文开始，我们将进入基于策略的强化学习，该类方法直接\*\*对策略进行参数化\*\*，从而显式地将策略表示出来，策略的获取不再和值函数有关，并根据期望报酬相对于策略参数的梯度进行更新。

## 11.1 强化学习分类

强化学习按照学习方式的不同可以分为：基于值函数的强化学习、基于策略的强化学习和两者结合的 Actor-Critic 方法。

- 基于值函数的强化学习：通过学习值函数，隐式地获取策略（如 $\epsilon$-贪婪策略）；

- 基于策略地强化学习：不需要学习值函数，而是直接通过对策略参数化进行学习，因而是显式的；

- Actor-Critic：既学习值函数又学习策略，集众家之所长。

![](https://imgkr.cn-bj.ufileos.com/693a30d6-cf2d-4b98-898d-a624b1c9f718.png)

<center>图1 强化学习分类</center>

## 11.2 基于值函数的强化学习方法的不足

DQN 系列强化学习算法主要的问题主要有三点。

1. 对\*\*连续动作\*\*的处理能力不足。DQN 之类的方法一般都是只处理离散动作，无法处理连续动作，像开车这样的环境中，动作往往是油门大小和方向盘角度，这两个值都是一定范围内的实数，有无数种选择，以 DQN 为代表的基于值函数的方法，需要通过最大化操作确定动作，最大化本质上是一个优化过程，那么强化学习没学习一步都要进行这样的优化势必导致学习过程相当缓慢，但是基于策略的强化学习方法却很容易建模。

2. 对\*\*受限状态\*\*下的问题处理能力不足。在使用特征来描述状态空间中的某一个状态时，有可能因为个体观测的限制或者建模的局限，导致真实环境下本来不同的两个状态却在我们建模后拥有相同的特征描述，进而很有可能导致我们的基于值函数的方法无法得到最优解。此时使用基于策略的强化学习方法也很有效。

3. 无法解决\*\*随机策略\*\*问题。基于值函数的强化学习方法对应的最优策略通常是\*\*确定性策略\*\*，因为其是从众多动作价值中选择一个最大价值的动作，而有些问题的最优策略却是随机策略，这种情况下同样是无法通过基于值函数的学习来求解的。这时也可以考虑使用基于策略的强化学习方法。

由于上面这些原因，基于值函数的强化学习方法不能通吃所有的场景，我们需要新的解决上述类别问题的方法，这就是今天要讲的基于策略的强化学习方法。

## 11.3 基于策略的强化学习表示

在之前的部分我们都是使用参数 $\theta$ 来近似值函数或动作值函数的：

$$\begin{aligned}

V\_{\theta}(s) & \approx V^{\pi}(s) \\

Q\_{\theta}(s, a) & \approx Q^{\pi}(s, a)

\end{aligned}$$

对于基于策略的方法我们采用了同样的思路，不同的是现在直接对策略进行参数化：

$$\pi\_{\theta}(s,a) = P(a|s,\theta)\approx \pi(a|s)$$

将策略表示成一个连续的函数后，我们就可以用连续函数的优化方法来寻找最优的策略了。而最常用的方法就是梯度上升法了，那么这个梯度对应的优化目标如何定义呢？

## 11.4 策略目标函数

给定参数为 $\theta$ 的策略 $\pi\_{\theta}(s, a)$，目的是要找到最优的 $\theta$，但是如何评价一个策略 $\pi\_{\theta}$ 的好坏呢？

在片段环境中我们可以使用\*\*初始值\*\*作为优化目标，即：

$$J\_{1}(\theta)=V^{\pi\_{\theta}}\left(s\_{1}\right)=\mathbb{E}\_{\pi\_{\theta}}\left[v\_{1}\right]$$

但是在连续环境中是没有明确的初始状态的，那么我们的优化目标可以定义\*\*平均价值\*\*，即：

$$J\_{a v V}(\theta)=\sum\_{s} d^{\pi\_{\theta}}(s) V^{\pi\_{\theta}}(s)$$

其中 $d^{\pi\_{\theta}}(s)$ 是基于策略 $\pi\_{\theta}$ 生成的马尔科夫链关于状态的静态分布。

或者定义为每一时间步的\*\*平均奖励\*\*，即：

$$J\_{\operatorname{avR}}(\theta)=\sum\_{s} d^{\pi\_{\theta}}(s) \sum\_{a} \pi\_{\theta}(s, a) \mathcal{R}\_{s}^{a}$$

无论我们是采用 $J\_{1}$,$J\_{a v V}$ 还是 $J\_{\operatorname{avR}}$ 来表示优化目标，最终对 $\theta$ 求导的梯度都可以表示为：

$$\nabla\_{\theta} J(\theta)=\mathbb{E}\_{\pi\_{\theta}}\left[\nabla\_{\theta} \log \pi\_{\theta}(s, a) Q^{\pi\_{\theta}}(s, a)\right]$$

这就是\*\*策略梯度理论\*\*。

当然我们还可以采用很多其他可能的优化目标来做梯度上升，此时我们的梯度式子里面的 $\nabla\_{\theta}log \pi\_{\theta}(s,a)$ 部分并不改变，变化的只是后面的 $Q\_{\pi}(s,a)]$ 部分。对于 $\nabla\_{\theta}log \pi\_{\theta}(s,a)$,我们一般称为\*\*得分函数\*\*（score function）。

现在梯度的式子已经有了，后面剩下的就是策略函数 $\pi\_{\theta}(s,a)$ 的设计了。

## 11.5 策略函数的设计

现在我们回头看一下策略函数 $\pi\_{\theta}(s,a)$ 的设计，在前面它一直是一个数学符号。

最常用的策略函数就是\*\*softmax\*\*策略函数了，它主要应用于离散空间中，softmax 策略使用描述状态和动作的特征 $\phi(s,a)$ 与参数 $\theta$ 的线性组合来权衡一个动作发生的几率,即:

$$\pi\_{\theta}(s,a) = \frac{e^{\phi(s,a)^T\theta}}{\sum\limits\_be^{\phi(s,b)^T\theta}}$$

则通过求导很容易求出对应的得分函数为：

$$\nabla\_{\theta}log \pi\_{\theta}(s,a) = \phi(s,a) - \mathbb{E}\_{\pi\_{\theta}}[\phi(s,.)]$$

在连续动作空间中常用的策略为\*\*高斯策略\*\*，该策略对应的动作从高斯分布 $\mathbb{N(\phi(s)^T\theta, \sigma^2)}$ 中产生。高斯策略对应的得分函数求导可以得到为:

$$\nabla\_{\theta}log \pi\_{\theta}(s,a) = \frac{(a-\phi(s)^T\theta)\phi(s)}{\sigma^2}$$

## 11.6 Monte-Carlo 策略梯度(REINFORCE)

这里我们讨论最简单的策略梯度算法，蒙特卡罗策略梯度 reinforce 算法, 使用价值函数 $v(s)$ 来近似代替策略梯度公式里面的 $Q\_{\pi}(s,a)$。算法的流程很简单，如下所示：

![](https://imgkr.cn-bj.ufileos.com/599e4dfd-21fc-4c2c-9177-0ca772002692.png)

## 11.7 代码实现 REINFORCE

代码针对的环境的是 CliffWalkingEnv，在该环境中智能体在一个 4x12 的网格中移动，状态编号如下所示：

```

[[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11],

[12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],

[24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],

[36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47]]

```

在任何阶段开始时，初始状态都是状态 36，状态 47 是唯一的终止状态，悬崖对应的是状态 37 到 46。智能体有 4 个可选动作（UP = 0，RIGHT = 1，DOWN = 2，LEFT = 3）。智能体每走一步都会得到-1 的奖励，跌入悬崖会得到-100 的奖励并重置到起点，当达到目标时，片段结束。

完整的代码如下：

```

import gym

import itertools

import matplotlib

import numpy as np

import sys

import tensorflow as tf

import collections

if "../" not in sys.path:

sys.path.append("../")

from Lib.envs.cliff\_walking import CliffWalkingEnv

from Lib import plotting

matplotlib.style.use('ggplot')

env = CliffWalkingEnv()

class PolicyEstimator():

"""

策略函数逼近器

"""

def \_\_init\_\_(self, learning\_rate=0.01, scope="policy\_estimator"):

with tf.variable\_scope(scope):

self.state = tf.placeholder(tf.int32, [], "state")

self.action = tf.placeholder(dtype=tf.int32, name="action")

self.target = tf.placeholder(dtype=tf.float32, name="target")

# 使用状态的独热形式表达

state\_one\_hot = tf.one\_hot(self.state, int(env.observation\_space.n))

self.output\_layer = tf.contrib.layers.fully\_connected(

inputs=tf.expand\_dims(state\_one\_hot, 0),

num\_outputs=env.action\_space.n,

activation\_fn=None,

weights\_initializer=tf.zeros\_initializer)

self.action\_probs = tf.squeeze(tf.nn.softmax(self.output\_layer))

self.picked\_action\_prob = tf.gather(self.action\_probs, self.action)

# 损失

self.loss = -tf.log(self.picked\_action\_prob) \* self.target

self.optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate)

self.train\_op = self.optimizer.minimize(

self.loss, global\_step=tf.contrib.framework.get\_global\_step())

def predict(self, state, sess=None):

sess = sess or tf.get\_default\_session()

return sess.run(self.action\_probs, {self.state: state})

def update(self, state, target, action, sess=None):

sess = sess or tf.get\_default\_session()

feed\_dict = {self.state: state, self.target: target, self.action: action}

\_, loss = sess.run([self.train\_op, self.loss], feed\_dict)

return loss

class ValueEstimator():

"""

Value Function approximator.

"""

def \_\_init\_\_(self, learning\_rate=0.1, scope="value\_estimator"):

with tf.variable\_scope(scope):

self.state = tf.placeholder(tf.int32, [], "state")

self.target = tf.placeholder(dtype=tf.float32, name="target")

# This is just table lookup estimator

state\_one\_hot = tf.one\_hot(self.state, int(env.observation\_space.n))

self.output\_layer = tf.contrib.layers.fully\_connected(

inputs=tf.expand\_dims(state\_one\_hot, 0),

num\_outputs=1,

activation\_fn=None,

weights\_initializer=tf.zeros\_initializer)

self.value\_estimate = tf.squeeze(self.output\_layer)

self.loss = tf.squared\_difference(self.value\_estimate, self.target)

self.optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate)

self.train\_op = self.optimizer.minimize(

self.loss, global\_step=tf.contrib.framework.get\_global\_step())

def predict(self, state, sess=None):

sess = sess or tf.get\_default\_session()

return sess.run(self.value\_estimate, {self.state: state})

def update(self, state, target, sess=None):

sess = sess or tf.get\_default\_session()

feed\_dict = {self.state: state, self.target: target}

\_, loss = sess.run([self.train\_op, self.loss], feed\_dict)

return loss

def reinforce(env, estimator\_policy, estimator\_value, num\_episodes, discount\_factor=1.0):

"""

REINFORCE (Monte Carlo策略梯度)算法.使用策略梯度优化策略函数逼近器.

参数:

env: OpenAI环境.

estimator\_policy: 待优化的策略函数

estimator\_value: Value function approximator, used as a baseline

num\_episodes: 迭代次数

discount\_factor: 折扣因子

返回值:

EpisodeStats对象，包含两个numpy数组，分别存储片段长度和片段奖励.

"""

# 统计有用信息

stats = plotting.EpisodeStats(

episode\_lengths=np.zeros(num\_episodes),

episode\_rewards=np.zeros(num\_episodes))

Transition = collections.namedtuple("Transition", ["state", "action", "reward", "next\_state", "done"])

for i\_episode in range(num\_episodes):

# 重置环境

state = env.reset()

episode = []

# 环境中的每一步

for t in itertools.count():

# 单步执行

action\_probs = estimator\_policy.predict(state)

action = np.random.choice(np.arange(len(action\_probs)), p=action\_probs)

next\_state, reward, done, \_ = env.step(action)

# 记录转移

episode.append(Transition(

state=state, action=action, reward=reward, next\_state=next\_state, done=done))

# 更新统计

stats.episode\_rewards[i\_episode] += reward

stats.episode\_lengths[i\_episode] = t

print("\rStep {} @ Episode {}/{} ({})".format(

t, i\_episode + 1, num\_episodes, stats.episode\_rewards[i\_episode - 1]), end="")

# sys.stdout.flush()

if done:

break

state = next\_state

# 遍历每个片段并进行策略更新

for t, transition in enumerate(episode):

# The return after this timestep

total\_return = sum(discount\_factor \*\* i \* t.reward for i, t in enumerate(episode[t:]))

# Calculate baseline/advantage

# baseline\_value = estimator\_value.predict(transition.state)

# advantage = total\_return - baseline\_value

# Update our value estimator

# estimator\_value.update(transition.state, total\_return)

# 更新策略逼近器

estimator\_policy.update(transition.state, total\_return, transition.action)

return stats

tf.reset\_default\_graph()

global\_step = tf.Variable(0, name="global\_step", trainable=False)

policy\_estimator = PolicyEstimator()

value\_estimator = ValueEstimator()

with tf.Session() as sess:

sess.run(tf.initialize\_all\_variables())

# 注意,由于策略存在随机性，为了学习好的策略，一般片段数量为2000~5000

stats = reinforce(env, policy\_estimator, value\_estimator, 2000, discount\_factor=1.0)

plotting.plot\_episode\_stats(stats, smoothing\_window=25)

```

实验结果如下：

随着迭代的进行，片段长度逐渐下降，说明代理很快就能到达目标。

![](https://imgkr.cn-bj.ufileos.com/f4d8ae0a-ce44-489e-b452-333530e81ded.png)

随着迭代的进行，片段奖励逐渐上升并趋于稳定。

![](https://imgkr.cn-bj.ufileos.com/3bc696b7-936a-4f41-8963-4b82027daf14.png)