**强化学习理论学习与代码实现**

# 第十二章 Actor-Critic演员评论家

我们在上一章中介绍了策略梯度(Policy Gradient)方法，并代码练习了蒙特卡罗策略梯度reinforce算法。但是由于该算法需要完整的状态序列，同时单独对策略函数进行迭代更新，不太容易收敛。

在本章我们将讨论一种将策略(Policy Based)和价值(Value Based)相结合的方法：Actor-Critic算法，在强化学习领域最受欢迎的A3C算法，DDPG算法，PPO算法等都是AC框架，所以AC重要性不言而喻。

## 12.1 Actor-Critic算法简介

Actor-Critic从名字上看包括两部分，演员(Actor)和评价家(Critic)。其中Actor使用的是上一章讲到的策略函数，负责生成动作(Action)并和环境交互。而Critic使用我们之前讲到了的价值函数，负责评估Actor的表现，并指导Actor下一阶段的动作。

回想上一篇的策略梯度，策略函数就是我们的Actor，但是那里是没有Critic的，当时使用了蒙特卡罗法来计算每一步的价值部分替代了Critic的功能，但是场景比较受限。因此现在我们使用类似DQN中用的价值函数来替代蒙特卡罗法，作为一个比较通用的Critic。

也就是说在Actor-Critic算法中，我们需要做两组近似，第一组是策略函数的近似：

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

第二组是价值函数的近似，对于状态价值和动作价值函数分别是：

$$\hat{v}(s, w) \approx v\_{\pi}(s)$$

$$\hat{q}(s,a,w) \approx q\_{\pi}(s,a)$$

对于我们上一节讲到的蒙特卡罗策略梯度reinforce算法，需要进行改造才能变成Actor-Critic算法，将会在下一节中详细介绍。

首先，在蒙特卡罗策略梯度reinforce算法中，策略的参数更新公式是：

$$\theta = \theta + \alpha \nabla\_{\theta}log \pi\_{\theta}(s\_t,a\_t) v\_t$$

梯度更新部分中，$\nabla\_{\theta}log \pi\_{\theta}(s\_t,a\_t)$是得分函数，无需改变，要变成Actor的话改动的是$v\_t$，这块不能再使用蒙特卡罗法来得到，而应该从Critic得到。

而对于Critic来说，完全可以参考之前DQN的做法，即用一个Q网络来做为Critic, 这个Q网络的输入可以是状态，而输出是每个动作的价值或者最优动作的价值。

总体上来说，就是Critic通过Q网络计算状态的最优价值vt, 而Actor利用$v\_t$这个最优价值迭代更新策略函数的参数θ,进而选择动作，并得到反馈和新的状态，Critic使用反馈和新的状态更新Q网络参数w, 在后面Critic会使用新的网络参数w来帮Actor计算状态的最优价值$v\_t$。

## 12.2 Actor-Critic框架引出

在上一章中我们已经得到策略梯度的更新公式如下：

$$\nabla\_{\theta} J(\theta)\approx \frac{1}{N} \sum\_{n=1}^{N} \sum\_{t=1}^{T\_{n}}\left(\sum\_{t=t'}^{T\_{n}} \gamma^{t'-t} r\_{t'}^{n}-b\right) \nabla \log \pi\_{\theta}\left(a\_{t}^{n} | s\_{t}^{n}\right)$$

这里将$\sum\_{t=t'}^{T\_{n}} \gamma^{t'-t} r\_{t'}^{n}$记为$G\_t^n$，由于$G\_t^n$通过交互得到，其值非常不稳定（由于环境的动态性，$G\_t^n$本身也是一个分布），方差会比较大，因此需要寻找减少方差的办法。一种方法就是在上一章中采用的添加基线b， 这个b会使得$G\_t-b$的期望不变，但是方差会变小，常用的baseline函数就是$V(s\_t)$，在此基础上，为了进一步降低$G\_t$的随机性，我们用$G\_t^n$的期望$E(G\_t^n)$替代$G\_t^n$，这样上面的更新公式变为：

$$\nabla\_{\theta} J(\theta)\approx \frac{1}{N} \sum\_{n=1}^{N} \sum\_{t=1}^{T\_{n}}\left(E(G\_t^n)-V(s\_t^n)\right) \nabla \log \pi\_{\theta}\left(a\_{t}^{n} | s\_{t}^{n}\right)$$

在根据Q学习部分（），可知期望$E(G\_t^n)$就是在状态$s\_t$下执行动作$a\_t$，并遵循策略$\pi$所能得到的Q值，即$E(G\_t)$=$Q^{\pi\_{\theta} }\left(s^n\_t,a^n\_t\right)$,由此得到下式：

$$\nabla\_{\theta} J(\theta)\approx \frac{1}{N} \sum\_{n=1}^{N} \sum\_{t=1}^{T\_{n}}\left(Q^{\pi\_{\theta} }\left(s^n\_t,a^n\_t\right)-V(s\_t^n)\right) \nabla \log \pi\_{\theta}\left(a\_{t}^{n} | s\_{t}^{n}\right)$$

上式中存在的问题是，需要两个网络来分别预测Q和V，这就无形中增加了误差来源，考虑到贝尔曼等式，即：

$$Q^{\pi\_{\theta} }\left(s^n\_t,a^n\_t\right)=E\left[ r\_t^n+V^\pi(s\_{t+1}^n)\right]$$

这里将期望去掉（<font color='red'>个人理解，虽然去掉期望会导致有偏，但是最终还是会收敛到真实值</font>）：

$$Q^{\pi\_{\theta} }\left(s^n\_t,a^n\_t\right)= r\_t^n+V^{\pi\_\theta}(s\_{t+1}^n)$$

那么最终就得到：

$$\nabla\_{\theta} J(\theta)\approx \frac{1}{N} \sum\_{n=1}^{N} \sum\_{t=1}^{T\_{n}}\left(r\_t^n+V^{\pi\_\theta}(s\_{t+1}^n)-V(s\_t^n)\right) \nabla \log \pi\_{\theta}\left(a\_{t}^{n} | s\_{t}^{n}\right)$$

这样只需要一个网络就可以估算出V值了，而估算V的网络正是我们在Q-learning 中做的，所以我们就把这个网络叫做Critic。这样就在Policy Gradient算法的基础上引进了Q-learning 算法了。

## 12.3 Actor-Critic算法流程

Critic使用神经网络来计算TD误差并更新网络参数，Actor将TD误差作为输入，也使用神经网络来更新网络参数

算法输入：迭代轮数T，状态特征维度n, 动作集A, 步长α,β，衰减因子γ, 探索率ϵ, Critic网络结构和Actor网络结构。

输出：Actor 网络参数θ, Critic网络参数w

1.随机初始化所有的状态和动作对应的价值Q

2.for i from 1 to T，进行迭代。

&emsp;&emsp;a)初始化S为当前状态序列的第一个状态, 得到其特征向量$\phi(S)$

&emsp;&emsp;b)在Actor网络中使用$\phi(S)$作为输入，输出动作A,基于动作A得到新的状态S′,奖励r。

&emsp;&emsp;c)在Critic网络中分别使用$\phi(S)$，$\phi(S)， \phi(S’)$作为输入，得到Q值输出V(S)，V(S′)

&emsp;&emsp;d)计算TD误差$\delta = R +\gamma V(S’) -V(S)$

&emsp;&emsp;e)使用均方差损失函数$\sum\limits(R +\gamma V(S’) -V(S,w))^2$作Critic网络参数w的梯度更新

&emsp;&emsp;f)更新Actor网络参数θ:

$$\theta = \theta + \alpha \nabla\_{\theta}log \pi\_{\theta}(S\_t,A)\delta$$

对于Actor的得分函数∇θlogπθ(St,A),可以选择softmax或者高斯分值函数。

## 12.4 代码练习

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。

AC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

"""

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")

# 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=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)

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():

"""

值函数逼近器

"""

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 actor\_critic(env, estimator\_policy, estimator\_value, num\_episodes, discount\_factor=1.0):

"""

Actor Critic 算法.通过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: 值函数逼近器，用作评论家

num\_episodes: 回合数

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

"""

# Keeps track of useful statistics

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):

state = env.reset()

episode = []

for t in itertools.count():

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)

episode.append(Transition(

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

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

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

# 计算TD目标

value\_next = estimator\_value.predict(next\_state)

td\_target = reward + discount\_factor \* value\_next

td\_error = td\_target - estimator\_value.predict(state)

# 更新值函数逼近

estimator\_value.update(state, td\_target)

# 更新策略逼近

# 使用TD误差作为优势估计

estimator\_policy.update(state, td\_error, action)

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

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

if done:

break

state = next\_state

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())

stats = actor\_critic(env, policy\_estimator, value\_estimator, 300)

plotting.plot\_episode\_stats(stats, smoothing\_window=10)

```

结果如下：

![](https://imgkr.cn-bj.ufileos.com/8101e40f-597c-40d6-b408-1f2927247d8d.png)

![](https://imgkr.cn-bj.ufileos.com/60831e55-e11b-41d7-9687-58e40388fb5c.png)