Keras+OpenAI强化学习实践：行为-评判模型

By 机器之心2017年8月14日 14:24

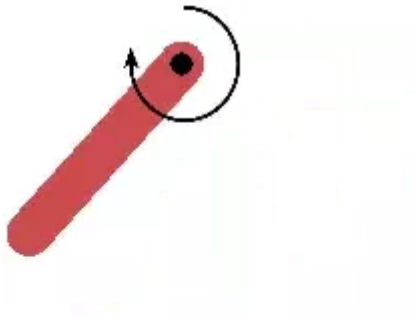
> 本文先给出行为-评判模型（actor-critic model）的基本原理，包括链式求导法则等，随后再从模型的参数、模型的训练和模型的测试等方面用代码段解释行为-评判模型，最后，文章给出了该教程的全部代码。

像之前的教程一样，我们首先快速了解一下已取得的惊人成果：在一个连续的输出空间场景下，从完全不明白「胜利」的含义开始，现在我们可以探索环境并「完成」试验。

将自身置身于模拟环境中。这就相当于要求你在没有游戏说明书和特定目标的场景下玩一场游戏，且不可中断，直到你赢得整个游戏（这有些近乎残忍）。不仅如此：一系列动作可能产生的结果状态是无穷无尽的（即连续观测空间）！然而，DQN 通过调控并缓慢更新各动作内部参数的值可以快速解决这个看似不可能的任务。

更复杂的环境

从以前的 MountainCar 环境向 Pendulum 环境的升级与 CartPole 到 MountainCar 的升级极其相似：我们正在从一个离散的环境扩展到连续的环境。Pendulum 环境具有无限的输入空间，这意味着你在任何给定时间可以进行不限次数的动作。为何 DQN 不再适用此类环境了？DQN 的实现不是完全独立于环境动作的结构吗？



不同于 MountainCar-v0，Pendulum-v0 通过给我们提供了无穷大的输入空间而构成了更大的挑战。

虽然它与动作无关，但 DQN 的基本原理是拥有有限的输出空间。毕竟，我们要考虑该如何构建代码：预测会在每个时间步长（给定当前环境状态）下为每个可能的动作分配一个分数，并简单地采用得分最高的动作。我们之前已经简化了强化学习的问题，以便高效地为每个动作分配分数。但是，如果我们输入空间无限大，这还有可能吗？我们需要一个无限大的表格来记录所有的 Q 值！

Action	Q
3123	0.201
3124	0.660938188
3125	0.448031374
3126	0.108435243
3127	0.712210279
3128	0.109236197
3129	0.75745233
3130	0.296933752
3131	0.344963051
3132	0.472829123
3133	0.634630915

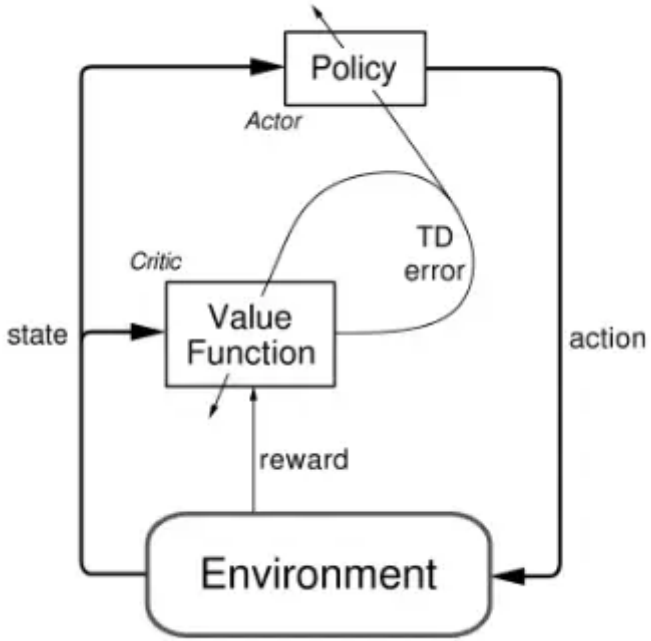
无限大的数据表听起来很不靠谱！

我们该如何着手解决这个看似不可能的任务呢？毕竟，我们现在要做比以前更加疯狂的事：不仅仅只是赢下一场没有攻略的游戏，现在我们还要应对一个被无数条指令控制的游戏！让我们来看看为什么 DQN 只接收有限数量的动作输入。

我们从模型的搭建方式来分析。我们必须能够在每个时间步迭代更新特定的动作位置的改变方式。这正是为什么我们让模型预测 Q 值，而不是直接预测下一步的动作。如果选择了后者，我们不知道如何更新模型以更好地预测，以及从对未来的预测中获利。

因此，本质问题源于一个事实——类似于模型已经输出与所有可能发生的行动相关的奖励的列表运算结果。如果我们把这个模型拆解开会怎样？如果我们有两个独立的模型：一个输出期望的动作（在连续空间中），另一个以它的输出作为输入，以产生 DQN 的 Q 值？这貌似解决了我们的问题，而且这正是行为-评判模型（actor-critic model）的基本原理！

行为-评判模型原理



不同于 DQN 算法，行为-评判模型（如名字所示）有两个独立的网络：一个基于当前的环境状态预测出即将被采用的动作，另一个用于计算状态和动作下的价值。

正如上节所述，整个行为—评判（AC）方法可行的前提是有两个交互模型。多个神经网络之间相互关联的主题在强化学习和监督学习（即 GAN、AC、A3C、DDQN（升级版 DQN）等）中越发凸显。初次了解这些架构可能有些困难，但这绝对值得去做：你将能够理解和编程实现一些现代领域研究前沿的算法！

回到主题，AC 模型有两个恰如其分的名字：行为和评判。前者接受当前的环境状态，并决定从哪个状态获得最佳动作。它以非常类似于人类自身的行为方式来实现 DQN 算法。评判模块通过从 DQN 中接受环境状态和动作并返回一个表征动作状态的分数来完成评判功能。

把这想象成是一个孩子（「行为模块」）与其父母（「评判模块」）的游乐场。孩子正在环顾四周，探索周边环境中的所有可能选择，例如滑动幻灯片，荡秋千，在草地上玩耍。父母会照看孩子并基于其所为，对孩子给出批评或补充。父母的决定依赖于环境的事实无可否认：毕竟，如果孩子试图在真实的秋千上玩耍，相比于试图在幻灯片上这样做，孩子更值得表扬！

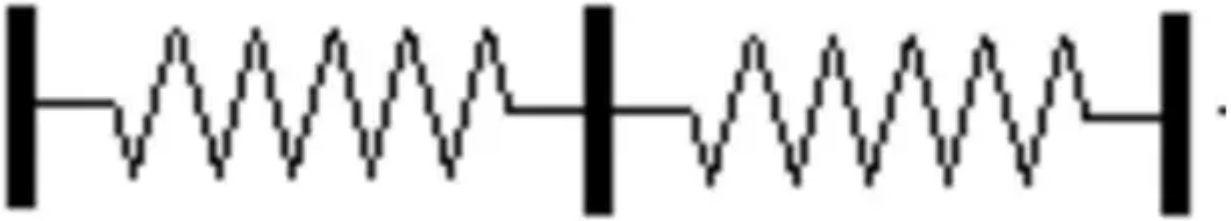
简介：链式法则（可选）

你需要理解的主要理论在很大程度上支撑着现代机器学习：链式法则。毫不夸张的说链式法则可能是掌握理解实用机器学习的最关键的（即使有些简单）的想法之一。事实上，如果只是直观地了解链式法则的概念，你并不需要很深厚的数学背景。我会非常快速地讲解链式法则，但如果你已了解，请随意跳到下一部分，下一部分我们将看到开发 AC 模型的实际概述，以及链条法则如何适用于该规划。

$$\frac{dy}{dx} = \frac{du}{dx} \times \frac{dy}{du}$$

一个看似可能来自你第一节微积分课堂上的简单概念，构成了实用机器学习的现代基础，因为它在反向推算和类似算法中有着令人难以置信的加速运算效果。

这个等式看起来非常直观：毕竟只是「重写了分子/分母」。这个「直观的解释」有一个主要问题：等式中的推导完全是倒退的！关键是要记住，数学中引入直观的符号是为了帮助我们理解概念。因此，由于链式法则的计算方式非常类似于简化分数的运算过程，人们才引入这种「分数」符号。那么试图通过符号来解释概念的人正在跳过关键的一步：为什么这些符号可以通用？如同这里，为什么要像这样进行求导？



可以借助经典的弹簧实例可视化运动中的链条规则

基本概念实际上并不比这个符号更难理解。想象一下，我们把一捆绳子一根根地系在一起，类似于把一堆弹簧串联起来。假设你固定了这个弹簧系统的一端，你的目标是以 10 英尺/秒的速度摇晃另一端，那么你可以用这个速度摇动你的末端，并把它传播到另一端。或者你可以连接一些中间系统，以较低的速率摇动中间连接，例如，5 英尺/秒。也就是说，在 5 英尺/秒的情况下，你只需要以 2 英尺/秒的速度摇动末端，因为你从开始到终点做的任何运动都会被传递到终点位置。这是因为物理连接迫使一端的运动被传递到末端。注意：和其它类比一样，这里有一些不当之处，但这主要是为了可视化。

类似地，如果我们有两个系统，其中一个系统的输出是另一个系统的输入，微调「反馈网络」的参数将会影响其输出，该输出会被传播下去并乘以任何进一步的变化值并贯穿整个网络。

AC 模型概述

因此，我们必须制定一个 ActorCritic 类，它包含一些之前实现过的 DQN，但是其训练过程更复杂。因为我们需要一些更高级的功能，我们必须使用包含了基础库 Keras 的开源框架：Tensorflow。注意：你也可以在 Theano 中实现这一点，但是我以前没有使用过它，所以没有包含其代码。如果你选择这么做，请随时向 Theano 提交此代码的扩展。

模型实现包含四个主要部分，其直接并行如何实现 DQN 代理：

- 模型参数/配置
- 训练代码
- 预测代码

AC 参数

第一步，导入需要的库

```
import gym
import numpy as np
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Input
from keras.layers.merge import Add, Multiply
from keras.optimizers import Adam
import keras.backend as K

import tensorflow as tf

import random
from collections import deque
```

参数与 DQN 中的参数非常类似。毕竟，这个行为-评判模型除了两个独立的模块之外，还要做与 DQN 相同的任务。我们还继续使用我们在 DQN 报告中讨论的「目标网络攻击」，以确保网络成功收敛。唯一的新参数是「tau」，并且涉及在这种情况下如何进行目标网络学习的细微变化：

```
class ActorCritic:
    def __init__(self, env, sess):
        self.env  = env
        self.sess = sess
        self.learning_rate = 0.001
        self.epsilon = 1.0
        self.epsilon_decay = .995
```

```
    self.gamma = .95
    self.tau   = .125
    self.memory = deque(maxlen=2000)
```

在以下的训练部分中，详细解释了 tau 参数的准确用法，它的作用其实就是推动预测模型向目标模型逐步转换。现在，我们找到了主要的关注点：定义模型。正如我们所描述的，我们有两个独立的模型，每个模型都与它自己的目标网络相关联。

我们从定义行为模型开始。行为模型的目的是根据当前环境状态，得出应当采取的最佳动作。再次，这个模型需要处理我们提供的数字数据，这意味着没有空间也没有必要在网络中添加任何比我们迄今为止使用的密集/完全连接层更复杂的层。因此，行为模型只是一系列全连接层，将环境观察的状态映射到环境空间上的一个点：

```
def create_actor_model(self):
    state_input = Input(shape=self.env.observation_space.shape)
    h1 = Dense(24, activation='relu')(state_input)
    h2 = Dense(48, activation='relu')(h1)
    h3 = Dense(24, activation='relu')(h2)
    output = Dense(self.env.action_space.shape[0],
      activation='relu')(h3)

    model = Model(input=state_input, output=output)
    adam  = Adam(lr=0.001)
    model.compile(loss="mse", optimizer=adam)
    return state_input, model
```

主要的区别是我们返回了一个对输入层的引用。本节结尾对此原因的解释十分清楚，但简而言之，这解释了我们为什么对行为模型的训练过程采取不同的处理。

行为模型中的棘手部分是决定如何训练它，这就是链式法则发挥作用的地方。但在讨论之前，让我们考虑一下为什么它与标准评论/ DQN 网络的训练不同。毕竟，我们不是简单地去适应在 DQN（根据当前状态拟合模型）的情况下，基于当前的和打折的未来奖励得出接下来采用的最佳动作是

哪一个？问题在于：如果我们能够按照要求去做，那么这个问题将会解决。问题在于我们如何确定「最佳动作」是什么，因为 Q 值现在是在评判网络中单独计算出来的。

所以，为了解决这个问题，我们选择了一种替代方法。不同于找到「最佳选择」和拟合，我们实际上选择了爬山算法（梯度上升）。对于不熟悉这个算法的人来说，登山是一个形象的比喻：从你当地的 POV，找到斜率最大的倾斜方向，并沿着该方向逐步移动。换句话说，爬山正试图通过原始的冲动并沿着局部斜率最大的方向来达到全局最大值。可以想象在某些情况下，该方法大错特错，但通常情况下，它具备很好的实用性。

因此，我们想使用该算法来更新我们的行为模型：我们想确定（行为模型中的）参数的什么变化会导致 Q 值最大幅度的增加（由评判模型预测得出）。由于行为模型的输出是动作，评判模型通过环境状态+动作对来评估，我们在此可以看到链式法则如何发挥作用。我们想看看如何改变行为模型的参数才会改变最终的 Q 值，使用行为网络的输出作为我们的「中间链接」（下面的代码全部在「__init __（self）」方法中）：

```
self.actor_state_input, self.actor_model = \
    self.create_actor_model()
_, self.target_actor_model = self.create_actor_model()
self.actor_critic_grad = tf.placeholder(tf.float32,
    [None, self.env.action_space.shape[0]])

actor_model_weights = self.actor_model.trainable_weights
self.actor_grads = tf.gradients(self.actor_model.output,
    actor_model_weights, -self.actor_critic_grad)
grads = zip(self.actor_grads, actor_model_weights)
self.optimize =  tf.train.AdamOptimizer(
    self.learning_rate).apply_gradients(grads)
```

我们看到在这里我们持有模型权重和输出（动作）之间的梯度。我们还通过负的 self.actor_critic_grad（因为我们想在这种情况下使用梯度上升）来放缩它，梯度由占位符持有。对于那些不熟悉 Tensorflow 或首次接触的读者，你只需要知道运行 Tensorflow 会话时，占位符将扮演「输入数据」的角色。我不会详细介绍它的工作原理，因为 tensorflow.org 教程的材料相当全面。

再来看看评判网络，基本上我们面临着相反的问题。即，网络定义稍微复杂一些，但是训练比较简单。评判网络旨在将环境状态和动作作为输入，并计算出相应的估值。我们通过合并一系列全连接层以及得出最终的 Q 值预测之前的中间层来实现这一点：

```
def create_critic_model(self):
    state_input = Input(shape=self.env.observation_space.shape)
    state_h1 = Dense(24, activation='relu')(state_input)
    state_h2 = Dense(48)(state_h1)

    action_input = Input(shape=self.env.action_space.shape)
    action_h1    = Dense(48)(action_input)

    merged    = Add()([state_h2, action_h1])
    merged_h1 = Dense(24, activation='relu')(merged)
    output = Dense(1, activation='relu')(merged_h1)
    model  = Model(input=[state_input,action_input],
      output=output)

    adam  = Adam(lr=0.001)
    model.compile(loss="mse", optimizer=adam)
    return state_input, action_input, model
```

需要注意的是如何处理输入和返回的不对称性。对于第一点，我们在环境状态输入中有一个额外的 FC（全连接）层。我这样做是因为这是推荐 AC 网络使用的结构，但它可能与处理两个输入的 FC 层效果差不多（或稍差）。至于后面一点（我们正在返回的值），我们需要保留输入状态和动作的引用，因为我们需要使用它们更新行为网络：

```
self.critic_state_input, self.critic_action_input, \
    self.critic_model = self.create_critic_model()
_, _, self.target_critic_model = self.create_critic_model()
self.critic_grads = tf.gradients(self.critic_model.output,
    self.critic_action_input)

# Initialize for later gradient calculations
self.sess.run(tf.initialize_all_variables())
```

我们在这里设置要计算的缺失梯度：关于动作权重的输出 Q。这是训练代码中直接调用的，我们现在来深入探讨。

AC 模型训练

该代码的最后一个主要与 DQN 不同的部分是实际的训练代码。然而，我们使用了从记忆（LSTM 结构）中吸取教训和学习的基本结构。由于我们有两种训练方法，我们将代码分成不同的训练函数，并将它们称为：

```
def train(self):
    batch_size = 32
    if len(self.memory) < batch_size:
        return
    rewards = []
    samples = random.sample(self.memory, batch_size)
    self._train_critic(samples)
    self._train_actor(samples)
```

现在我们定义两种训练方法。不过，与 DQN 非常相似：我们只是简单地找到未来打折的奖励和训练方法。唯一的区别是，我们正在对状态/动作对进行训练，并使用 target_critic_model 来预测未来的奖励，而不是仅使用行为来预测：

```
def _train_critic(self, samples):
    for sample in samples:
        cur_state, action, reward, new_state, done = sample
        if not done:
            target_action =
                self.target_actor_model.predict(new_state)
            future_reward = self.target_critic_model.predict(
                [new_state, target_action])[0][0]
            reward += self.gamma * future_reward
        self.critic_model.fit([cur_state, action],
            reward, verbose=0)
```

对于行为模型，我们幸运地解决了所有难题！我们已经设置了梯度如何在网络中运作，现在只需传入当前的动作和状态并调用该函数：

```
def _train_actor(self, samples):
    for sample in samples:
        cur_state, action, reward, new_state, _ = sample
        predicted_action = self.actor_model.predict(cur_state)
        grads = self.sess.run(self.critic_grads, feed_dict={
            self.critic_state_input:  cur_state,
            self.critic_action_input: predicted_action
        })[0]
        self.sess.run(self.optimize, feed_dict={
            self.actor_state_input: cur_state,
            self.actor_critic_grad: grads
        })
```

如上所述，我们利用了目标模型。所以我们必须在每个时间步更新其权重。但是，更新过程太慢了。具体说，我们将目标模型的估值保持在一个分数 self.tau 上，并将其更新为余数（1-self.tau）分数的相应模型权重。行为/评判模型均如此处理，但下面只给出行为模型的代码（你可以在文章底部的完整代码中看到评判模型代码）：

```
def _update_actor_target(self):
    actor_model_weights  = self.actor_model.get_weights()
    actor_target_weights =self.target_critic_model.get_weights()

    for i in range(len(actor_target_weights)):
        actor_target_weights[i] = actor_model_weights[i]
    self.target_critic_model.set_weights(actor_target_weights
```

AC 模型预测

这与我们在 DQN 中的做法一样，所以没有什么好说的：

```
def act(self, cur_state):
  self.epsilon *= self.epsilon_decay
  if np.random.random() < self.epsilon:
   return self.env.action_space.sample()
  return self.actor_model.predict(cur_state)
```

预测代码

预测代码也与之前的强化学习算法相同。也就是说，我们只需反复试验，并对代理进行预测、记忆和训练：

```
def main():
    sess = tf.Session()
    K.set_session(sess)
    env = gym.make("Pendulum-v0")
    actor_critic = ActorCritic(env, sess)

    num_trials = 10000
    trial_len  = 500

    cur_state = env.reset()
    action = env.action_space.sample()
    while True:
        env.render()
        cur_state = cur_state.reshape((1,
            env.observation_space.shape[0]))
        action = actor_critic.act(cur_state)
        action = action.reshape((1, env.action_space.shape[0]))

        new_state, reward, done, _ = env.step(action)
        new_state = new_state.reshape((1,
            env.observation_space.shape[0]))
```

```
        actor_critic.remember(cur_state, action, reward,
           new_state, done)
        actor_critic.train()

        cur_state = new_state
```

完整代码

这是使用 AC（Actor-Critic）对「Pendulum-v0」环境进行训练的完整代码！

```
"""
solving pendulum using actor-critic model
"""

import gym
import numpy as np
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Input
from keras.layers.merge import Add, Multiply
from keras.optimizers import Adam
import keras.backend as K

import tensorflow as tf

import random
from collections import deque

# determines how to assign values to each state, i.e. takes the state
# and action (two-input model) and determines the corresponding value
class ActorCritic:
        def __init__(self, env, sess):
                self.env  = env
                self.sess = sess
```

```
        self.learning_rate = 0.001
        self.epsilon = 1.0
        self.epsilon_decay = .995
        self.gamma = .95
        self.tau   = .125

        # ===================================================================== #
        #                               Actor Model                             #
        # Chain rule: find the gradient of chaging the actor network params in  #
        # getting closest to the final value network predictions, i.e. de/dA    #
        # Calculate de/dA as = de/dC * dC/dA, where e is error, C critic, A act #
        # ===================================================================== #

        self.memory = deque(maxlen=2000)
        self.actor_state_input, self.actor_model = self.create_actor_model()
        _, self.target_actor_model = self.create_actor_model()

        self.actor_critic_grad = tf.placeholder(tf.float32,
            [None, self.env.action_space.shape[0]]) # where we will feed de/dC (from critic)

        actor_model_weights = self.actor_model.trainable_weights
        self.actor_grads = tf.gradients(self.actor_model.output,
            actor_model_weights, -self.actor_critic_grad) # dC/dA (from actor)
        grads = zip(self.actor_grads, actor_model_weights)
        self.optimize = tf.train.AdamOptimizer(self.learning_rate).apply_gradients(grads)

        # ===================================================================== #
        #                               Critic Model                            #
        # ===================================================================== #

        self.critic_state_input, self.critic_action_input, \
            self.critic_model = self.create_critic_model()
        _, _, self.target_critic_model = self.create_critic_model()

        self.critic_grads = tf.gradients(self.critic_model.output,
            self.critic_action_input) # where we calcaulte de/dC for feeding above

        # Initialize for later gradient calculations
        self.sess.run(tf.initialize_all_variables())

    # ========================================================================= #
    #                              Model Definitions                            #
```

```
    # ========================================================================== #

    def create_actor_model(self):
        state_input = Input(shape=self.env.observation_space.shape)
        h1 = Dense(24, activation='relu')(state_input)
        h2 = Dense(48, activation='relu')(h1)
        h3 = Dense(24, activation='relu')(h2)
        output = Dense(self.env.action_space.shape[0], activation='relu')(h3)

        model = Model(input=state_input, output=output)
        adam  = Adam(lr=0.001)
        model.compile(loss="mse", optimizer=adam)
        return state_input, model

    def create_critic_model(self):
        state_input = Input(shape=self.env.observation_space.shape)
        state_h1 = Dense(24, activation='relu')(state_input)
        state_h2 = Dense(48)(state_h1)

        action_input = Input(shape=self.env.action_space.shape)
        action_h1    = Dense(48)(action_input)

        merged    = Add()([state_h2, action_h1])
        merged_h1 = Dense(24, activation='relu')(merged)
        output = Dense(1, activation='relu')(merged_h1)
        model  = Model(input=[state_input,action_input], output=output)

        adam  = Adam(lr=0.001)
        model.compile(loss="mse", optimizer=adam)
        return state_input, action_input, model

    # ========================================================================== #
    #                         Model Training                                     #
    # ========================================================================== #

    def remember(self, cur_state, action, reward, new_state, done):
        self.memory.append([cur_state, action, reward, new_state, done])

    def _train_actor(self, samples):
        for sample in samples:
            cur_state, action, reward, new_state, _ = sample
            predicted_action = self.actor_model.predict(cur_state)
```

```
            grads = self.sess.run(self.critic_grads, feed_dict={
                self.critic_state_input:  cur_state,
                self.critic_action_input: predicted_action
            })[0]

            self.sess.run(self.optimize, feed_dict={
                self.actor_state_input: cur_state,
                self.actor_critic_grad: grads
            })

    def _train_critic(self, samples):
        for sample in samples:
            cur_state, action, reward, new_state, done = sample
            if not done:
                target_action = self.target_actor_model.predict(new_state)
                future_reward = self.target_critic_model.predict(
                    [new_state, target_action])[0][0]
                reward += self.gamma * future_reward
            self.critic_model.fit([cur_state, action], reward, verbose=0)

    def train(self):
        batch_size = 32
        if len(self.memory) < batch_size:
            return

        rewards = []
        samples = random.sample(self.memory, batch_size)
        self._train_critic(samples)
        self._train_actor(samples)

    # ========================================================================= #
    #                   Target Model Updating                                   #
    # ========================================================================= #

    def _update_actor_target(self):
        actor_model_weights  = self.actor_model.get_weights()
        actor_target_weights = self.target_critic_model.get_weights()

        for i in range(len(actor_target_weights)):
            actor_target_weights[i] = actor_model_weights[i]
        self.target_critic_model.set_weights(actor_target_weights)
```

```
        def _update_critic_target(self):
            critic_model_weights  = self.critic_model.get_weights()
            critic_target_weights = self.critic_target_model.get_weights()

            for i in range(len(critic_target_weights)):
                critic_target_weights[i] = critic_model_weights[i]
            self.critic_target_model.set_weights(critic_target_weights)

        def update_target(self):
            self._update_actor_target()
            self._update_critic_target()

        # ========================================================================= #
        #                         Model Predictions                               #
        # ========================================================================= #

        def act(self, cur_state):
            self.epsilon *= self.epsilon_decay
            if np.random.random() < self.epsilon:
                return self.env.action_space.sample()
            return self.actor_model.predict(cur_state)

    def main():
        sess = tf.Session()
        K.set_session(sess)
        env = gym.make("Pendulum-v0")
        actor_critic = ActorCritic(env, sess)

        num_trials = 10000
        trial_len  = 500

        cur_state = env.reset()
        action = env.action_space.sample()
        while True:
            env.render()
            cur_state = cur_state.reshape((1, env.observation_space.shape[0]))
            action = actor_critic.act(cur_state)
            action = action.reshape((1, env.action_space.shape[0]))

            new_state, reward, done, _ = env.step(action)
            new_state = new_state.reshape((1, env.observation_space.shape[0]))
```

```
            actor_critic.remember(cur_state, action, reward, new_state, done)
            actor_critic.train()

            cur_state = new_state

if __name__ == "__main__":
    main()
```

[原文地址：https://medium.com/towards-data-science/reinforcement-learning-w-keras-openai-actor-critic-models-f084612cfd69](https://medium.com/towards-data-science/reinforcement-learning-w-keras-openai-actor-critic-models-f084612cfd69)

声明：本文由机器之心编译出品，原文来自Medium，作者Yash Patel，转载请查看要求，机器之心对于违规侵权者保有法律追诉权。

[工程强化学习深度强化学习深度学习DQNactor-critic](#)

[提交评论](#)

登录后参与评论[去登录](#)

[关于我们寻求报道商务合作](#)

友情链接

Synced Global机器之心 Medium 博客PaperWeekly网易智能动脉网

联系电话：+86 010-57150141

联系邮箱：contact@jiqizhixin.com