Keras+OpenAI强化学习实践：深度Q网络

By 机器之心2017年8月22日 11:18

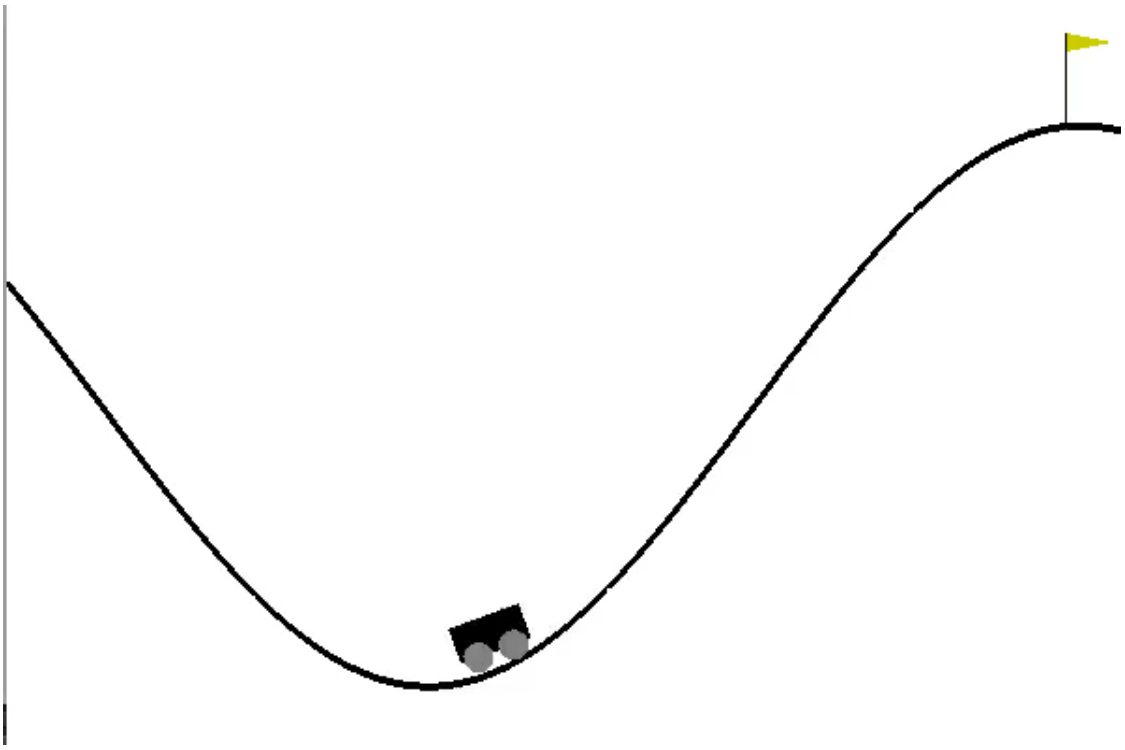
本文先给出 Q 学习（Q-learning）的基本原理，然后再具体从 DQN 网络的超参数、智能体、模型和训练等方面详细解释了深度 Q 网络，最后，文章给出了该教程的全部代码。

在之前的 Keras/OpenAI 教程中，我们讨论了一个将深度学习应用于强化学习环境的基础案例，它的效果非常显著。想象作为训练数据的完全随机序列（series）。任何两个序列都不可能高度彼此重复，因为这些都是随机产生的。然而，成功的试验之间存在相同的关键特征，例如在 CartPole 游戏中，当杆往右靠时需要将车向右推，反之亦然。因此，通过在所有这些试验数据上训练我们的神经网络，我们提取了有助于成功的共同模式（pattern），并能够平滑导致其产生独立故障的细节。

话虽如此，我们认为这次的环境比上次要困难得多，即游戏：MountainCar。

更复杂的环境

即使看上去我们应该能够应用与上周相同的技术，但是有一个关键特征使它变得不可能：我们无法生成训练数据。与简单的 CartPole 例子不同，采取随机移动通常只会导致实验的结果很差（谷底）。也就是说，我们的实验结果最后都是相同的-200。这用作训练数据几乎没有用。想象一下，如果你无论在考试中做出什么答案，你都会得到 0%，那么你将如何从这些经验中学习？



「MountainCar-v0」环境的随机输入不会产生任何对于训练有用的输出。

由于这些问题，我们必须找出一种能逐步改进以前实验的方法。为此，我们使用强化学习最基本的方法：Q-learning！

DQN 的理论背景

Q-learning 的本质是创建一个「虚拟表格」，这个表格包含了当前环境状态下每个可能的动作能得到多少奖励。下面来详细说明：

Action	Q
1	0.3432
2	0.2341
3	0.219

网络可以想象为内生有电子表格的网络，该表格含有当前环境状态下可能采取的每个可能的动作的值。

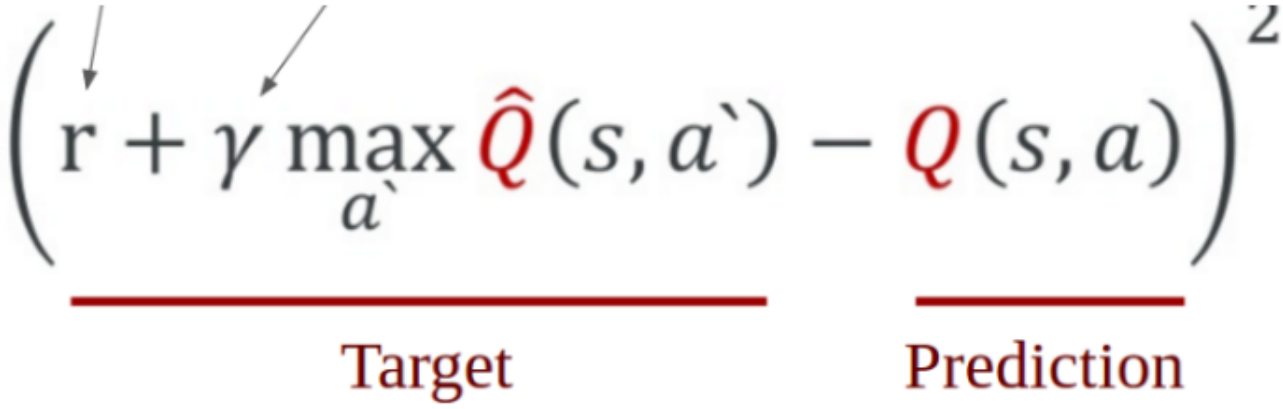
「虚拟表格」是什么意思？想像一下，对于输入空间的每个可能的动作，你都可以为每个可能采取的动作赋予一个分数。如果这可行，那么你可以很容易地「打败」环境：只需选择具有最高分数的动作！但是需要注意 2 点：首先，这个分数通常被称为「Q-分数」，此算法也由此命名。第二，与任何其它得分一样，这些 Q-分数在其模拟的情境外没有任何意义。也就是说，它们没有确定的意义，但这没关系，因为我们只需要做比较。

为什么对于每个输入我们都需要一个虚拟表格？难道没有统一的表格吗？原因是这样做不和逻辑：这与在谷底谈采取什么动作是最好的，及在向左倾斜时的最高点讨论采取什么动作是最好的是一样的道理。

现在，我们的主要问题（为每个输入建立虚拟表格）是不可能的：我们有一个连续的（无限）输入空间！我们可以通过离散化输入空间来解决这个问题，但是对于本问题来说，这似乎是一个非常棘手的解决方案，并且在将来我们会一再遇到。那么，我们如何解决呢？那就是通过将神经网络应用于这种情况：这就是 DQN 中 D 的来历！

DQN agent

现在，我们现在已经将问题聚焦到：找到一种在给定当前状态下为不同动作赋值 Q-分数的方法。这是使用任何神经网络时遇到的非常自然的第一个问题的答案：我们模型的输入和输出是什么？本模型中你需要了解的数学方程是以下等式（不用担心，我们会在下面讲解）：



如上所述，Q 代表了给定当前状态（s）和采取的动作（a）时我们模型估计的价值。然而，目标是确定一个状态价值的总和。那是什么意思？即从该位置获得的即时奖励和将来会获得的预期奖励之和。也就是说，我们要考虑一个事实，即一个状态的价值往往不仅反映了它的直接收益，而且还反映了它的未来收益。在任何情况下，我们会将未来的奖励折现，因为对于同样是收到$100 的两种情况（一种为将来，一种为现在），我会永远选择现在的交易，因为未来是会变化的。γ因子反映了此状态预期未来收益的贬值。

这就是我们需要的所有数学！下面是实际代码的演示！

DQN agent 实现

深度 Q 网络为持续学习（continuous learning），这意味着不是简单地累积一批实验/训练数据并将其传入模型。相反，我们通过之前运行的实验创建训练数据，并且直接将运行后创建的数据馈送如模型。如果现在感到好像有些模糊，别担心，该看看代码了。代码主要在定义一个 DQN 类，其中将实现所有的算法逻辑，并且我们将定义一组简单的函数来进行实际的训练。

DQN 超参数

首先，我们将讨论一些与 DQN 相关的参数。它们大多数是实现主流神经网络的标准参数：

```
class DQN:
    def __init__(self, env):
        self.env    = env
        self.memory  = deque(maxlen=2000)

        self.gamma = 0.95
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.01
```

让我们来一步一步地讲解这些超参数。第一个是环境（env），这仅仅是为了在建立模型时便于引用矩阵的形状。「记忆（memory）」是 DQN 的关键组成部分：如前所述，我们不断通过实验训练模型。然而与直接训练实验的数据不同，我们将它们先添加到内存中并随机抽样。为什么这样做呢，难道仅仅将最后 x 个实验数据作为样本进行训练不好吗？原因有点微妙。设想我们只使用最近的实验数据进行训练：在这种情况下，我们的结果只会学习其最近的动作，这可能与未来的预测没有直接的关系。特别地，在本环境下，如果我们在斜坡右侧向下移动，使用最近的实验数据进行训练将需要在斜坡右侧向上移动的数据上进行训练。但是，这与在斜坡左侧的情景需决定采取的动作无关。所以，通过抽取随机样本，将保证不会偏离训练集，而是理想地学习我们将遇到的所有环境。

我们现在来讨论模型的超参数：gamma、epsilon 以及 epsilon 衰减和学习速率。第一个是前面方程中讨论的未来奖励的折现因子（<1），最后一个是标准学习速率参数，我们不在这里讨论。第二个是 RL 的一个有趣方面，值得一谈。在任何一种学习经验中，我们总是在探索与利用之间做出选择。这不仅限于计算机科学或学术界：我们每天都在做这件事！

考虑你家附近的饭店。你最后一次尝试新饭店是什么时候？可能很久以前。这对应于你从探索到利用的转变：与尝试找到新的更好的机会不同，你根据自己以往的经验找到最好的解决方案，从而最大化效用。对比当你刚搬家时：当时你不知道什么饭店是好的，所以被诱惑去探索新选择。换句话说，这时存在明确的学习趋势：当你不了解它们时，探索所有的选择，一旦你对其中的一些建立了意见，就逐渐转向利用。以同样的方式，我们希望我们的模型能够捕捉这种自然的学习模型，而 epsilon 扮演着这个角色。

Epsilon 表示我们将致力于探索的时间的一小部分。也就是说，实验的分数 self.epsilon，我们将仅仅采取随机动作，而不是我们预测在这种情况下最好的动作。如上所述，我们希望在开始时形成稳定评估之前更经常地采取随机动作：因此开始时初始化ε接近 1.0，并在每一个连续的时间步长中以小于 1 的速率衰减它。

DQN 模型

在上面的 DQN 的初始化中排除了一个关键环节：用于预测的实际模型！在原来的 Keras RL 教程中，我们直接给出数字向量形式的输入和输出。因此，除了全连接层之外，不需要在网络中使用更复杂的层。具体来说，我们将模型定义为：

```
def create_model(self):
    model   = Sequential()
    state_shape  = self.env.observation_space.shape
    model.add(Dense(24, input_dim=state_shape[0],
        activation="relu"))
    model.add(Dense(48, activation="relu"))
    model.add(Dense(24, activation="relu"))
    model.add(Dense(self.env.action_space.n))
    model.compile(loss="mean_squared_error",
        optimizer=Adam(lr=self.learning_rate))
    return model
```

并用它来定义模型和目标模型（如下所述）：

```
def __init__(self, env):
    self.env     = env
    self.memory  = deque(maxlen=2000)

    self.gamma = 0.95
    self.epsilon = 1.0
    self.epsilon_min = 0.01
    self.epsilon_decay = 0.995
    self.learning_rate = 0.01
    self.tau = .05

    self.model = self.create_model()
    # "hack" implemented by DeepMind to improve convergence
    self.target_model = self.create_model()
```

事实上，有两个单独的模型，一个用于做预测，一个用于跟踪「目标值」，这是反直觉的。明确地说，模型（self.model）的作用是对要采取的动作进行实际预测，目标模型（self.target_model）的作用是跟踪我们想要模型采取的动作。

为什么不用一个模型做这两件事呢？毕竟，如果预测要采取的动作，那不会间接地确定我们想要模型采取的模式吗？这实际上是 DeepMind 发明的深度学习的「不可思议的技巧」之一，它用于在 DQN 算法中获得收敛。如果使用单个模型，它可以（通常会）在简单的环境（如 CartPole）中收敛。但是，在这些更为复杂的环境中并不收敛的原因在于我们如何对模型进行训练：如前所述，我们正在对模型进行「即时」训练。

因此，在每个时间步长进行训练模型，如果我们使用单个网络，实际上也将在每个时间步长时改变「目标」。想想这将多么混乱！那就如同，开始老师告诉你要完成教科书中的第 6 页，当你完成了一半时，她把它改成了第 9 页，当你完成一半的时候，她告诉你做第 21 页！因此，由于缺乏明确方向以利用优化器，即梯度变化太快难以稳定收敛，将导致收敛不足。所以，作为代偿，我们有一个变化更慢的网络以跟踪我们的最终目标，和一个最终实现这些目标的网络。

DQN 训练

训练涉及三个主要步骤：记忆、学习和重新定位目标。第一步基本上只是随着实验的进行向记忆添加数据：

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

这里没有太多的注意事项，除了我们必须存储「done」阶段，以了解我们以后如何更新奖励函数。转到 DQN 主体的训练函数。这是使用存储记忆的地方，并积极从我们过去看到的内容中学习。首先，从整个存储记忆中抽出一个样本。我们认为每个样本是不同的。正如我们在前面的等式中看到的，我们要将 Q-函数更新为当前奖励之和与预期未来奖励的总和（贬值为 gamma）。在实验结束时，将不再有未来的奖励，所以该状态的价值为此时我们收到的奖励之和。然而，在非终止状态，如果我们能够采取任何可能的动作，将会得到的最大的奖励是什么？我们得到：

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

    samples = random.sample(self.memory, batch_size)
```

```
    for sample in samples:
        state, action, reward, new_state, done = sample
        target = self.target_model.predict(state)
        if done:
            target[0][action] = reward
        else:
            Q_future = max(
                self.target_model.predict(new_state)[0])
            target[0][action] = reward + Q_future * self.gamma
        self.model.fit(state, target, epochs=1, verbose=0)
```

最后，我们必须重新定位目标，我们只需将主模型的权重复制到目标模型中。然而，与主模型训练的方法不同，目标模型更新较慢：

```
def target_train(self):
    weights = self.model.get_weights()
    target_weights = self.target_model.get_weights()
    for i in range(len(target_weights)):
        target_weights[i] = weights[i]
    self.target_model.set_weights(target_weights)
```

DQN 动作

最后一步是让 DQN 实际执行希望的动作，在给定的 epsilon 参数基础上，执行的动作在随机动作与基于过去训练的预测动作之间选择，如下所示：

```
def act(self, state):
    self.epsilon *= self.epsilon_decay
    self.epsilon = max(self.epsilon_min, self.epsilon)
    if np.random.random() < self.epsilon:
        return self.env.action_space.sample()
    return np.argmax(self.model.predict(state)[0])
```

训练 agent

现在训练我们开发的复杂的 agent。将其实例化，传入经验数据，训练 agent，并更新目标网络：

```
def main():
    env     = gym.make("MountainCar-v0")
    gamma   = 0.9
    epsilon = .95

    trials  = 100
    trial_len = 500

    updateTargetNetwork = 1000
    dqn_agent = DQN(env=env)
    steps = []
    for trial in range(trials):
        cur_state = env.reset().reshape(1,2)
        for step in range(trial_len):
            action = dqn_agent.act(cur_state)
            env.render()
            new_state, reward, done, _ = env.step(action)

            reward = reward if not done else -20
            print(reward)
            new_state = new_state.reshape(1,2)
            dqn_agent.remember(cur_state, action,
                reward, new_state, done)


            dqn_agent.replay()
            dqn_agent.target_train()

            cur_state = new_state
            if done:
                break
        if step >= 199:
            print("Failed to complete trial")
        else:
            print("Completed in {} trials".format(trial))
            break
```

完整的代码

这就是使用 DQN 的「MountainCar-v0」环境的完整代码！

```
import gym

import numpy as np

import random
```

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam
from collections import deque
class DQN:
  def __init__(self, env):
    self.env     = env
    self.memory  = deque(maxlen=2000)

    self.gamma = 0.85
    self.epsilon = 1.0
    self.epsilon_min = 0.01
    self.epsilon_decay = 0.995
    self.learning_rate = 0.005
    self.tau = .125
    self.model        = self.create_model()
    self.target_model = self.create_model()
  def create_model(self):
    model   = Sequential()
    state_shape  = self.env.observation_space.shape
    model.add(Dense(24, input_dim=state_shape[0], activation="relu"))
    model.add(Dense(48, activation="relu"))
    model.add(Dense(24, activation="relu"))
    model.add(Dense(self.env.action_space.n))
```

```
        model.compile(loss="mean_squared_error",
            optimizer=Adam(lr=self.learning_rate))
        return model
    def act(self, state):
        self.epsilon *= self.epsilon_decay
        self.epsilon = max(self.epsilon_min, self.epsilon)
        if np.random.random() < self.epsilon:
            return self.env.action_space.sample()
        return np.argmax(self.model.predict(state)[0])
    def remember(self, state, action, reward, new_state, done):
        self.memory.append([state, action, reward, new_state, done])
    def replay(self):
        batch_size = 32
        if len(self.memory) < batch_size:
            return
        samples = random.sample(self.memory, batch_size)
        for sample in samples:
            state, action, reward, new_state, done = sample
            target = self.target_model.predict(state)
            if done:
                target[0][action] = reward
            else:
                Q_future = max(self.target_model.predict(new_state)[0])
                target[0][action] = reward + Q_future * self.gamma
```

```
            self.model.fit(state, target, epochs=1, verbose=0)
    def target_train(self):
        weights = self.model.get_weights()
        target_weights = self.target_model.get_weights()
        for i in range(len(target_weights)):
            target_weights[i] = weights[i] * self.tau + target_weights[i] * (1 - self.tau)
        self.target_model.set_weights(target_weights)
    def save_model(self, fn):
        self.model.save(fn)
def main():
    env     = gym.make("MountainCar-v0")
    gamma   = 0.9
    epsilon = .95
    trials  = 1000
    trial_len = 500
    # updateTargetNetwork = 1000
    dqn_agent = DQN(env=env)
    steps = []
    for trial in range(trials):
        cur_state = env.reset().reshape(1,2)
        for step in range(trial_len):
            action = dqn_agent.act(cur_state)
            new_state, reward, done, _ = env.step(action)
            # reward = reward if not done else -20
```

```
            new_state = new_state.reshape(1,2)
            dqn_agent.remember(cur_state, action, reward, new_state, done)

            dqn_agent.replay()       # internally iterates default (prediction) model
            dqn_agent.target_train() # iterates target model
            cur_state = new_state
            if done:
                break
        if step >= 199:
            print("Failed to complete in trial {}".format(trial))
            if step % 10 == 0:
                dqn_agent.save_model("trial-{}.model".format(trial))
        else:
            print("Completed in {} trials".format(trial))
            dqn_agent.save_model("success.model")
            break
if __name__ == "__main__":
    main()
```

原文链接：https://medium.com/towards-data-science/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c

工程KerasOpenAI强化学习DQN游戏

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

京 ICP 备 12027496

全球人工智能信息服务

友情链接

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

联系电话：+86 010-57150141

联系邮箱：contact@jiqizhixin.com