简单粗暴 TensorFlow

发布 *0.3 beta*

Xihan Li（雪麒）

2018 年 08 月 28 日

Contents

基于 *Eager Execution* / *Based on Eager Execution*

本手册是一篇精简的 TensorFlow 入门指导，基于 TensorFlow 的 Eager Execution（动态图）模式，力图让具备一定机器学习及 Python 基础的开发者们快速上手 TensorFlow。

友情提醒：如果发现阅读中有难以理解的部分，请检查自己对每章的“前置知识”部分是否有清楚的理解。

答疑区 - TensorFlow 中文社区“简单粗暴 TensorFlow”版面：https://www.tensorflowers.cn/b/48 （如果您对本教程有任何疑问，请至 TensorFlow 中文社区的该版面发问）

PDF 下载：https://www.tensorflowers.cn/t/6230

GitHub: https://github.com/snowkylin/TensorFlow-cn

This handbook is a concise introduction to TensorFlow based on TensorFlow' s Eager Execution mode, trying to help developers get started with TensorFlow quickly with some basic machine learning and Python knowledge.

Friendly reminder: If you find something difficult to understand in reading, please check if you have a clear understanding of the "Prerequisites" part of each chapter.

Q&A area - TensorFlow Chinese community "A Concise Handbook of TensorFlow" forum: https://www.tensorflowers.cn/b/48 (If you have any questions about this tutorial, please ask in this forum of the TensorFlow Chinese community)

PDF download: https://www.tensorflowers.cn/t/6230

GitHub: https://github.com/snowkylin/TensorFlow-cn

CHAPTER 1

前言

2018 年 3 月 30 日，Google 在加州山景城举行了第二届 TensorFlow Dev Summit 开发者峰会，并宣布正式发布 TensorFlow 1.8 版本。笔者有幸获得 Google 的资助亲临峰会现场，见证了这一具有里程碑式意义的新版本发布。众多新功能的加入和支持展示了 TensorFlow 的雄心壮志，同时早在 2017 年秋就开始测试的 Eager Execution（动态图机制）在这一版本中终于正式加入，并成为了入门 TensorFlow 的官方推荐模式。

> The easiest way to get started with TensorFlow is using Eager Execution.
>
> ——https://www.tensorflow.org/get_started/

在此之前，TensorFlow 所基于的传统 Graph Execution 的弊端，如入门门槛高、调试困难、灵活性差、无法使用 Python 原生控制语句等早已被开发者诟病许久。一些新的基于动态图机制的深度学习框架（如 PyTorch）也横空出世，并以其易用性和快速开发的特性而占据了一席之地。尤其是在学术研究等需要快速迭代模型的领域，PyTorch 等新兴深度学习框架已经成为主流。笔者所在的数十人的机器学习实验室中，竟只有笔者一人“守旧”地使用 TensorFlow。然而，直到目前，市面上相关的 TensorFlow 相关的中文技术书籍及资料仍然基于传统的 Graph Execution 模式，让不少初学者（尤其是刚学过机器学习课程的大学生）望而却步。由此，在 TensorFlow 正式支持 Eager Execution 之际，有必要出现一本全新的技术手册，帮助初学者及需要快速迭代模型的研究者，以一个全新的角度快速入门 TensorFlow。

同时，本手册还有第二个任务。市面上与 TensorFlow 相关的中文技术书籍大部分都以深度学习为主线，而将 TensorFlow 作为这些深度学习模型的实现方式。这样固然有体系完整的优点，然而对于已经对机器学习或深度学习理论有所了解，希望侧重于学习 TensorFlow 本身的读者而言，就显得不够友好。同时，虽然 TensorFlow 有官方的教学文档（https://tensorflow.google.cn/tutorials），然而在体例上显得逻辑性不足，缺乏一般教学文档从浅入深，层次递进的特性，而更类似于一系列技术文档的罗列。于是，笔者希望编写一本手册，以尽量精简的篇幅展示 TensorFlow 作为一个计算框架的主要特性，并弥补官方手册的不足，力图能让已经有一定机器学习/深度学习知识及编程能力的读者迅速上手 TensorFlow，并在实际编程过程中可以随

时查阅并解决实际问题。

本手册的主要特征有：

- 主要基于 TensorFlow 最新的 Eager Execution（动态图）模式，以便于模型的快速迭代开发。但依然会包含传统的 Graph Execution 模式，代码上尽可能兼容两者；
- 定位以教学及工具书为主，编排以 TensorFlow 的各项概念和功能为核心，力求能够让 TensorFlow 开发者快速查阅。各章相对独立，不一定需要按顺序阅读。正文中不会出现太多关于深度学习和机器学习的理论介绍，但会提供若干阅读推荐以便初学者掌握相关基础知识；
- 代码实现均进行仔细推敲，力图简洁高效和表意清晰。模型实现均统一使用 TensorFlow 官方文档 最新提出的继承 `tf.keras.Model` 和 `tf.keras.layers.Layer` 的方式（在其他技术文档中鲜少介绍），保证代码的高度可复用性。每个完整项目的代码总行数均不过百行，让读者可以快速理解并举一反三；
- 注重详略，少即是多，不追求巨细靡遗和面面俱到，不进行大篇幅的细节论述。

在整本手册中，带“*”的部分均为选读。

本手册的暂定名称《简单粗暴 TensorFlow》是向我的好友兼同学 Chris Wu 编写的《简单粗暴 LaTeX》（https://github.com/wklchris/Note-by-LaTeX）致敬。该手册清晰精炼，是 LaTeX 领域不可多得的中文资料，也是我在编写这一技术文档时所学习的对象。本手册最初是在我的好友 Ji-An Li 所组织的深度学习研讨小组中，由我作为预备知识的讲义而编写和使用。好友们的才学卓著与无私分享的品格也是编写此拙作的重要助力。

本手册的英文版由我的好友 Zida Jin（1-4 章）和 Ming（5-6 章）翻译，并由 Ji-An Li 和笔者审校。三位朋友牺牲了自己的大量宝贵时间翻译和校对本手册，同时 Ji-An Li 亦对本手册的教学内容和代码细节提供了诸多宝贵意见。我谨向好友们为本手册的辛勤付出致以衷心的感谢。

衷心感谢 Google 中国开发者关系团队和 TensorFlow 工程团队的成员们对本手册编写所提供的帮助。其中包括开发者关系团队的 Luke Cheng 在本手册写作全程提供的思路启发和持续鼓励，开发者关系团队的 Rui Li, Pryce Mu 和 TensorFlow 社群维护的小伙伴们在本手册宣发及推广上提供的大力支持，以及 TensorFlow 团队的 Tiezhen Wang 在本手册工程细节方面提供的诸多建议和补充。

关于本手册的意见和建议，欢迎在 https://github.com/snowkylin/TensorFlow-cn/issues 提交。这是一个开源项目，您的宝贵意见将促进本手册的持续更新。

Xihan Li（雪麒）

2018 年 8 月于燕园

CHAPTER 2

TensorFlow 安装

TensorFlow 的最新安装步骤可参考官方网站上的说明（https://tensorflow.google.cn/install）。TensorFlow 支持 Python、Java、Go、C 等多种编程语言以及 Windows、OSX、Linux 等多种操作系统，此处及后文均以主流的 Python 语言为准。

以下提供简易安装和正式安装两种途径，供不同层级的读者选用。

2.1 简易安装

如果只是安装一个运行在自己电脑上的，无需 GPU 的简易环境，不希望在环境配置上花费太多精力，建议按以下步骤安装（以 Windows 系统为例）：

- 下载并安装 Python 集成环境 Anaconda （Python 3.6 版本）；
- 下载并安装 Python 的 IDE PyCharm （Community 版本，或学生可申请 Professional 版本的免费授权）；
- 打开开始菜单中的“Anaconda Prompt”，输入 `pip install tensorflow`。

完毕。

2.2 正式安装

该部分包含了更多安装上的细节（如建立 conda 环境），以及 GPU 版本 TensorFlow 的环境配置方法。

2.2.1 安装前的环境配置

正式安装 TensorFlow 前，需要为其配置合适的环境。步骤如下：

1. 检查自己的电脑是否具有 NVIDIA 显卡。如有，建议安装 GPU 版本的 TensorFlow，以利用 GPU 强大的计算加速能力[1] ，否则可以安装 CPU 版本。具体而言，该显卡的 CUDA Compute Capability 须不低于 3.0，可以到 NVIDIA 的官方网站 查询自己所用显卡的 CUDA Compute Capability；
2. 安装 Python 环境。此处建议安装 Anaconda，这是一个开源的 Python 发行版本，提供了一个完整的科学计算环境，包括 NumPy、SciPy 等常用科学计算库。当然，你有权选择自己喜欢的 Python 环境。注意截至本手册撰写时，TensorFlow 在 Windows 下的安装仅支持 Python 3.X 版本；
    - 安装 Anaconda 时，可以选择将 Anaconda 目录添加到系统的 PATH 中（虽然安装程序不推荐这样做），这样可以直接在命令行环境下使用 Anaconda 的各项功能。当然，不添加的话也可以使用开始菜单中的 Anaconda Prompt 进入命令行的 Anaconda 环境。
3. （针对 GPU 版本）安装 NVIDIA 显卡驱动程序、CUDA Toolkit 和 cuDNN 。值得注意的事项有：
    - 建议的顺序是：先安装最新版 NVIDIA 显卡驱动程序，再安装 CUDA（安装时不要选择同时安装驱动），最后安装 cuDNN。CUDA 附带的显卡驱动程序可能过旧；
    - 在 Ubuntu 下有一个很简易的驱动安装方法：在系统设置（System Setting）里面选软件与更新（Software & Updates），然后点选 Additional Drivers 里面的 “Using NVIDIA binary driver” 选项并点选右下角的 “Apply Changes” 即可，系统即会自动安装 NVIDIA 驱动。否则，NVIDIA 显卡驱动程序在 Linux 系统上的安装往往不会一帆风顺，注意在安装前禁用系统自带的开源显卡驱动 Nouveau、禁用主板的 Secure Boot 功能。更详细的指导可以参考 这篇文章 ；
    - CUDA Toolkit 和 cuDNN 的版本一定要与 TensorFlow 官方网站安装说明的版本一致，注意官方网站安装说明里要求安装的版本可能并非最新版本；
    - cuDNN 的安装方式比较特殊，你需要手动将下载的安装包复制到 CUDA 的安装目录下。

2.2.2 安装

在 Anaconda 环境下的安装过程如下（以 Windows 系统为例）：

1. 新建一个叫做 `tensorflow` 的 conda 环境

```
conda create -n tensorflow python=X.X # 注意这里的 X.X 填写自己 Python 环境的版本，例如 3.6
```

2. 激活环境

[1] GPU 加速的效果与模型类型和 GPU 的性能有关，如果 CPU 性能较高，但 GPU 仅有入门级的性能，其实速度提升不大，大概 1-2 倍。不过如果 GPU 性能强大的话（例如，本手册写作时，NVIDIA GeForce GTX 1080 Ti 或 NVIDIA GeForce TITAN 系列是市场上性能较强大的显卡型号），对于特定模型，十几倍甚至更高的加速效果也是可以达到的。同时，GPU 的加速效果与任务本身也有关。入门级的 TensorFlow 模型往往不需要太高的计算性能，CPU 版本的 TensorFlow 足以胜任，因此可以待到掌握 TensorFlow 的基本知识后，再决定是否购入更高级的 GPU 以得到更快的训练速度。

```
activate tensorflow
```

3. 使用 pip 安装 TensorFlow

安装 CPU 版本

```
pip install tensorflow
```

安装 GPU 版本

```
pip install tensorflow-gpu
```

如有需要，也可以安装 TensorFlow 的 Nightly 版本，该版本较之于正式版本会具有一些最新的特性（例如在 TensorFlow 1.8 版本以前，本手册主要使用的 Eager Execution 模式只在 Nightly 版本中提供），然而稳定度可能稍弱。在一个新的虚拟环境里运行 `pip install tf-nightly` （CPU 版本）或 `pip install tf-nightly-gpu` （GPU 版本）即可。注意，若安装 GPU 版本，其往往要求安装比正式版要求中更新的 CUDA 和 cuDNN。好在 CUDA 和 cuDNN 的不同版本是可以共存的。

如果使用 pip 命令安装速度较慢，可以尝试 清华大学开源软件镜像站的 TensorFlow 镜像。

2.3 第一个程序

安装完毕后，我们来编写一个简单的程序来验证安装。

在命令行下输入 `activate tensorflow` 进入之前建立的安装有 TensorFlow 的 conda 环境，再输入 `python` 进入 Python 环境，逐行输入以下代码：

```
import tensorflow as tf
tf.enable_eager_execution()

A = tf.constant([[1, 2], [3, 4]])
B = tf.constant([[5, 6], [7, 8]])
C = tf.matmul(A, B)

print(C)
```

如果能够最终输出：

```
tf.Tensor(
[[19 22]
[43 50]], shape=(2, 2), dtype=int32)
```

说明 TensorFlow 已安装成功。运行途中可能会输出一些 TensorFlow 的提示信息，属于正常现象。

此处使用的是 Python 语言，关于 Python 语言的入门教程可以参考 http://www.runoob.com/python3/python3-tutorial.html 或 https://www.liaoxuefeng.com ，本手册之后将默认读者拥有 Python 语言的基本

知识。不用紧张，Python 语言易于上手，而 TensorFlow 本身也不会用到 Python 语言的太多高级特性。关于 Python 的 IDE，建议使用 PyCharm 。如果你是学生并有.edu 结尾的邮箱的话，可以在 这里 申请免费的授权。如果没有，也可以下载社区版本的 PyCharm，主要功能差别不大。

CHAPTER 3

TensorFlow 基础

本章介绍 TensorFlow 的基本操作。

前置知识：

- Python 基本操作 （赋值、分支及循环语句、使用 import 导入库）；
- Python 的 With 语句 ；
- NumPy ， Python 下常用的科学计算库。TensorFlow 与之结合紧密；
- 向量 和 矩阵 运算（矩阵的加减法、矩阵与向量相乘、矩阵与矩阵相乘、矩阵的转置等。测试题：$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} = ?$）；
- 函数的导数 ， 多元函数求导 （测试题：$f(x,y) = x^2 + xy + y^2, \frac{\partial f}{\partial x} = ?, \frac{\partial f}{\partial y} = ?$）；
- 线性回归 ；
- 梯度下降方法 求函数的局部最小值。

3.1 TensorFlow 1+1

我们可以先简单地将 TensorFlow 视为一个科学计算库（类似于 Python 下的 NumPy）。这里以计算 $1+1$ 和 $\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}$ 作为 Hello World 的示例。

```
import tensorflow as tf
tf.enable_eager_execution()

a = tf.constant(1)
b = tf.constant(1)
c = tf.add(a, b)    # 也可以直接写 c = a + b, 两者等价

print(c)

A = tf.constant([[1, 2], [3, 4]])
B = tf.constant([[5, 6], [7, 8]])
C = tf.matmul(A, B)

print(C)
```

输出:

```
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(
[[19 22]
[43 50]], shape=(2, 2), dtype=int32)
```

以上代码声明了 a、b、A、B 四个 **张量** (Tensor), 并使用了 tf.add() 和 tf.matmul() 两个 **操作** (Operation) 对张量进行了加法和矩阵乘法运算, 运算结果即时存储于 c、C 两个张量内。张量的重要属性是其形状 (shape) 和类型 (dtype)。这里 a、b、c 是纯量, 形状为空, 类型为 int32; A、B、C 为 2×2 的矩阵, 形状为 (2, 2), 类型为 int32。

在机器学习中, 我们经常需要计算函数的导数。TensorFlow 提供了强大的 **自动求导机制**来计算导数。以下代码展示了如何使用 tf.GradientTape() 计算函数 $y(x) = x^2$ 在 $x = 3$ 时的导数:

```
import tensorflow as tf
tf.enable_eager_execution()

x = tf.get_variable('x', shape=[1], initializer=tf.constant_initializer(3.))
with tf.GradientTape() as tape:     # 在 tf.GradientTape() 的上下文内, 所有计算步骤都会被记录以用于求导
    y = tf.square(x)
y_grad = tape.gradient(y, x)        # 计算 y 关于 x 的导数
print([y.numpy(), y_grad.numpy()])
```

输出:

```
[array([9.], dtype=float32), array([6.], dtype=float32)]
```

这里 x 是一个初始化为 3 的 **变量** (Variable), 使用 tf.get_variable() 声明。与普通张量一样, 变量同样具

有形状(shape)和类型(dtype)属性,不过使用变量需要有一个初始化过程,可以通过在 `tf.get_variable()` 中指定 `initializer` 参数来指定所使用的初始化器。这里使用 `tf.constant_initializer(3.)` 将变量 `x` 初始化为 float32 类型的 `3.`[1]。变量与普通张量的一个重要区别是其默认能够被 TensorFlow 的自动求导机制所求导,因此往往被用于定义机器学习模型的参数。`tf.GradientTape()` 是一个自动求导的记录器,在其中的变量和计算步骤都会被自动记录。上面的示例中,变量 `x` 和计算步骤 `y = tf.square(x)` 被自动记录,因此可以通过 `y_grad = tape.gradient(y, x)` 求张量 `y` 对变量 `x` 的导数。

在机器学习中,更加常见的是对多元函数求偏导数,以及对向量或矩阵的求导。这些对于 TensorFlow 也不在话下。以下代码展示了如何使用 `tf.GradientTape()` 计算函数 $L(w,b)=\|Xw+b-y\|^2$ 在 $w=(1,2)^T, b=1$ 时分别对 w,b 的偏导数。其中 $X=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}, y=\begin{bmatrix}1\\ 2\end{bmatrix}$。

```
X = tf.constant([[1., 2.], [3., 4.]])
y = tf.constant([[1.], [2.]])
w = tf.get_variable('w', shape=[2, 1], initializer=tf.constant_initializer([[1.], [2.]]))
b = tf.get_variable('b', shape=[1], initializer=tf.constant_initializer([1.]))
with tf.GradientTape() as tape:
    L = 0.5 * tf.reduce_sum(tf.square(tf.matmul(X, w) + b - y))
w_grad, b_grad = tape.gradient(L, [w, b])        # 计算 L(w, b) 关于 w, b 的偏导数
print([L.numpy(), w_grad.numpy(), b_grad.numpy()])
```

输出:

```
[62.5, array([[35.],
   [50.]], dtype=float32), array([15.], dtype=float32)]
```

这里,`tf.square()` 操作代表对输入张量的每一个元素求平方,不改变张量形状。`tf.reduce_sum()` 操作代表对输入张量的所有元素求和,输出一个形状为空的纯量张量(可以通过 `axis` 参数来指定求和的维度,不指定则默认对所有元素求和)。TensorFlow 中有大量的张量操作 API,包括数学运算、张量形状操作(如 `tf.reshape()`)、切片和连接(如 `tf.concat()`)等多种类型,可以通过查阅 TensorFlow 的官方 API 文档[2]来进一步了解。

从输出可见,TensorFlow 帮助我们计算出了

$$L((1,2)^T,1)=62.5$$

$$\frac{\partial L(w,b)}{\partial w}\Big|_{w=(1,2)^T,b=1}=\begin{bmatrix}35\\ 50\end{bmatrix}$$

$$\frac{\partial L(w,b)}{\partial b}\Big|_{w=(1,2)^T,b=1}=15$$

[1] Python 中可以使用整数后加小数点表示将该整数定义为浮点数类型。例如 `3.` 代表浮点数 `3.0`。

[2] 主要可以参考 Tensor Transformations 和 Math 两个页面。可以注意到,TensorFlow 的张量操作 API 在形式上和 Python 下流行的科学计算库 NumPy 非常类似,如果对后者有所了解的话可以快速上手。

3.2 基础示例：线性回归

考虑一个实际问题，某城市在 2013 年-2017 年的房价如下表所示：

年份	2013	2014	2015	2016	2017
房价	12000	14000	15000	16500	17500

现在，我们希望通过对该数据进行线性回归，即使用线性模型 $y = ax + b$ 来拟合上述数据，此处 `a` 和 `b` 是待求的参数。

首先，我们定义数据，进行基本的归一化操作。

```
import numpy as np

X_raw = np.array([2013, 2014, 2015, 2016, 2017])
y_raw = np.array([12000, 14000, 15000, 16500, 17500])

X = (X_raw - X_raw.min()) / (X_raw.max() - X_raw.min())
y = (y_raw - y_raw.min()) / (y_raw.max() - y_raw.min())
```

接下来，我们使用梯度下降方法来求线性模型中两个参数 `a` 和 `b` 的值[3]。

回顾机器学习的基础知识，对于多元函数 $f(x)$ 求局部极小值，梯度下降 的过程如下：

- 初始化自变量为 x_0 ，$k = 0$
- 迭代进行下列步骤直到满足收敛条件：
  - 求函数 $f(x)$ 关于自变量的梯度 $\nabla f(x_k)$
  - 更新自变量：$x_{k+1} = x_k - \gamma \nabla f(x_k)$ 。这里 γ 是学习率（也就是梯度下降一次迈出的“步子”大小）
  - $k \leftarrow k + 1$

接下来，我们考虑如何使用程序来实现梯度下降方法，求得线性回归的解 $\min_{a,b} L(a,b) = \sum_{i=1}^{n}(ax_i + b - y_i)^2$ 。

3.2.1 NumPy

机器学习模型的实现并不是 TensorFlow 的专利。事实上，对于简单的模型，即使使用常规的科学计算库或者工具也可以求解。在这里，我们使用 NumPy 这一通用的科学计算库来实现梯度下降方法。NumPy 提供了多维数组支持，可以表示向量、矩阵以及更高维的张量。同时，也提供了大量支持在多维数组上进行操作的函数（比如下面的 `np.dot()` 是求内积，`np.sum()` 是求和）。在这方面，NumPy 和 MATLAB 比较类似。

[3] 其实线性回归是有解析解的。这里使用梯度下降方法只是为了展示 TensorFlow 的运作方式。

在以下代码中，我们手工求损失函数关于参数 a 和 b 的偏导数[4]，并使用梯度下降法反复迭代，最终获得 a 和 b 的值。

```
a, b = 0, 0

num_epoch = 10000
learning_rate = 1e-3
for e in range(num_epoch):
    # 手动计算损失函数关于自变量（模型参数）的梯度
    y_pred = a * X + b
    grad_a, grad_b = (y_pred - y).dot(X), (y_pred - y).sum()

    # 更新参数
    a, b = a - learning_rate * grad_a, b - learning_rate * grad_b

print(a, b)
```

然而，你或许已经可以注意到，使用常规的科学计算库实现机器学习模型有两个痛点：

- 经常需要手工求函数关于参数的偏导数。如果是简单的函数或许还好，但一旦函数的形式变得复杂（尤其是深度学习模型），手工求导的过程将变得非常痛苦，甚至不可行。
- 经常需要手工根据求导的结果更新参数。这里使用了最基础的梯度下降方法，因此参数的更新还较为容易。但如果使用更加复杂的参数更新方法（例如 Adam 或者 Adagrad），这个更新过程的编写同样会非常繁杂。

而 TensorFlow 等深度学习框架的出现很大程度上解决了这些痛点，为机器学习模型的实现带来了很大的便利。

3.2.2 TensorFlow

TensorFlow 的 **Eager Execution（动态图）模式**[5] 与上述 NumPy 的运行方式十分类似，然而提供了更快速的运算（GPU 支持）、自动求导、优化器等一系列对深度学习非常重要的功能。以下展示了如何使用 TensorFlow 计算线性回归。可以注意到，程序的结构和前述 NumPy 的实现非常类似。这里，TensorFlow 帮助我们做了两件重要的工作：

- 使用 `tape.gradient(ys, xs)` 自动计算梯度；
- 使用 `optimizer.apply_gradients(grads_and_vars)` 自动更新模型参数。

```
X = tf.constant(X)
y = tf.constant(y)
```

[4] 此处的损失函数为均方差 $L(x) = \frac{1}{2}\sum_{i=1}^{5}(ax_i + b - y_i)^2$。其关于参数 a 和 b 的偏导数为 $\frac{\partial L}{\partial a} = \sum_{i=1}^{5}(ax_i + b - y)x_i$，$\frac{\partial L}{\partial b} = \sum_{i=1}^{5}(ax_i + b - y)$

[5] 与 Eager Execution 相对的是 Graph Execution（静态图）模式，即 TensorFlow 在 2018 年 3 月的 1.8 版本发布之前所主要使用的模式。本手册以面向快速迭代开发的动态模式为主，但会在附录中介绍静态图模式的基本使用，供需要的读者查阅。

```
a = tf.get_variable('a', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
b = tf.get_variable('b', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
variables = [a, b]

num_epoch = 10000
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1e-3)
for e in range(num_epoch):
    # 使用 tf.GradientTape() 记录损失函数的梯度信息
    with tf.GradientTape() as tape:
        y_pred = a * X + b
        loss = 0.5 * tf.reduce_sum(tf.square(y_pred - y))
    # TensorFlow 自动计算损失函数关于自变量（模型参数）的梯度
    grads = tape.gradient(loss, variables)
    # TensorFlow 自动根据梯度更新参数
    optimizer.apply_gradients(grads_and_vars=zip(grads, variables))
```

在这里，我们使用了前文的方式计算了损失函数关于参数的偏导数。同时，使用 `tf.train.GradientDescentOptimizer(learning_rate=1e-3)` 声明了一个梯度下降 **优化器**（Optimizer），其学习率为 1e-3。优化器可以帮助我们根据计算出的求导结果更新模型参数，从而最小化某个特定的损失函数，具体使用方式是调用其 `apply_gradients()` 方法。

注意到这里，更新模型参数的方法 `optimizer.apply_gradients()` 需要提供参数 `grads_and_vars`，即待更新的变量（如上述代码中的 `variables` ）及损失函数关于这些变量的偏导数（如上述代码中的 `grads` ）。具体而言，这里需要传入一个 Python 列表（List），列表中的每个元素是一个（变量的偏导数，变量）对。比如这里是 `[(grad_w, w), (grad_b, b)]` 。我们通过 `grads = tape.gradient(loss, variables)` 求出 tape 中记录的 `loss` 关于 `variables = [w, b]` 中每个变量的偏导数，也就是 `grads = [grad_w, grad_b]`，再使用 Python 的 `zip()` 函数将 `grads = [grad_w, grad_b]` 和 `vars = [w, b]` 拼装在一起，就可以组合出所需的参数了。

在实际应用中，我们编写的模型往往比这里一行就能写完的线性模型 `y_pred = tf.matmul(X, w) + b` 要复杂得多。所以，我们往往会编写一个模型类，然后在需要调用的时候使用 `y_pred = model(X)` 进行调用。关于模型类的编写方式可见下章。

CHAPTER 4

TensorFlow 模型

本章介绍如何使用 TensorFlow 快速搭建动态模型。

前置知识：

- Python 面向对象 （在 Python 内定义类和方法、类的继承、构造和析构函数，使用 super() 函数调用父类方法 ，使用 ___call___() 方法对实例进行调用 等）；
- 多层感知机、卷积神经网络、循环神经网络和强化学习（每节之前给出参考资料）。

4.1 模型（Model）与层（Layer）

如上一章所述，为了增强代码的可复用性，我们往往会将模型编写为类，然后在模型调用的地方使用 `y_pred = model(X)` 的形式进行调用。**模型类**的形式非常简单，主要包含 `__init__()` （构造函数，初始化）和 `call(input)` （模型调用）两个方法，但也可以根据需要增加自定义的方法。[1]

```
class MyModel(tf.keras.Model):
    def __init__(self):
        super().__init__()     # Python 2 下使用 super(MyModel, self).__init__()
        # 此处添加初始化代码（包含 call 方法中会用到的层）

    def call(self, inputs):
```

[1] 在 Python 类中，对类的实例 `myClass` 进行形如 `myClass()` 的调用等价于 `myClass.__call__()` 。在这里，我们的模型继承了 `tf.keras.Model` 这一父类。该父类中包含 `__call__()` 的定义，其中调用了 `call()` 方法，同时进行了一些 keras 的内部操作。这里，我们通过继承 `tf.keras.Model` 并重载 `call()` 方法，即可在保持 keras 结构的同时加入模型调用的代码。具体请见本章初 “前置知识” 的 `__call__()` 部分。

```
        # 此处添加模型调用的代码（处理输入并返回输出）
        return output
```

在这里，我们的模型类继承了 `tf.keras.Model` 。Keras 是一个用 Python 编写的高级神经网络 API，现已得到 TensorFlow 的官方支持和内置。继承 `tf.keras.Model` 的一个好处在于我们可以使用父类的若干方法和属性，例如在实例化类后可以通过 `model.variables` 这一属性直接获得模型中的所有变量，免去我们一个个显式指定变量的麻烦。

同时，我们引入 **“层”（Layer）** 的概念，层可以视为比模型粒度更细的组件单位，将计算流程和变量进行了封装。我们可以使用层来快速搭建模型。

上一章中简单的线性模型 `y_pred = tf.matmul(X, w) + b` ，我们可以通过模型类的方式编写如下：

```
import tensorflow as tf
tf.enable_eager_execution()

X = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
y = tf.constant([[10.0], [20.0]])


class Linear(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.dense = tf.keras.layers.Dense(units=1, kernel_initializer=tf.zeros_initializer(),
            bias_initializer=tf.zeros_initializer())

    def call(self, input):
        output = self.dense(input)
        return output


# 以下代码结构与前节类似
model = Linear()
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
for i in range(100):
    with tf.GradientTape() as tape:
        y_pred = model(X)       # 调用模型
        loss = tf.reduce_mean(tf.square(y_pred - y))
    grads = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
print(model.variables)
```

这里，我们没有显式地声明 `w` 和 `b` 两个变量并写出 `y_pred = tf.matmul(X, w) + b` 这一线性变换，而是在初始化部分实例化了一个全连接层（`tf.keras.layers.Dense` ），并在 call 方法中对这个层进行调用。全连接层封装了 `output = activation(tf.matmul(input, kernel) + bias)` 这一线性变换 + 激活函数的

计算操作，以及 kernel 和 bias 两个变量。当不指定激活函数时（即 activation(x) = x），这个全连接层就等价于我们上述的线性变换。顺便一提，全连接层可能是我们编写模型时使用最频繁的层。

如果我们需要显式地声明自己的变量并使用变量进行自定义运算，请参考自定义层。

4.2 基础示例：多层感知机（MLP）

我们从编写一个最简单的多层感知机（Multilayer Perceptron, MLP）开始，介绍 TensorFlow 的模型编写方式。这里，我们使用多层感知机完成 MNIST 手写体数字图片数据集 *[LeCun1998]* 的分类任务。

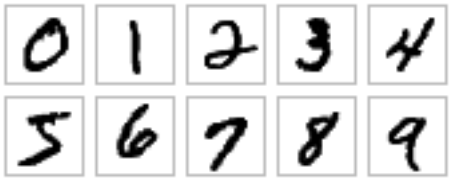


图 4.1: MNIST 手写体数字图片示例

先进行预备工作，实现一个简单的 DataLoader 类来读取 MNIST 数据集数据。

```
class DataLoader():
    def __init__(self):
        mnist = tf.contrib.learn.datasets.load_dataset("mnist")
        self.train_data = mnist.train.images                                    # np.array [55000,␣
↪784]
        self.train_labels = np.asarray(mnist.train.labels, dtype=np.int32)   # np.array [55000] of␣
↪int32
        self.eval_data = mnist.test.images                                      # np.array [10000,␣
↪784]
        self.eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)     # np.array [10000] of␣
↪int32

    def get_batch(self, batch_size):
        index = np.random.randint(0, np.shape(self.train_data)[0], batch_size)
        return self.train_data[index, :], self.train_labels[index]
```

多层感知机的模型类实现与上面的线性模型类似，所不同的地方在于层数增加了（顾名思义，“多层”感知机），以及引入了非线性激活函数（这里使用了 ReLU 函数，即下方的 activation=tf.nn.relu）。该模型输入一个向量（比如这里是拉直的 1×784 手写体数字图片），输出 10 维的信号，分别代表这张图片属于 0 到 9 的概率。这里我们加入了一个 predict 方法，对图片对应的数字进行预测。在预测的时候，选择概率最大的数字进行预测输出。

```
class MLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
```

```
        self.dense1 = tf.keras.layers.Dense(units=100, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(units=10)

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
        return x

    def predict(self, inputs):
        logits = self(inputs)
        return tf.argmax(logits, axis=-1)
```

定义一些模型超参数：

```
num_batches = 1000
batch_size = 50
learning_rate = 0.001
```

实例化模型，数据读取类和优化器：

```
model = MLP()
data_loader = DataLoader()
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

然后迭代进行以下步骤：

- 从 DataLoader 中随机取一批训练数据；
- 将这批数据送入模型，计算出模型的预测值；
- 将模型预测值与真实值进行比较，计算损失函数（loss）；
- 计算损失函数关于模型变量的导数；
- 使用优化器更新模型参数以最小化损失函数。

具体代码实现如下：

```
for batch_index in range(num_batches):
    X, y = data_loader.get_batch(batch_size)
    with tf.GradientTape() as tape:
        y_logit_pred = model(tf.convert_to_tensor(X))
        loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
        print("batch %d: loss %f" % (batch_index, loss.numpy()))
    grads = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
```

接下来，我们使用验证集测试模型性能。具体而言，比较验证集上模型预测的结果与真实结果，输出预测正确的样本数占总样本数的比例：

```
num_eval_samples = np.shape(data_loader.eval_labels)[0]
y_pred = model.predict(data_loader.eval_data).numpy()
print("test accuracy: %f" % (sum(y_pred == data_loader.eval_labels) / num_eval_samples))
```

输出结果：

```
test accuracy: 0.947900
```

可以注意到，使用这样简单的模型，已经可以达到 95% 左右的准确率。

4.3 卷积神经网络（CNN）

卷积神经网络（Convolutional Neural Network, CNN）是一种结构类似于人类或动物的视觉系统的人工神经网络，包含一个或多个卷积层（Convolutional Layer）、池化层（Pooling Layer）和全连接层（Dense Layer）。具体原理建议可以参考台湾大学李宏毅教授的《机器学习》课程的 Convolutional Neural Network 一章。

具体的实现见下，和 MLP 很类似，只是新加入了一些卷积层和池化层。

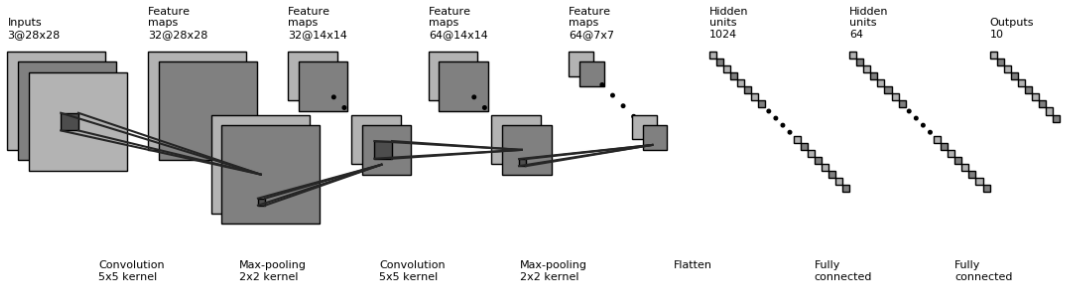


图 4.2: CNN 结构图示

```
class CNN(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.conv1 = tf.keras.layers.Conv2D(
            filters=32,             # 卷积核数目
            kernel_size=[5, 5],     # 感受野大小
            padding="same",         # padding 策略
            activation=tf.nn.relu   # 激活函数
        )
        self.pool1 = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2)
        self.conv2 = tf.keras.layers.Conv2D(
            filters=64,
```

```
            kernel_size=[5, 5],
            padding="same",
            activation=tf.nn.relu
        )
        self.pool2 = tf.keras.layers.MaxPool2D(pool_size=[2, 2], strides=2)
        self.flatten = tf.keras.layers.Reshape(target_shape=(7 * 7 * 64,))
        self.dense1 = tf.keras.layers.Dense(units=1024, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(units=10)

    def call(self, inputs):
        inputs = tf.reshape(inputs, [-1, 28, 28, 1])
        x = self.conv1(inputs)                  # [batch_size, 28, 28, 32]
        x = self.pool1(x)                       # [batch_size, 14, 14, 32]
        x = self.conv2(x)                       # [batch_size, 14, 14, 64]
        x = self.pool2(x)                       # [batch_size, 7, 7, 64]
        x = self.flatten(x)                     # [batch_size, 7 * 7 * 64]
        x = self.dense1(x)                      # [batch_size, 1024]
        x = self.dense2(x)                      # [batch_size, 10]
        return x

    def predict(self, inputs):
        logits = self(inputs)
```

将前节的 `model = MLP()` 更换成 `model = CNN()`，输出如下：

```
test accuracy: 0.988100
```

可以发现准确率有非常显著的提高。事实上，通过改变模型的网络结构（比如加入 Dropout 层防止过拟合），准确率还有进一步提升的空间。

4.4 循环神经网络（RNN）

循环神经网络（Recurrent Neural Network, RNN）是一种适宜于处理序列数据的神经网络，被广泛用于语言模型、文本生成、机器翻译等。关于 RNN 的原理，可以参考：

- Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs
- 台湾大学李宏毅教授的《机器学习》课程的 Recurrent Neural Network (part 1) Recurrent Neural Network (part 2) 两部分。
- LSTM 原理：Understanding LSTM Networks
- RNN 序列生成：*[Graves2013]*

这里，我们使用 RNN 来进行尼采风格文本的自动生成。[2]

这个任务的本质其实预测一段英文文本的接续字母的概率分布。比如，我们有以下句子:

```
I am a studen
```

这个句子（序列）一共有 13 个字符（包含空格）。当我们阅读到这个由 13 个字符组成的序列后，根据我们的经验，我们可以预测出下一个字符很大概率是 “t”。我们希望建立这样一个模型，输入 num_batch 个由编码后字符组成的，长为 seq_length 的序列，输入张量形状为 [num_batch, seq_length]，输出这些序列接续的下一个字符的概率分布，概率分布的维度为字符种类数 num_chars，输出张量形状为 [num_batch, num_chars]。我们从下一个字符的概率分布中采样作为预测值，然后滚雪球式地生成下两个字符，下三个字符等等，即可完成文本的生成任务。

首先，还是实现一个简单的 DataLoader 类来读取文本，并以字符为单位进行编码。

```
class DataLoader():
    def __init__(self):
        path = tf.keras.utils.get_file('nietzsche.txt',
            origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
        with open(path, encoding='utf-8') as f:
            self.raw_text = f.read().lower()
        self.chars = sorted(list(set(self.raw_text)))
        self.char_indices = dict((c, i) for i, c in enumerate(self.chars))
        self.indices_char = dict((i, c) for i, c in enumerate(self.chars))
        self.text = [self.char_indices[c] for c in self.raw_text]

    def get_batch(self, seq_length, batch_size):
        seq = []
        next_char = []
        for i in range(batch_size):
            index = np.random.randint(0, len(self.text) - seq_length)
            seq.append(self.text[index:index+seq_length])
            next_char.append(self.text[index+seq_length])
        return np.array(seq), np.array(next_char)       # [num_batch, seq_length], [num_batch]
```

接下来进行模型的实现。在 __init__ 方法中我们实例化一个常用的 BasicLSTMCell 单元，以及一个线性变换用的全连接层，我们首先对序列进行 One Hot 操作，即将编码 i 变换为一个 n 维向量，其第 i 位为 1，其余均为 0。这里 n 为字符种类数 num_char。变换后的序列张量形状为 [num_batch, seq_length, num_chars]。接下来，我们将序列从头到尾依序送入 RNN 单元，即将当前时间 t 的 RNN 单元状态 state 和 t 时刻的序列 inputs[:, t, :] 送入 RNN 单元，得到当前时间的输出 output 和下一个时间 t+1 的 RNN 单元状态。取 RNN 单元最后一次的输出，通过全连接层变换到 num_chars 维，即作为模型的输出。

具体实现如下:

[2] 此处的任务及实现参考了 https://github.com/keras-team/keras/blob/master/examples/lstm_text_generation.py

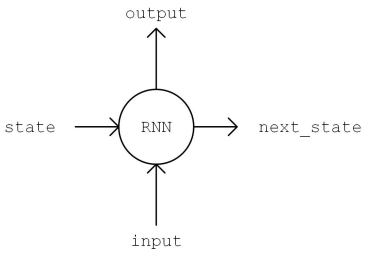


图 4.3: `output, state = self.cell(inputs[:, t, :], state)` 图示

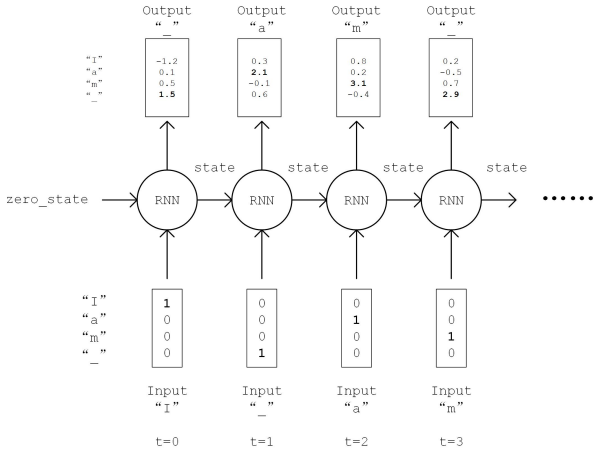


图 4.4: RNN 流程图示

```
class RNN(tf.keras.Model):
    def __init__(self, num_chars):
        super().__init__()
        self.num_chars = num_chars
        self.cell = tf.nn.rnn_cell.BasicLSTMCell(num_units=256)
        self.dense = tf.keras.layers.Dense(units=self.num_chars)

    def call(self, inputs):
        batch_size, seq_length = tf.shape(inputs)
        inputs = tf.one_hot(inputs, depth=self.num_chars)       # [batch_size, seq_length, num_
↪chars]
        state = self.cell.zero_state(batch_size=batch_size, dtype=tf.float32)
        for t in range(seq_length.numpy()):
            output, state = self.cell(inputs[:, t, :], state)
        output = self.dense(output)
        return output
```

训练过程与前节基本一致，在此复述：

- 从 DataLoader 中随机取一批训练数据；
- 将这批数据送入模型，计算出模型的预测值；
- 将模型预测值与真实值进行比较，计算损失函数（loss）；
- 计算损失函数关于模型变量的导数；
- 使用优化器更新模型参数以最小化损失函数。

```
data_loader = DataLoader()
model = RNN(len(data_loader.chars))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
for batch_index in range(num_batches):
    X, y = data_loader.get_batch(seq_length, batch_size)
    with tf.GradientTape() as tape:
        y_logit_pred = model(X)
        loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
        print("batch %d: loss %f" % (batch_index, loss.numpy()))
    grads = tape.gradient(loss, model.variables)
    optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
```

关于文本生成的过程有一点需要特别注意。之前，我们一直使用 `tf.argmax()` 函数，将对应概率最大的值作为预测值。然而对于文本生成而言，这样的预测方式过于绝对，会使得生成的文本失去丰富性。于是，我们使用 `np.random.choice()` 函数按照生成的概率分布取样。这样，即使是对应概率较小的字符，也有机会被取样到。同时，我们加入一个 `temperature` 参数控制分布的形状，参数值越大则分布越平缓（最大值和最小值的差值越小），生成文本的丰富度越高；参数值越小则分布越陡峭，生成文本的丰富度越低。

```
    def predict(self, inputs, temperature=1.):
        batch_size, _ = tf.shape(inputs)
        logits = self(inputs)
        prob = tf.nn.softmax(logits / temperature).numpy()
        return np.array([np.random.choice(self.num_chars, p=prob[i, :])
                         for i in range(batch_size.numpy())])
```

通过这种方式进行“滚雪球”式的连续预测，即可得到生成文本。

```
X_, _ = data_loader.get_batch(seq_length, 1)
for diversity in [0.2, 0.5, 1.0, 1.2]:
    X = X_
    print("diversity %f:" % diversity)
    for t in range(400):
        y_pred = model.predict(X, diversity)
        print(data_loader.indices_char[y_pred[0]], end='', flush=True)
        X = np.concatenate([X[:, 1:], np.expand_dims(y_pred, axis=1)], axis=-1)
```

生成的文本如下:

```
diversity 0.200000:
conserted and conseive to the conterned to it is a self--and seast and the selfes as a seast the␣
↪expecience and and and the self--and the sered is a the enderself and the sersed and as a the␣
↪concertion of the series of the self in the self--and the serse and and the seried enes and␣
↪seast and the sense and the eadure to the self and the present and as a to the self--and the␣
↪seligious and the enders

diversity 0.500000:
can is reast to as a seligut and the complesed
has fool which the self as it is a the beasing and us immery and seese for entoured underself of␣
↪the seless and the sired a mears and everyther to out every sone thes and reapres and seralise␣
↪as a streed liees of the serse to pease the cersess of the selung the elie one of the were as we␣
↪and man one were perser has persines and conceity of all self-el

diversity 1.000000:
entoles by
their lisevers de weltaale, arh pesylmered, and so jejurted count have foursies as is
descinty iamo; to semplization refold, we dancey or theicks-welf--atolitious on his
such which
here
oth idey of pire master, ie gerw their endwit in ids, is an trees constenved mase commars is leed␣
↪mad decemshime to the mor the elige. the fedies (byun their ope wopperfitious--antile and the it␣
↪as the f

diversity 1.200000:
```

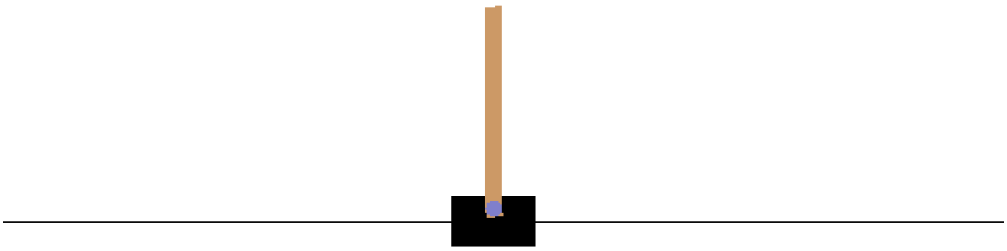
```
cain, elvotidue, madehoublesily
inselfy!--ie the rads incults of to prusely le]enfes patuateded:.--a coud--theiritibaior
↪"nrallysengleswout peessparify oonsgoscess teemind thenry ansken suprerial mus, cigitioum: 4reas.
↪ whouph: who
eved
arn inneves to sya" natorne. hag open reals whicame oderedte,[fingo is
zisternethta simalfule dereeg hesls lang-lyes thas quiin turjentimy; periaspedey tomm--whach
```

4.5 深度强化学习（DRL）

强化学习 （Reinforcement learning，RL）强调如何基于环境而行动，以取得最大化的预期利益。结合了深度学习技术后的强化学习更是如虎添翼。这两年广为人知的 AlphaGo 即是深度强化学习的典型应用。深度强化学习的基础知识可参考：

- Demystifying Deep Reinforcement Learning （中文编译）
- *[Mnih2013]*

这里，我们使用深度强化学习玩 CartPole（平衡杆）游戏。简单说，我们需要让模型控制杆的左右运动，以让其一直保持竖直平衡状态。



```
episode 112, epsilon 0.010000, score 358
episode 113, epsilon 0.010000, score 158
episode 114, epsilon 0.010000, score 156
```

图 4.5: CartPole 游戏

我们使用 OpenAI 推出的 Gym 环境库 中的 CartPole 游戏环境，具体安装步骤和教程可参考 官方文档 和 这里 。Gym 的基本调用方法如下：

```
import gym
```

```
env = gym.make('CartPole-v1')        # 实例化一个游戏环境，参数为游戏名称
state = env.reset()                  # 初始化环境，获得初始状态
while True:
    env.render()                     # 对当前帧进行渲染，绘图到屏幕
    action = model.predict(state)    # 假设我们有一个训练好的模型，能够通过当前状态预测出这时应该进行的动作
    next_state, reward, done, info = env.step(action)   # 让环境执行动作，获得执行完动作的下一个状态，动作的奖励，游戏是否已结束以及额外信息
    if done:                         # 如果游戏结束则退出循环
        break
```

那么，我们的任务就是训练出一个模型，能够根据当前的状态预测出应该进行的一个好的动作。粗略地说，一个好的动作应当能够最大化整个游戏过程中获得的奖励之和，这也是强化学习的目标。

以下代码展示了如何使用深度强化学习中的 Deep Q-Learning 方法来训练模型。

```
import tensorflow as tf
import numpy as np
import gym
import random
from collections import deque

tf.enable_eager_execution()
num_episodes = 500
num_exploration_episodes = 100
max_len_episode = 1000
batch_size = 32
learning_rate = 1e-3
gamma = 1.
initial_epsilon = 1.
final_epsilon = 0.01


# Q-network 用于拟合 Q 函数，和前节的多层感知机类似。输入 state，输出各个 action 下的 Q-value（CartPole 下为 2 维）。
class QNetwork(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.dense1 = tf.keras.layers.Dense(units=24, activation=tf.nn.relu)
        self.dense2 = tf.keras.layers.Dense(units=24, activation=tf.nn.relu)
        self.dense3 = tf.keras.layers.Dense(units=2)

    def call(self, inputs):
        x = self.dense1(inputs)
        x = self.dense2(x)
```

```
        x = self.dense3(x)
        return x

    def predict(self, inputs):
        q_values = self(inputs)
        return tf.argmax(q_values, axis=-1)


env = gym.make('CartPole-v1')       # 实例化一个游戏环境，参数为游戏名称
model = QNetwork()
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
replay_buffer = deque(maxlen=10000)
epsilon = initial_epsilon
for episode_id in range(num_episodes):
    state = env.reset()             # 初始化环境，获得初始状态
    epsilon = max(
        initial_epsilon * (num_exploration_episodes - episode_id) / num_exploration_episodes,
        final_epsilon)
    for t in range(max_len_episode):
        env.render()                # 对当前帧进行渲染，绘图到屏幕
        if random.random() < epsilon:               # epsilon-greedy 探索策略
            action = env.action_space.sample()      # 以 epsilon 的概率选择随机动作
        else:
            action = model.predict(
                tf.constant(np.expand_dims(state, axis=0), dtype=tf.float32)).numpy()
            action = action[0]
        next_state, reward, done, info = env.step(action)               # 让环境执行动作，获得执行完
动作的下一个状态，动作的奖励，游戏是否已结束以及额外信息
        reward = -10. if done else reward                               # 如果游戏 Game Over，给予大
的负奖励
        replay_buffer.append((state, action, reward, next_state, done)) # 将 (state, action,␣
↪reward, next_state) 的四元组（外加 done 标签表示是否结束）放入经验重放池
        state = next_state

        if done:                                                        # 游戏结束则退出本轮循环，进
行下一个 episode
            print("episode %d, epsilon %f, score %d" % (episode_id, epsilon, t))
            break

        if len(replay_buffer) >= batch_size:
            batch_state, batch_action, batch_reward, batch_next_state, batch_done = \
                [np.array(a, dtype=np.float32) for a in zip(*random.sample(replay_buffer, batch_
↪size))] # 从经验回放池中随机取一个 batch 的四元组
            q_value = model(tf.constant(batch_next_state, dtype=tf.float32))
            y = batch_reward + (gamma * tf.reduce_max(q_value, axis=1)) * (1 - batch_done)  # 按照
论文计算 y 值
```

```
            with tf.GradientTape() as tape:
                loss = tf.losses.mean_squared_error(        # 最小化 y 和 Q-value 的距离
                    labels=y,
                    predictions=tf.reduce_sum(model(tf.constant(batch_state)) *
                                              tf.one_hot(batch_action, depth=2), axis=1)
                )
            grads = tape.gradient(loss, model.variables)
            optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))       # 计算梯度
并更新参数
```

4.6 自定义层 *

可能你还会问，如果现有的这些层无法满足我的要求，我需要定义自己的层怎么办?

事实上，我们不仅可以继承 tf.keras.Model 编写自己的模型类，也可以继承 tf.keras.layers.Layer 编写自己的层。

```
class MyLayer(tf.keras.layers.Layer):
    def __init__(self):
        super().__init__()
        # 初始化代码

    def build(self, input_shape):     # input_shape 是一个 TensorShape 类型对象，提供输入的形状
        # 在第一次使用该层的时候调用该部分代码，在这里创建变量可以使得变量的形状自适应输入的形状
        # 而不需要使用者额外指定变量形状。
        # 如果已经可以完全确定变量的形状，也可以在__init__ 部分创建变量
        self.variable_0 = self.add_variable(...)
        self.variable_1 = self.add_variable(...)

    def call(self, input):
        # 模型调用的代码（处理输入并返回输出）
        return output
```

例如，如果我们要自己实现一个本章第一节 中的全连接层，但指定输出维度为 1，可以按如下方式编写，在 build 方法中创建两个变量，并在 call 方法中使用创建的变量进行运算：

```
class LinearLayer(tf.keras.layers.Layer):
    def __init__(self):
        super().__init__()

    def build(self, input_shape):     # here input_shape is a TensorShape
        self.w = self.add_variable(name='w',
            shape=[input_shape[-1], 1], initializer=tf.zeros_initializer())
```

```
        self.b = self.add_variable(name='b',
            shape=[1], initializer=tf.zeros_initializer())

    def call(self, X):
        y_pred = tf.matmul(X, self.w) + self.b
        return y_pred
```

使用相同的方式，可以调用我们自定义的层 LinearLayer：

```
class Linear(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.layer = LinearLayer()

    def call(self, input):
        output = self.layer(input)
        return output
```

4.7 Graph Execution 模式 *

事实上，只要在编写模型的时候稍加注意，以上的模型都是可以同时兼容 Eager Execution 模式和 Graph Execution 模式的[3] 。注意，在 Graph Execution 模式下，`model(input_tensor)` 只需运行一次以完成图的建立操作。

例如，通过以下代码，同样可以调用本章第一节 建立的线性模型并进行线性回归：

```
    model = Linear()
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
    X_placeholder = tf.placeholder(name='X', shape=[None, 3], dtype=tf.float32)
    y_placeholder = tf.placeholder(name='y', shape=[None, 1], dtype=tf.float32)
    y_pred = model(X_placeholder)
    loss = tf.reduce_mean(tf.square(y_pred - y_placeholder))
    train_op = optimizer.minimize(loss)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for i in range(100):
            sess.run(train_op, feed_dict={X_placeholder: X, y_placeholder: y})
        print(sess.run(model.variables))
```

[3] 除了本章实现的 RNN 模型以外。在 RNN 模型的实现中，我们通过 Eager Execution 动态获取了 seq_length 的长度，使得我们可以方便地动态控制 RNN 的展开长度。然而 Graph Execution 不支持这一点，为了达到相同的效果，我们需要固定 seq_length 的长度，或者使用 `tf.nn.dynamic_rnn` （文档）。

CHAPTER 5

TensorFlow 扩展

本章介绍一些最为常用的 TensorFlow 扩展功能。虽然这些功能称不上“必须”，但能让模型训练和调用的过程更加方便。

前置知识：

- Python 的序列化模块 Pickle （非必须）
- Python 的特殊函数参数 **kwargs （非必须）

5.1 Checkpoint：变量的保存与恢复

很多时候，我们希望在模型训练完成后能将训练好的参数（变量）保存起来。在需要使用模型的其他地方载入模型和参数，就能直接得到训练好的模型。可能你第一个想到的是用 Python 的序列化模块 `pickle` 存储 `model.variables`。但不幸的是，TensorFlow 的变量类型 `ResourceVariable` 并不能被序列化。

好在 TensorFlow 提供了 tf.train.Checkpoint 这一强大的变量保存与恢复类，可以使用其 `save()` 和 `restore()` 方法将 TensorFlow 中所有包含 Checkpointable State 的对象进行保存和恢复。具体而言，`tf.train.Optimizer` 实现, `tf.Variable`, `tf.keras.Layer` 实现或者 `tf.keras.Model` 实现都可以被保存。其使用方法非常简单，我们首先声明一个 Checkpoint：

```
checkpoint = tf.train.Checkpoint(model=model)
```

这里 `tf.train.Checkpoint()` 接受的初始化参数比较特殊，是一个 `**kwargs` 。具体而言，是一系列的键值对，键名可以随意取，值为需要保存的对象。例如，如果我们希望保存一个继承 `tf.keras.Model` 的模型实例 `model` 和一个继承 `tf.train.Optimizer` 的优化器 `optimizer` ，我们可以这样写：

```
checkpoint = tf.train.Checkpoint(myAwesomeModel=model, myAwesomeOptimizer=optimizer)
```

这里 myAwesomeModel 是我们为待保存的模型 model 所取的任意键名。注意，在恢复变量的时候，我们还将使用这一键名。

接下来，当模型训练完成需要保存的时候，使用：

```
checkpoint.save(save_path_with_prefix)
```

就可以。save_path_with_prefix 是保存文件的目录 + 前缀。例如，在源代码目录建立一个名为 save 的文件夹并调用一次 checkpoint.save('./save/model.ckpt')，我们就可以在可以在 save 目录下发现名为 checkpoint 、model.ckpt-1.index 、model.ckpt-1.data-00000-of-00001 的三个文件，这些文件就记录了变量信息。checkpoint.save() 方法可以运行多次，每运行一次都会得到一个.index 文件和.data 文件，序号依次累加。

当在其他地方需要为模型重新载入之前保存的参数时，需要再次实例化一个 checkpoint，同时保持键名的一致。再调用 checkpoint 的 restore 方法。就像下面这样：

```
model_to_be_restored = MyModel()                                    # 待恢复参数的同一模型
checkpoint = tf.train.Checkpoint(myAwesomeModel=model_to_be_restored)    # 键名保持为
“myAwesomeModel”
checkpoint.restore(save_path_with_prefix_and_index)
```

即可恢复模型变量。save_path_with_prefix_and_index 是之前保存的文件的目录 + 前缀 + 编号。例如，调用 checkpoint.restore('./save/model.ckpt-1') 就可以载入前缀为 model.ckpt，序号为 1 的文件来恢复模型。

当保存了多个文件时，我们往往想载入最近的一个。可以使用 tf.train.latest_checkpoint(save_path) 这个辅助函数返回目录下最近一次 checkpoint 的文件名。例如如果 save 目录下有 model.ckpt-1.index 到 model.ckpt-10.index 的 10 个保存文件，tf.train.latest_checkpoint('./save') 即返回 ./save/model.ckpt-10 。

总体而言，恢复与保存变量的典型代码框架如下：

```
# train.py 模型训练阶段

model = MyModel()
checkpoint = tf.train.Checkpoint(myModel=model)      # 实例化 Checkpoint，指定保存对象为 model（如果需
要保存 Optimizer 的参数也可加入）
# 模型训练代码
checkpoint.save('./save/model.ckpt')                 # 模型训练完毕后将参数保存到文件，也可以在模型训
练过程中每隔一段时间就保存一次
```

```
# test.py 模型使用阶段

model = MyModel()
```

```
checkpoint = tf.train.Checkpoint(myModel=model)              # 实例化 Checkpoint，指定恢复对象为 model
checkpoint.restore(tf.train.latest_checkpoint('./save'))     # 从文件恢复模型参数
# 模型使用代码
```

顺便一提，`tf.train.Checkpoint` 与以前版本常用的 `tf.train.Saver` 相比，强大之处在于其支持在 Eager Execution 下“延迟”恢复变量。具体而言，当调用了 `checkpoint.restore()`，但模型中的变量还没有被建立的时候，Checkpoint 可以等到变量被建立的时候再进行数值的恢复。Eager Execution 下，模型中各个层的初始化和变量的建立是在模型第一次被调用的时候才进行的（好处在于可以根据输入的张量形状而自动确定变量形状，无需手动指定）。这意味着当模型刚刚被实例化的时候，其实里面还一个变量都没有，这时候使用以往的方式去恢复变量数值是一定会报错的。比如，你可以试试在 train.py 调用 `tf.keras.Model` 的 `save_weight()` 方法保存 model 的参数，并在 test.py 中实例化 model 后立即调用 `load_weight()` 方法，就会出错，只有当调用了一遍 model 之后再运行 `load_weight()` 方法才能得到正确的结果。可见，`tf.train.Checkpoint` 在这种情况下可以给我们带来相当大的便利。另外，`tf.train.Checkpoint` 同时也支持 Graph Execution 模式。

最后提供一个实例，以前章的*多层感知机模型* 为例展示模型变量的保存和载入：

```
import tensorflow as tf
import numpy as np
from zh.model.mlp.mlp import MLP
from zh.model.mlp.utils import DataLoader

tf.enable_eager_execution()
mode = 'test'
num_batches = 1000
batch_size = 50
learning_rate = 0.001
data_loader = DataLoader()


def train():
    model = MLP()
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    checkpoint = tf.train.Checkpoint(myAwesomeModel=model)      # 实例化 Checkpoint，设置保存对象为 model
    for batch_index in range(num_batches):
        X, y = data_loader.get_batch(batch_size)
        with tf.GradientTape() as tape:
            y_logit_pred = model(tf.convert_to_tensor(X))
            loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
            print("batch %d: loss %f" % (batch_index, loss.numpy()))
        grads = tape.gradient(loss, model.variables)
        optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
        if (batch_index + 1) % 100 == 0:                        # 每隔 100 个 Batch 保存一次
```

```
            checkpoint.save('./save/model.ckpt')                # 保存模型参数到文件


def test():
    model_to_be_restored = MLP()
    checkpoint = tf.train.Checkpoint(myAwesomeModel=model_to_be_restored)      # 实例化 Checkpoint，
设置恢复对象为新建立的模型 model_to_be_restored
    checkpoint.restore(tf.train.latest_checkpoint('./save'))    # 从文件恢复模型参数
    num_eval_samples = np.shape(data_loader.eval_labels)[0]
    y_pred = model_to_be_restored.predict(tf.constant(data_loader.eval_data)).numpy()
    print("test accuracy: %f" % (sum(y_pred == data_loader.eval_labels) / num_eval_samples))


if __name__ == '__main__':
    if mode == 'train':
        train()
    if mode == 'test':
        test()
```

在代码目录下建立 save 文件夹并运行代码进行训练后，save 文件夹内将会存放每隔 100 个 batch 保存一次的模型变量数据。将第 7 行改为 `model = 'test'` 并再次运行代码，将直接使用最后一次保存的变量值恢复模型并在测试集上测试模型性能，可以直接获得 95% 左右的准确率。

5.2 TensorBoard：训练过程可视化

有时，你希望查看模型训练过程中各个参数的变化情况（例如损失函数 loss 的值）。虽然可以通过命令行输出来查看，但有时显得不够直观。而 TensorBoard 就是一个能够帮助我们将训练过程可视化的工具。

目前，Eager Execution 模式下的 TensorBoard 支持尚在 tf.contrib.summary 内，可能以后会有较多变化，因此这里只做简单示例。首先在代码目录下建立一个文件夹（如./tensorboard）存放 TensorBoard 的记录文件，并在代码中实例化一个记录器：

```
summary_writer = tf.contrib.summary.create_file_writer('./tensorboard')
```

接下来，将训练的代码部分通过 with 语句放在 `summary_writer.as_default()` 和 `tf.contrib.summary.always_record_summaries()` 的上下文中，并对需要记录的参数（一般是 scalar）运行 `tf.contrib.summary.scalar(name, tensor, step=batch_index)` 即可。这里的 step 参数可根据自己的需要自行制定，一般可设置为当前训练过程中的 batch 序号。整体框架如下：

```
summary_writer = tf.contrib.summary.create_file_writer('./tensorboard')
with summary_writer.as_default(), tf.contrib.summary.always_record_summaries():
    # 开始模型训练
    for batch_index in range(num_batches):
```

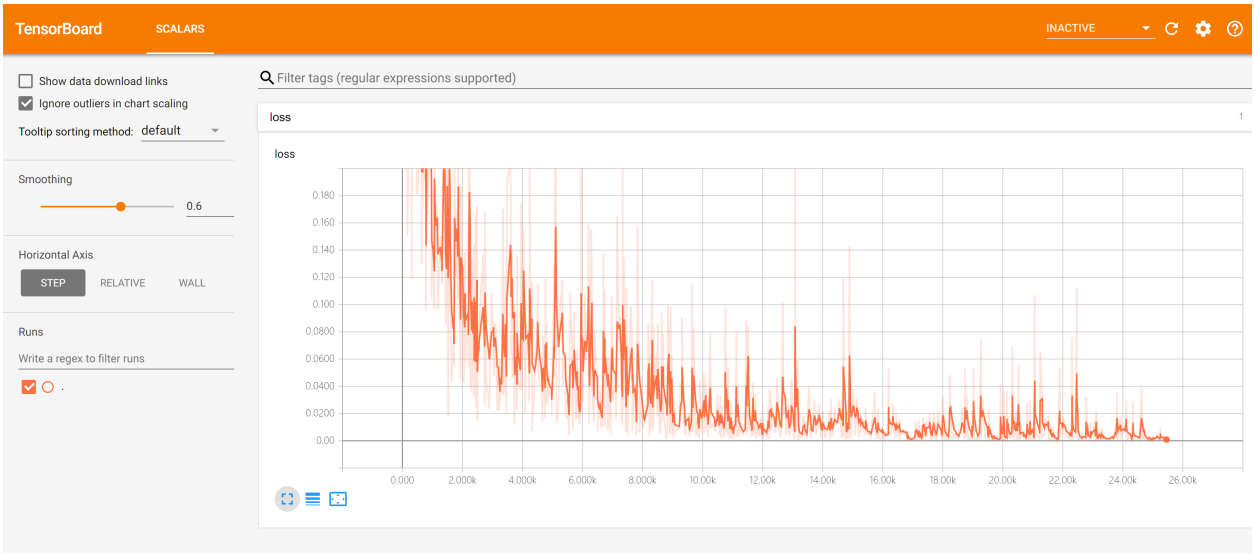
```
        # 训练代码，当前 batch 的损失值放入变量 loss 中
        tf.contrib.summary.scalar("loss", loss, step=batch_index)
        tf.contrib.summary.scalar("MyScalar", my_scalar, step=batch_index)  # 还可以添加其他自定义的
变量
```

每运行一次 `tf.contrib.summary.scalar()`，记录器就会向记录文件中写入一条记录。除了最简单的标量（scalar）以外，TensorBoard 还可以对其他类型的数据（如图像，音频等）进行可视化，详见 API 文档 。

当我们要对训练过程可视化时，在代码目录打开终端（如需要的话进入 TensorFlow 的 conda 环境），运行：

```
tensorboard --logdir=./tensorboard
```

然后使用浏览器访问命令行程序所输出的网址（一般是 http://计算机名称:6006），即可访问 TensorBoard 的可视界面，如下图所示：



默认情况下，TensorBoard 每 30 秒更新一次数据。不过也可以点击右上角的刷新按钮手动刷新。

TensorBoard 的使用有以下注意事项：

- 如果需要重新训练，需要删除掉记录文件夹内的信息并重启 TensorBoard（或者建立一个新的记录文件夹并开启 TensorBoard，`--logdir` 参数设置为新建立的文件夹）；
- 记录文件夹目录保持全英文。

最后提供一个实例，以前章的*多层感知机模型* 为例展示 TensorBoard 的使用：

```
import tensorflow as tf
import numpy as np
from zh.model.mlp.mlp import MLP
from zh.model.mlp.utils import DataLoader

tf.enable_eager_execution()
```

```
num_batches = 10000
batch_size = 50
learning_rate = 0.001
model = MLP()
data_loader = DataLoader()
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
summary_writer = tf.contrib.summary.create_file_writer('./tensorboard')      # 实例化记录器
with summary_writer.as_default(), tf.contrib.summary.always_record_summaries():
    for batch_index in range(num_batches):
        X, y = data_loader.get_batch(batch_size)
        with tf.GradientTape() as tape:
            y_logit_pred = model(tf.convert_to_tensor(X))
            loss = tf.losses.sparse_softmax_cross_entropy(labels=y, logits=y_logit_pred)
            print("batch %d: loss %f" % (batch_index, loss.numpy()))
            tf.contrib.summary.scalar("loss", loss, step=batch_index)       # 记录当前 loss
        grads = tape.gradient(loss, model.variables)
        optimizer.apply_gradients(grads_and_vars=zip(grads, model.variables))
```

5.3 GPU 的使用与分配

很多时候的场景是：实验室/公司研究组里有许多学生/研究员都需要使用 GPU，但多卡的机器只有一台，这时就需要注意如何分配显卡资源。

命令 `nvidia-smi` 可以查看机器上现有的 GPU 及使用情况（在 Windows 下，将 `C:\Program Files\NVIDIA Corporation\NVSMI` 加入 Path 环境变量中即可，或 Windows 10 下可使用任务管理器的“性能”标签查看显卡信息）。

使用环境变量 `CUDA_VISIBLE_DEVICES` 可以控制程序所使用的 GPU。假设发现四卡的机器上显卡 0,1 使用中，显卡 2,3 空闲，Linux 终端输入：

```
export CUDA_VISIBLE_DEVICES=2,3
```

或在代码中加入

```
import os
os.environ['CUDA_VISIBLE_DEVICES'] = "2,3"
```

即可指定程序只在显卡 2,3 上运行。

默认情况下，TensorFlow 将使用几乎所有可用的显存，以避免内存碎片化所带来的性能损失。可以通过 `tf.ConfigProto` 类来设置 TensorFlow 使用显存的策略。具体方式是实例化一个 `tf.ConfigProto` 类，设置参数，并在运行 `tf.enable_eager_execution()` 时指定 Config 参数。以下代码通过 `allow_growth` 选项设置 TensorFlow 仅在需要时申请显存空间：

```
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
tf.enable_eager_execution(config=config)
```

以下代码通过 per_process_gpu_memory_fraction 选项设置 TensorFlow 固定消耗 40% 的 GPU 显存：

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
tf.enable_eager_execution(config=config)
```

Graph Execution 下，也可以在实例化新的 session 时传入 tf.ConfigPhoto 类来进行设置。

CHAPTER 6

附录：静态的 TensorFlow

6.1 TensorFlow 1+1

TensorFlow 本质上是一个符号式的（基于计算图的）计算框架。这里以计算 1+1 作为 Hello World 的示例。

```
import tensorflow as tf

# 定义一个“计算图”
a = tf.constant(1)  # 定义一个常量 Tensor（张量）
b = tf.constant(1)
c = a + b  # 等价于 c = tf.add(a, b)，c 是张量 a 和张量 b 通过 Add 这一 Operation（操作）所形成的新张量

sess = tf.Session()     # 实例化一个 Session（会话）
c_ = sess.run(c)        # 通过 Session 的 run() 方法对计算图里的节点（张量）进行实际的计算
print(c_)
```

输出：

```
2
```

上面这个程序只能计算 1+1，以下程序通过 `tf.placeholder()`（占位符张量）和 `sess.run()` 的 `feed_dict=` 参数展示了如何使用 TensorFlow 计算任意两个数的和：

```
import tensorflow as tf

```

```
a = tf.placeholder(dtype=tf.int32)  # 定义一个占位符 Tensor
b = tf.placeholder(dtype=tf.int32)
c = a + b

a_ = input("a = ")  # 从终端读入一个整数并放入变量 a_
b_ = input("b = ")

sess = tf.Session()
c_ = sess.run(c, feed_dict={a: a_, b: b_})  # feed_dict 参数传入为了计算 c 所需要的张量的值
print("a + b = %d" % c_)
```

运行程序:

```
>>> a = 2
>>> b = 3
a + b = 5
```

**变量 ** （Variable）是一种特殊类型的张量，使用 `tf.get_variable()` 建立，与编程语言中的变量很相似。使用变量前需要先初始化，变量内存储的值可以在计算图的计算过程中被修改。以下示例如何建立一个变量，将其值初始化为 0，并逐次累加 1。

```
import tensorflow as tf

a = tf.get_variable(name='a', shape=[])
initializer = tf.assign(a, 0)   # tf.assign(x, y) 返回一个“将张量 y 的值赋给变量 x”的操作
a_plus_1 = a + 1    # 等价于 a + tf.constant(1)
plus_one_op = tf.assign(a, a_plus_1)

sess = tf.Session()
sess.run(initializer)
for i in range(5):
    sess.run(plus_one_op)                    # 对变量 a 执行加一操作
    a_ = sess.run(a)                         # 获得变量 a 的值并存入 a_
    print(a_)
```

输出:

```
1.0
2.0
3.0
4.0
5.0
```

以下代码和上述代码等价，在声明变量时指定初始化器，并通过 `tf.global_variables_initializer()` 一次性初始化所有变量，在实际工程中更常用：

```
import tensorflow as tf

a = tf.get_variable(name='a', shape=[], initializer=tf.zeros_initializer)   # 指定初始化器为全 0 初始化
a_plus_1 = a + 1
plus_one_op = tf.assign(a, a_plus_1)

sess = tf.Session()
sess.run(tf.global_variables_initializer()) # 初始化所有变量
for i in range(5):
    sess.run(plus_one_op)
    a_ = sess.run(a)
    print(a_)
```

矩阵乃至张量运算是科学计算（包括机器学习）的基本操作。以下程序展示如何计算两个矩阵 $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ 和 $\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$ 的乘积：

```
import tensorflow as tf

A = tf.ones(shape=[2, 3])   # tf.ones(shape) 定义了一个形状为 shape 的全 1 矩阵
B = tf.ones(shape=[3, 2])
C = tf.matmul(A, B)

sess = tf.Session()
C_ = sess.run(C)
print(C_)
```

输出：

```
[[3. 3.]
 [3. 3.]]
```

Placeholder（占位符张量）和 Variable（变量张量）也同样可以为向量、矩阵乃至更高维的张量。

6.2 基础示例：线性回归

与前面的 NumPy 和 Eager Execution 模式不同，TensorFlow 的 Graph Execution 模式使用 **符号式编程** 来进行数值运算。首先，我们需要将待计算的过程抽象为数据流图，将输入、运算和输出都用符号化的节点来表达。然后，我们将数据不断地送入输入节点，让数据沿着数据流图进行计算和流动，最终到达我们需要的特定输出节点。以下代码展示了如何基于 TensorFlow 的符号式编程方法完成与前节相同的任务。其中，

`tf.placeholder()` 即可以视为一种“符号化的输入节点”，使用 `tf.get_variable()` 定义模型的参数（Variable 类型的张量可以使用 `tf.assign()` 进行赋值），而 `sess.run(output_node, feed_dict={input_node: data})` 可以视作将数据送入输入节点，沿着数据流图计算并到达输出节点并返回值的过程。

```
import tensorflow as tf

# 定义数据流图
learning_rate_ = tf.placeholder(dtype=tf.float32)
X_ = tf.placeholder(dtype=tf.float32, shape=[5])
y_ = tf.placeholder(dtype=tf.float32, shape=[5])
a = tf.get_variable('a', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
b = tf.get_variable('b', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)

y_pred = a * X_ + b
loss = tf.constant(0.5) * tf.reduce_sum(tf.square(y_pred - y_))

# 反向传播，手动计算变量（模型参数）的梯度
grad_a = tf.reduce_sum((y_pred - y_) * X_)
grad_b = tf.reduce_sum(y_pred - y_)

# 梯度下降法，手动更新参数
new_a = a - learning_rate_ * grad_a
new_b = b - learning_rate_ * grad_b
update_a = tf.assign(a, new_a)
update_b = tf.assign(b, new_b)

train_op = [update_a, update_b]
# 数据流图定义到此结束
# 注意，直到目前，我们都没有进行任何实质的数据计算，仅仅是定义了一个数据图

num_epoch = 10000
learning_rate = 1e-3
with tf.Session() as sess:
    # 初始化变量 a 和 b
    tf.global_variables_initializer().run()
    # 循环将数据送入上面建立的数据流图中进行计算和更新变量
    for e in range(num_epoch):
        sess.run(train_op, feed_dict={X_: X, y_: y, learning_rate_: learning_rate})
    print(sess.run([a, b]))
```

在上面的两个示例中，我们都是手工计算获得损失函数关于各参数的偏导数。但当模型和损失函数都变得十分复杂时（尤其是深度学习模型），这种手动求导的工程量就难以接受了。TensorFlow 提供了 **自动求导机制**，免去了手工计算导数的繁琐。利用 TensorFlow 的求导函数 `tf.gradients(ys, xs)` 求出损失函数 loss 关于 a，b 的偏导数。由此，我们可以将上节中的两行手工计算导数的代码

```
# 反向传播，手动计算变量（模型参数）的梯度
grad_a = tf.reduce_sum((y_pred - y_) * X_)
grad_b = tf.reduce_sum(y_pred - y_)
```

替换为

```
grad_a, grad_b = tf.gradients(loss, [a, b])
```

计算结果将不会改变。

甚至不仅于此，TensorFlow 附带有多种 **优化器**（optimizer），可以将求导和梯度更新一并完成。我们可以将上节的代码

```
# 反向传播，手动计算变量（模型参数）的梯度
grad_a = tf.reduce_sum((y_pred - y_) * X_)
grad_b = tf.reduce_sum(y_pred - y_)

# 梯度下降法，手动更新参数
new_a = a - learning_rate_ * grad_a
new_b = b - learning_rate_ * grad_b
update_a = tf.assign(a, new_a)
update_b = tf.assign(b, new_b)

train_op = [update_a, update_b]
```

整体替换为

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate_)
grad = optimizer.compute_gradients(loss)
train_op = optimizer.apply_gradients(grad)
```

这里，我们先实例化了一个 TensorFlow 中的梯度下降优化器 `tf.train.GradientDescentOptimizer()` 并设置学习率。然后利用其 `compute_gradients(loss)` 方法求出 `loss` 对所有变量（参数）的梯度。最后通过 `apply_gradients(grad)` 方法，根据前面算出的梯度来梯度下降更新变量（参数）。

以上三行代码等价于下面一行代码：

```
train_op = tf.train.GradientDescentOptimizer(learning_rate=learning_rate_).minimize(loss)
```

简化后的代码如下：

```
import tensorflow as tf

learning_rate_ = tf.placeholder(dtype=tf.float32)
X_ = tf.placeholder(dtype=tf.float32, shape=[5])
y_ = tf.placeholder(dtype=tf.float32, shape=[5])
```

```
a = tf.get_variable('a', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)
b = tf.get_variable('b', dtype=tf.float32, shape=[], initializer=tf.zeros_initializer)

y_pred = a * X_ + b
loss = tf.constant(0.5) * tf.reduce_sum(tf.square(y_pred - y_))

# 反向传播，利用 TensorFlow 的梯度下降优化器自动计算并更新变量（模型参数）的梯度
train_op = tf.train.GradientDescentOptimizer(learning_rate=learning_rate_).minimize(loss)

num_epoch = 10000
learning_rate = 1e-3
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for e in range(num_epoch):
        sess.run(train_op, feed_dict={X_: X, y_: y, learning_rate_: learning_rate})
    print(sess.run([a, b]))
```

Bibliography

[LeCun1998] 25. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998. http://yann.lecun.com/exdb/mnist/

[Graves2013] Graves, Alex. "Generating Sequences With Recurrent Neural Networks." ArXiv:1308.0850 [Cs], August 4, 2013. http://arxiv.org/abs/1308.0850.

[Mnih2013] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing Atari with Deep Reinforcement Learning." ArXiv:1312.5602 [Cs], December 19, 2013. http://arxiv.org/abs/1312.5602.