第 3 章
TensorFlow 架构与主要功能

3-1 常用的深度学习框架

维基百科[1]列举了近 20 种的深度学习框架，有许多已经逐渐被淘汰了，以 Python/C++ 语言的框架而言，目前比较流行的只剩以下四种，具体见表 3.1。

表 3.1 Python/C++ 语言框架

框架名称	程序语言	优点	缺点
TensorFlow/Keras	Python、C++、JS、Java、Go	1.Keras 简单、易入门，支持多种程序语言； 2. 效果佳	与 Python 未完全整合，如模型结构不能直接使用 if
PyTorch	Python、C++、Java	1. 与 Python 整合较好，有弹性； 2. 学界采用占比有增加的趋势	执行速度较慢
Caffe	C++	1. 执行速度非常快； 2. 专长于影像应用，NVIDIA 有一改良版	1. 复杂； 2. 无 NLP 模块
Apache MXNet	Python、Scala、Julia	AWS/Azure 云端支持	复杂

TensorFlow、PyTorch 分居占有率的前两名，其他框架均望尘莫及，如图 3.1 所示。因此推荐读者若是使用 Python 语言，熟悉 TensorFlow、PyTorch 就绰绰有余了；若偏好 C/C++，则可以使用 Caffe 框架。

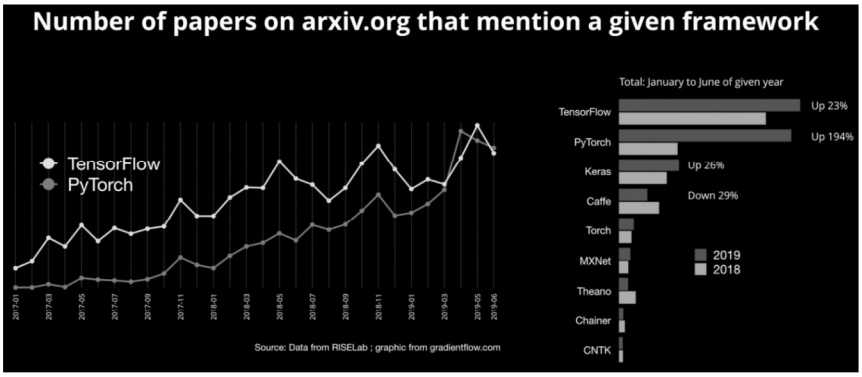

图 3.1 深度学习框架在 Arxiv 网站的论文采用比率

(图片来源：*Which deep learning framework is the best?* [2])

本书的范例以 TensorFlow/Keras 为主，不会涉及其他深度学习框架，以期对 TensorFlow/Keras 进行全面而深入的探讨。

3-2 TensorFlow 架构

梯度下降法是神经网络主要求解的方法，计算过程需要使用大量的张量运算，另外，在反向传导的过程中，要进行偏微分，并需解决多层结构的神经网络问题。因此，大多数的深度学习框架至少都会具备以下功能。

(1) 张量运算：包括各种向量、矩阵运算。

(2) 自动微分。

(3) 神经网络及各种神经层 (Layers)。

学习的路径可以从简单的张量运算开始，再逐渐熟悉高阶的神经层函数，以奠定扎实的基础。

接着再来了解 TensorFlow 执行环境 (Runtime)，如图 3.2 所示。它支持 GPU、分布式计算、低阶 C API 等，也支持多种网络协议。

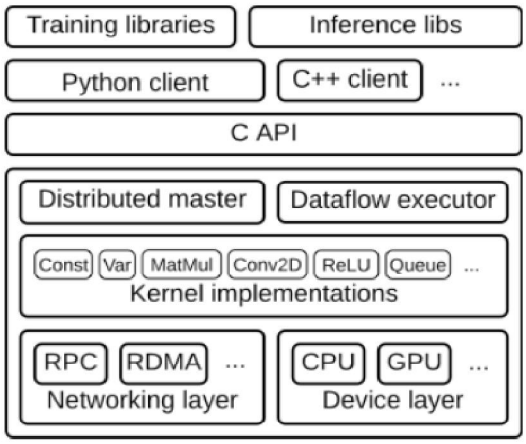

图 3.2 TensorFlow 执行环境 (Runtime)

(图片来源：TensorFlow GitHub [3])

TensorFlow 执行环境包括以下三个版本。

(1) TensorFlow：一般计算机版本。

(2) TensorFlowjs：网页版本，适合边缘运算的装置及虚拟机装置 (Docker/Kubernetes)。

(3) TensorFlow Lite：轻量版，适合指令周期及内存有限的移动设备和物联网装置。

程序设计堆栈 (Programming Stack) 如图 3.3 所示，在 2.x 版后以 Keras 为主轴，TensorFlow 团队依照 Keras 的规格重新开发，逐步整合其他模块，比独立开发的 Keras 框架功能更加强大，因而 Keras “大神” François Chollet 也宣告 Keras 独立框架不再升级，甚至将 Keras 官网也改为介绍 TensorFlow 的 Keras 模块，而非自家开发的 Keras 独立框架。所以，现在要撰写 Keras 程序，应以 TensorFlow 为主，避免再使用 Keras 独立框架。

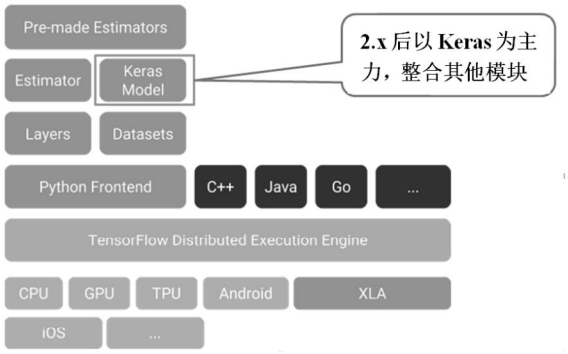

图 3.3　TensorFlow 程序设计堆栈 (Programming Stack)

(图片来源：*Explained*：*Deep Learning in Tensorflow* [4])

TensorFlow Keras 模块逐步整合其他模块及工具，如果读者之前使用的是 Keras 独立框架，则可以补强以下项目。

(1) 低阶梯度下降的训练 (GradientTape)。

(2) 数据集载入 (Dataset and Loader)。

(3) 回调函数 (Callback)。

(4) 估计器 (Estimator)。

(5) 预训模型 (Keras Application)。

(6) TensorFlow Hub：进阶的预训模型。

(7) TensorBoard 可视化工具。

(8) TensorFlow Serving 部署工具。

我们会在后面的章节陆续探讨各项功能。

除了 Keras 的引进，特别要注意的是，**TensorFlow 2.x 版之后，默认模式一修改为 Eager Execution Mode**，舍弃了 Session 语法，若未调回原先的 Graph Execution Mode，则 1.x 版的程序执行时将会产生错误。由于目前网络上许多范例的程序新旧杂陈，还有很多 1.x 版的程序，读者应特别注意。

3-3　张量运算

TensorFlow 顾名思义就是提供张量的定义、计算与变量值的传递功能。因此，它最底层的功能即支持各项张量的数据类型与其运算函数，基本上都遵循 NumPy 库的设计理念与语法，包括传播 (Broadcasting) 机制的支持。

以下我们直接以实践代替长篇大论，读者请参阅 **03_3_ 张量运算 .ipynb**。

(1) 显示 TensorFlow 版本。程序代码如下：

```
# 载入库
import tensorflow as tf

# 显示版本
print(tf.__version__)
```

(2) 检查 GPU 是否存在。程序代码如下：

```
# check cuda available
tf.config.list_physical_devices('GPU')
```

执行结果如下，可显示 GPU 简略信息：

[PhysicalDevice(name='/physical_device：GPU：0',device_type='GPU')]

(3) 如果要知道 GPU 的详细规格，可安装 PyCuda 模块。请注意在 Windows 环境下，无法以 pip install pycuda 安装，可到 https://www.lfd.uci.edu/~gohlke/pythonlibs/?cm_mc_uid=08085305845514542921829&cm_mc_sid_50200000=1456395916#pycuda 下载对应 Python、Cuda Toolkit 版本的二进制文件。

举例来说，Python v3.8 且安装 Cuda Toolkit v10.1，则须下载 pycuda-2020.1+cuda101-cp38-cp38-win_amd64.whl，并执行 pip install pycuda-2020.1+cuda101-cp38-cp38-win_amd64.whl。接着就可以执行本书所附的范例：python GpuQuery.py。

执行结果如下，即可显示 GPU 的详细规格，重要信息排列在前面。

```
装置 0: NVIDIA GeForce GTX 1050 Ti
    计算能力: 6.1
    GPU记忆体: 4096 MB
    6 个处理器, 各有 128 个CUDA核心数, 共 768 个CUDA核心数

    ASYNC_ENGINE_COUNT: 5
    CAN_MAP_HOST_MEMORY: 1
    CLOCK_RATE: 1392000
    COMPUTE_CAPABILITY_MAJOR: 6
    COMPUTE_CAPABILITY_MINOR: 1
    COMPUTE_MODE: DEFAULT
    CONCURRENT_KERNELS: 1
    ECC_ENABLED: 0
    GLOBAL_L1_CACHE_SUPPORTED: 1
    GLOBAL_MEMORY_BUS_WIDTH: 128
```

(4) 声明常数 (constant)，参数可以是常数、list、NumPy array。程序代码如下：

```
# 声明常数(constant)，参数可以是常数、list、Numpy Array
x = tf.constant([[1, 2]])
```

(5) 支持四则运算。程序示例如下：

```
print(x+10)
print(x-10)
print(x*2)
print(x/2)
```

执行结果如下：

tf.Tensor([[11 12]], shape=(1, 2), dtype=int32)

tf.Tensor([[-9 -8]], shape=(1, 2), dtype=int32)

tf.Tensor([[2 4]], shape=(1, 2), dtype=int32)

tf.Tensor([[0.5 1.]], shape=(1, 2), dtype=float64)

注意：如果要显示数值，须转换为 NumPy array。例如：

(x+10).numpy()

(6) 四则运算也可以使用 TensorFlow 函数。程序代码如下：

```
# 转为负数
print(tf.negative(x))

# 常数、List、NumPy array 均可运算
print(tf.add(1, 2))
print(tf.add([1, 2], [3, 4]))

print(tf.square(5))
print(tf.reduce_sum([1, 2, 3]))

# 混用四则运算符号及TensorFlow 函数
print(tf.square(2) + tf.square(3))
```

reduce_sum 是沿着特定轴加总，输出会少一维，故 [1, 2, 3] 套用函数运算后等于 6。

(7) TensorFlow 会自动决定在 CPU 或 GPU 运算，可由下列指令侦测。一般而言，常数 (tf.constant) 放在 CPU，其他变量则放在 GPU 上，两者加总时，会将常数搬到 GPU 上再加总，过程不需要人为操作。但请**注意，PyTorch 框架稍显麻烦，必须手动将变量搬移至 CPU 或 GPU 运算，不允许一个变量在 CPU，另一个变量在 GPU**。侦测程序代码如下：

```
x1 = tf.constant([[1, 2, 3]], dtype=float)
print("x1 是否在 GPU #0 上:  ", x1.device.endswith('GPU:0'))

# 设定 x 为均匀分配乱数 3x3
x2 = tf.random.uniform([3, 3])
print("x2 是否在 GPU #0 上:  ", x2.device.endswith('GPU:0'))

x3=x1+x2
print("x3 是否在 GPU #0 上:  ", x3.device.endswith('GPU:0'))
```

执行结果：

x1 是否在 GPU #0 上：False

x2 是否在 GPU #0 上：True

x3 是否在 GPU #0 上：True

(8) 用户也能够使用 with tf.device("CPU：0") 或 with tf.device("GPU：0")，强制指定在 CPU 或 GPU 运算。程序代码如下：

```
import time

# 计算 10 次的时间
def time_matmul(x):
    start = time.time()
    for loop in range(10):
        tf.matmul(x, x)

    result = time.time()-start
    print("{:0.2f}ms".format(1000*result))

# 强制指定在CPU运算
print("On CPU:", end='')
with tf.device("CPU:0"):
    x = tf.random.uniform([1000, 1000])
    assert x.device.endswith("CPU:0")
    time_matmul(x)

# 强制指定在GPU运算
if tf.config.list_physical_devices("GPU"):
    print("On GPU:", end='')
    with tf.device("GPU:0"):
        x = tf.random.uniform([1000, 1000])
        assert x.device.endswith("GPU:0")
        time_matmul(x)
```

① 第一次执行结果如下，CPU 运算比 GPU 快：

On CPU：64.00ms

On GPU：311.49ms

② 多次执行后，GPU 反而比 CPU 运算快了很多：

On CPU：58.00ms

On GPU：1.00ms

(9) 稀疏矩阵 (Sparse Matrix) 运算：稀疏矩阵是指矩阵内只有很少数的非零元素，如果依一般的矩阵存储会非常浪费内存，运算也是如此，因为大部分项目为零，不需浪费时间计算，所以，科学家针对稀疏矩阵设计出了特殊的数据存储结构及运算算法，TensorFlow 也支持此类数据类型，如图 3.4 所示。

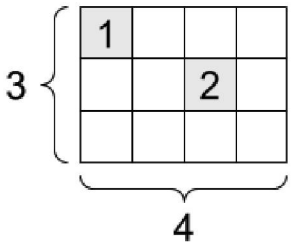

图 3.4　稀疏矩阵

(10) TensorFlow 稀疏矩阵只需设定有值的位置和数值，并设定维度如下：

```
# 稀疏矩阵只需设定有值的位置及数值
sparse_tensor = tf.sparse.SparseTensor(indices=[[0, 0], [1, 2]],
                                       values=[1, 2],
                                       dense_shape=[3, 4])
print(sparse_tensor)
```

执行结果如下：

```
SparseTensor(indices=tf.Tensor(
[[0 0]
 [1 2]], shape=(2, 2), dtype=int64), values=tf.Tensor([1 2], shape=(2,), dtype=int32), dense_shape=tf.Tensor([3 4], shape=(2,),
dtype=int64))
```

(11) 转为正常的矩阵格式。程序代码如下：

```
# 转为正常的矩阵格式
x = tf.sparse.to_dense(sparse_tensor)
print(type(x))

# 2.31 以前版本会出错
x.numpy()
```

执行结果如下：

```
<class 'tensorflow.python.framework.ops.EagerTensor'>

array([[1, 0, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 0]])
```

(12) 如果要执行 TensorFlow 1.x 版 Graph Execution Mode 的程序，则需禁用

(Disable)2.x 版的功能，并改变加载框架的命名空间 (Namespace)。程序代码如下：

```
if tf.__version__[0] != '1':           # 是否为 TensorFlow 1.x版
    import tensorflow.compat.v1 as tf  # 改变载入框架的命名空间(Namespace)
    tf.disable_v2_behavior()           # 使 2.x 版功能失效(Disable)
```

TensorFlow 改良反而带来了后向兼容性差的困扰，1.x 版的程序在 2.x 版的默认模式下均无法执行，虽然可以把 2.x 版的默认模式切换回 1.x 版的模式，但是要自行修改的地方较多，而且也缺乏未来性，因此，在这里建议大家以下几点。

① Eager Execution Mode 已是 TensorFlow 的主流，不要再使用 1.x 版的 Session 或 TFLearn 等旧的架构，套用一句电视剧对白，**已经回不去了**。

② 手上有许多 1.x 版的程序，如果很重要，非用不可，可利用官网移转 (Migration) 指南 [5] 进行修改，单一文件比较可行，但若是复杂的框架，可能就要花费大量时间了。

③ 官网也有提供指令，能够一次升级整个目录的所有程序，如下：

tf_upgrade_v2 --intree <1.x 版程序目录 > --outtree < 输出目录 >

详细使用方法可参阅 TensorFlow 官网升级指南 [6]。

(13) 禁用 2.x 版的功能后，测试 1.x 版程序，Graph Execution Mode 程序须使用 tf.Session。程序代码如下：

```
# 测试1.x版程序
x = tf.constant([[1, 2]])
neg_x = tf.negative(x)

with tf.Session() as sess:   # 使用 Session
    result = sess.run(neg_x)
    print(result)
```

(14) GPU 内存管理：由于 TensorFlow 对 GPU 内存的垃圾回收 (Garbage Collection) 机制并不完美，因此，常会出现下列 GEMM 错误：

```
InternalError:  Blas GEMM launch failed : a.shape=(32, 784), b.shape=(784, 256), m=32, n=256, k=784
        [[node sequential/dense/MatMul (defined at <ipython-input-3-9d42ad511782>:5) ]] [Op:__inference_train_function_581]
```

此信息表示 GPU 内存不足，尤其是使用 Jupyter Notebook 时，因为 Jupyter Notebook 是一个网页程序，关掉某一个 Notebook 文件，网站仍然在执行中，所以不代表该文件的资源会被回收，通常要选择 **Kernel > Restart** 选项才会回收资源。另一个方法，就是限制 GPU 的使用配额。以下是 TensorFlow 2.x 版的方式，1.x 版并不适用。程序代码如下：

```
# 限制 TensorFlow 只能使用 GPU 2GB 内存
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        # 限制 第一个 GPU 只能使用 2GB 内存
        tf.config.experimental.set_virtual_device_configuration(gpus[0],
            [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=1024*2)])

        # 显示 GPU 个数
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # 显示错误信息
        print(e)
```

(15) 用户也可以不使用 GPU。程序代码如下：

```
import os

os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

3-4 自动微分

反向传导时，会更新每一层的权重，这时就会用到偏微分运算，如图 3.5 所示。所以，深度学习框架的第二项主要功能就是自动微分。

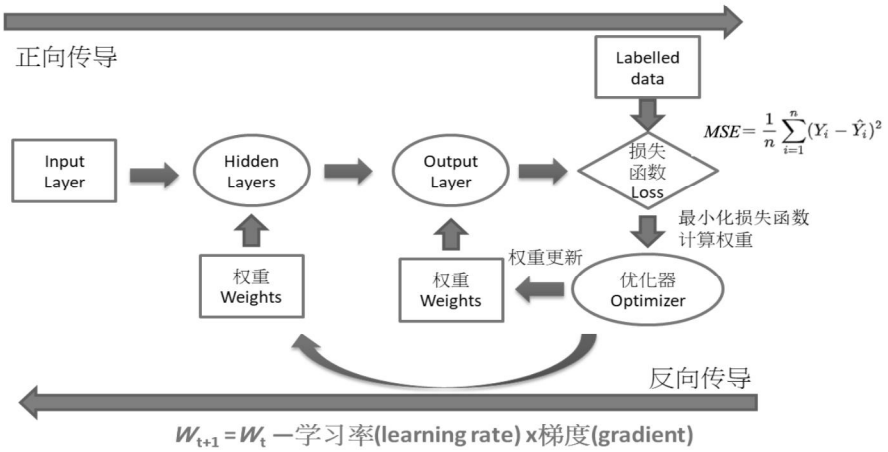

图 3.5 神经网络权重求解过程

同样地，我们直接以实践代替长篇大论，请参阅 **03_2_ 自动微分 .ipynb**。

(1) 使用 tf.GradientTape() 函数可自动微分，再使用 g.gradient(y, x) 可取得 y 对 x 作偏微分的梯度。程序代码如下：

```
import numpy as np
import tensorflow as tf

x = tf.Variable(3.0)          # 声明 TensorFlow 变量(Variable)

with tf.GradientTape() as g: # 自动微分
    y = x * x                 # y = x^2

dy_dx = g.gradient(y, x)      # 取得梯度， f'(x) = 2x, x=3 ==> 6

print(dy_dx.numpy())          # 转换为 NumPy array 格式
```

执行结果：

$f(x) = x^2$

$f'(x) = 2x$

$f'(3) = 2 * 3 = 6$。

(2) 声明为变量 (tf.Variable) 时，该变量会自动参与自动微分，但声明为常数 (tf.constant) 时，如欲参与自动微分，则需额外设定 g.watch()。程序代码如下：

```
import numpy as np
import tensorflow as tf

x = tf.constant(3.0)          # 声明 TensorFlow 常数

with tf.GradientTape() as g: # 自动微分
    g.watch(x)               # 设定常数参与自动微分
    y = x * x                # y = x^2

dy_dx = g.gradient(y, x)     # 取得梯度， f'(x) = 2x, x=3 ==> 6

print(dy_dx.numpy())         # 转换为 NumPy array 格式
```

执行结果：与上面 (1) 中相同。

(3) 计算二阶导数：使用 tf.GradientTape()、g.gradient(y, x) 函数两次，即能取得二阶导数。程序代码如下：

```
x = tf.constant(3.0)              # 声明 TensorFlow 常数
with tf.GradientTape() as g:      # 自动微分
    g.watch(x)
    with tf.GradientTape() as gg: # 自动微分
        gg.watch(x)               # 设定常数参与自动微分
        y = x * x                 # y = x^2

    dy_dx = gg.gradient(y, x)     # 一阶导数
d2y_dx2 = g.gradient(dy_dx, x)    # 二阶导数

print(f'一阶导数={dy_dx.numpy()}, 二阶导数={d2y_dx2.numpy()}')
```

执行结果：一阶导数 =6.0，二阶导数 =2.0。

$f(0)= x^2$

$f'(x) = 2x$

$f''(x) = 2$

$f''(3) = 2$。

(4) 多变量计算导数：各自使用 g.gradient(y, x) 函数，可取得每一个变量的梯度。若使用 g.gradient() 两次或以上，则 tf.GradientTape() 须加参数 persistent=True，使 tf.GradientTape() 不会被自动回收，用完之后，可使用“del g”删除 GradientTape 对象。程序代码如下：

```
x = tf.Variable(3.0)          # 声明 TensorFlow 常数
with tf.GradientTape(persistent=True) as g:  # 自动微分
    y = x * x                 # y = x^2
    z = y * y                 # z = y^2

dz_dx = g.gradient(z, x)      # 4*x^3
dy_dx = g.gradient(y, x)      # 2*x

del g                         # 不用时可删除 GradientTape 对象

print(f'dy/dx={dy_dx.numpy()}, dz/dx={dz_dx.numpy()}')
```

执行结果：dy/dx=6，dz/dx=108。

$z = f(x)= y^2 = x^4$

$f'(x) = 4x^3$

$f'(3) = 108$。

(5) 借此机会我们认识一下 PyTorch 自动微分的语法，它与 TensorFlow 稍有差异。程序代码如下：

```
import torch        # 载入库

x = torch.tensor(3.0, requires_grad=True)  # 设定 x 参与自动微分
y=x*x               # y = x^2

y.backward()        # 反向传导

print(x.grad)       # 取得梯度
```

① requires_grad=True 参数声明 x 参与自动微分。

②调用 y.backward()，要求做反向传导。

③调用 x.grad 取得梯度。

范例. 利用TensorFlow自动微分求解简单线性回归的参数(w、b)。

程序：请参阅 **03_3_ 简单线性回归 .ipynb**。流程如图 3.6 所示。

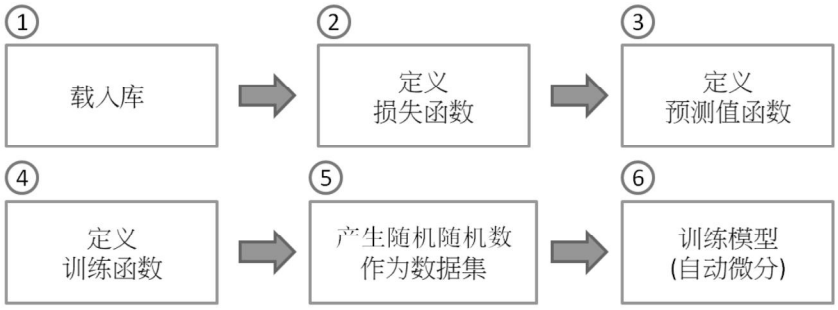

图 3.6　程序设计流程

(1) 载入库。程序代码如下：

```
# 载入库
import numpy as np
import tensorflow as tf
```

(2) 定义损失函数 $MSE=\dfrac{\sum(y-\hat{y})^2}{n}$。程序代码如下：

```
# 定义损失函数
def loss(y, y_pred):
    return tf.reduce_mean(tf.square(y - y_pred))
```

(3) 定义预测值函数 $y=wx+b$。程序代码如下：

```
# 定义预测值函数
def predict(X):
    return w * X + b
```

(4) 定义训练函数：在自动微分中需重新计算损失函数值，assign_sub 函数相当于“-=”。程序代码如下：

```python
# 定义训练函数
def train(X, y, epochs=40, lr=0.0001):
    current_loss=0                                  # 损失函数值
    for epoch in range(epochs):                     # 执行训练周期
        with tf.GradientTape() as t:                # 自动微分
            t.watch(tf.constant(X))                 # 创建 TensorFlow 常数参与自动微分
            current_loss = loss(y, predict(X))      # 计算损失函数值

        dw, db = t.gradient(current_loss, [w, b]) # 取得 w, b 个别的梯度

        # 更新权重：新权重 = 原权重 – 学习率(learning_rate) * 梯度(gradient)
        w.assign_sub(lr * dw) # w -= lr * dw
        b.assign_sub(lr * db) # b -= lr * db

        # 显示每一训练周期的损失函数
        print(f'Epoch {epoch}: Loss: {current_loss.numpy()}')
```

(5) 产生随机数作为数据集，进行测试。程序代码如下：

```python
# 产生线性随机数据100批，介于 0~50
n = 100
X = np.linspace(0, 50, n)
y = np.linspace(0, 50, n)

# 数据里加一点噪声(noise)
X += np.random.uniform(-10, 10, n)
y += np.random.uniform(-10, 10, n)
```

(6) 执行训练。程序代码如下：

```python
# w、b 初始值均设为 0
w = tf.Variable(0.0)
b = tf.Variable(0.0)

# 执行训练
train(X, y)

# w、b 的最佳解
print(f'w={w.numpy()}, b={b.numpy()}')
```

① 执行结果：w=0.9464，b=0.0326。

② 损失函数值随着训练周期越来越小，如下：

```
Epoch 0: Loss: 890.1063232421875
Epoch 1: Loss: 607.071533203125
Epoch 2: Loss: 419.7763671875
Epoch 3: Loss: 295.8358459472656
Epoch 4: Loss: 213.81948852539062
Epoch 5: Loss: 159.5459747314453
Epoch 6: Loss: 123.63106536865234
Epoch 7: Loss: 99.86465454101562
Epoch 8: Loss: 84.1374282836914
Epoch 9: Loss: 73.7300033569336
Epoch 10: Loss: 66.84292602539062
Epoch 11: Loss: 62.28538513183594
Epoch 12: Loss: 59.269386291503906
Epoch 13: Loss: 57.27349090576172
Epoch 14: Loss: 55.95263671875
Epoch 15: Loss: 55.078487396240234
Epoch 16: Loss: 54.49993133544922
Epoch 17: Loss: 54.116981506347656
Epoch 18: Loss: 53.86347579956055
Epoch 19: Loss: 53.69562911987305
Epoch 20: Loss: 53.584468841552734
```

(7) 显示结果：回归线确实居于样本点中线。程序代码如下：

```
import matplotlib.pyplot as plt

plt.scatter(X, y, label='data')
plt.plot(X, predict(X), 'r-', label='predicted')
plt.legend()
```

执行结果：如图 3.7 所示。

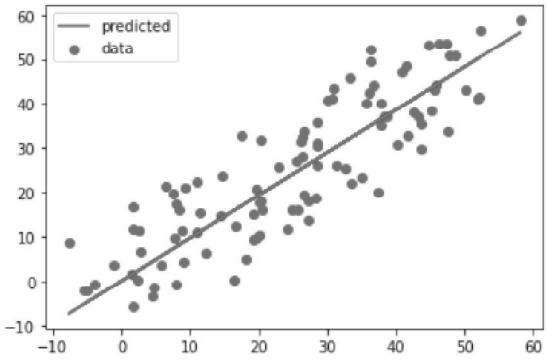

图 3.7　自动微分求解简单线性回归执行结果

有了 TensorFlow 自动微分的功能，正向与反向传导变得非常简单，只要熟悉了运作的架构，后续复杂的模型就可以运用自如。

3-5　神经网络层

上一节运用自动微分实现了一条简单线性回归线的求解，然而神经网络是多条回归线的组合，并且每一条回归线可能再乘上非线性的 Activation Function，假如使用自动微分函数逐一定义每条公式，层层串连，程序可能要很多个循环才能完成。所以为了简化程序开发的复杂度，TensorFlow/Keras 直接建构了各式各样的神经层函数，可以使用神经层组合神经网络的结构，用户只需要专注算法的设计即可，轻松不少。

如图 3.8 所示，神经网络是多个神经层组合而成的，包括输入层 (Input Layer)、隐藏层 (Hidden Layer) 及输出层 (Output Layer)，其中隐藏层可以有任意多层。一般而言，隐藏层大于或等于两层，即称为深度学习。

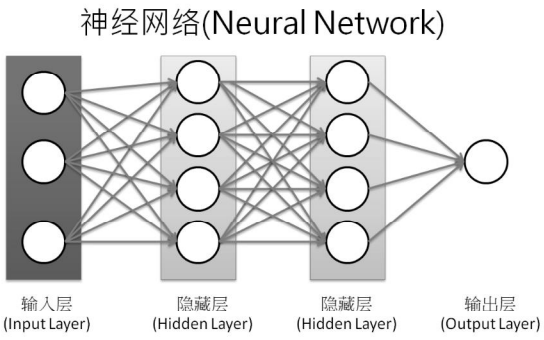

图 3.8　神经网络示意图

TensorFlow/Keras 提供了数十种神经层，分成以下类别，用户可参阅 Keras 官网说明 (https://keras.io/api/layers/)。

(1) 核心类别 (Core Layer)：包括完全连接层 (Full Connected Layer)、激励神经层 (Activation layer)、嵌入层 (Embedding layer) 等。

(2) 卷积层 (Convolutional Layer)。

(3) 池化层 (Pooling Layer)。

(4) 循环层 (Recurrent Layer)。

(5) 前置处理层 (Preprocessing layer)：提供 One-Hot Encoding、影像前置处理、数据增补 (Data Augmentation) 等。

我们先来看看两个最简单的完全连接层范例。

范例1. 使用完全连接层估算简单线性回归的参数(w、b)。

程序：请参阅 03_4_ 简单的完全连接层 .ipynb。

(1) 产生随机数据，与上一节范例相同。程序代码如下：

```
# 载入库
import numpy as np
import tensorflow as tf

# 产生线性随机数据100批，介于 0-50
n = 100
X = np.linspace(0, 50, n)
y = np.linspace(0, 50, n)

# 数据中加一点噪声
X += np.random.uniform(-10, 10, n)
y += np.random.uniform(-10, 10, n)
```

(2) 建立模型：神经网络仅使用一个完全连接层，而且输入只有一个神经元，即 X，输出也只有一个神经元，即 y。Dense 本身有一个参数 use_bias，即是否有偏差项，默认值为 True，除了一个神经元输出外，还会有一个偏差项。以上设定其实就等于 $y=wx+b$。为聚焦概念的说明，暂时不解释其他参数，在后面章节会有详尽说明。程序代码如下：

```
# 定义完全连接层(Dense)
# units：输出神经元个数，input_shape：输入神经元个数
layer1 = tf.keras.layers.Dense(units=1, input_shape=[1])

# 神经网络包含一个完全连接层
model = tf.keras.Sequential([layer1])
```

(3) 定义模型的损失函数及优化器。程序代码如下：

```
# 定义模型的损失函数为 MSE，优化器为 Adam
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam())
```

(4) 模型训练：只需一个指令 model.fit(X, y) 即可，训练过程的损失函数变化都会存在 history 变量中。程序代码如下：

```
history = model.fit(X, y, epochs=500, verbose=False)
```

(5) 训练过程绘图。程序代码如下：

```
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['Microsoft JhengHei']
plt.rcParams['axes.unicode_minus'] = False

plt.xlabel('训练周期', fontsize=20)
plt.ylabel("损失函数", fontsize=20)
plt.plot(history.history['loss'])
```

执行结果：损失函数值随着训练周期越来越小，如图 3.9 所示。

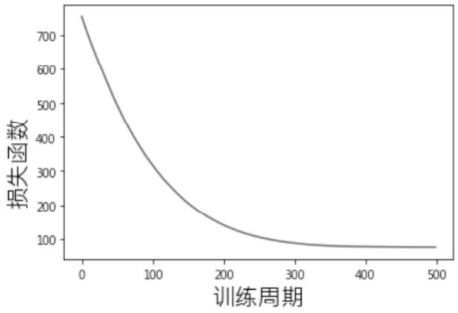

图 3.9　训练过程执行结果

(6) 取得模型参数 w 为第一层的第一个参数，b 为输出层的第一个参数。程序代码如下：

```
w = layer1.get_weights()[0][0][0]
b = layer1.get_weights()[1][0]

print(f"w：{w:.4f} , b：{b:.4f}")
```

执行结果：w：0.8798，b：3.5052，因输入数据为随机随机数。

(7) 绘图显示回归线。程序代码如下：

```
import matplotlib.pyplot as plt

plt.scatter(X, y, label='data')
plt.plot(X, X * w + b, 'r-', label='predicted')
plt.legend()
```

执行结果：如图 3.10 所示。

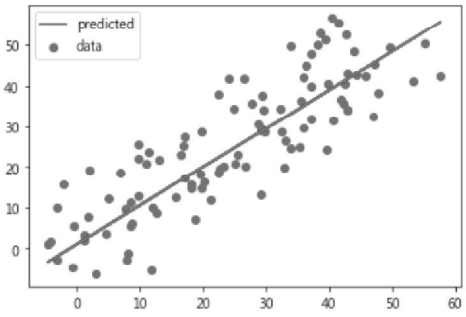

图 3.10　绘图显示回归线执行结果

与自动微分比较，这种方法程序更简单，只要设定模型结构、损失函数、优化器后，

呼叫训练 (fit) 函数即可。

下面我们再看一个有趣的例子，利用神经网络自动求出华氏与摄氏温度的换算公式。

范例2. 使用完全连接层推算华氏与摄氏温度的换算公式。

$$华氏\ (F)= 摄氏\ (C)*(9/5)+ 32$$

(1) 利用换算公式，随机产生 151 个数据。程序代码如下：

```
# 载入库
import numpy as np
import tensorflow as tf

# 随机产生151个数据
n = 151
C = np.linspace(-50, 100, n)
F = C * (9/5) + 32

for i, x in enumerate(C):
    print(f"华氏(F)：{F[i]:.2f} , 摄氏(C)：{x:.0f}")
```

(2) 建立模型：神经网络只有一个完全连接层，而且输入只有一个神经元，即摄氏温度，输出只有一个神经元，即华氏温度。程序代码如下：

```
# 定义完全连接层(Dense)
# units：输出神经元个数，input_shape：输入神经元个数
layer1 = tf.keras.layers.Dense(units=1, input_shape=[1])

# 神经网路包含一层完全连接层
model = tf.keras.Sequential([layer1])

# 定义模型的损失函数为 MSE，优化器为 Adam
model.compile(loss='mean_squared_error',
              optimizer=tf.keras.optimizers.Adam(0.1))
```

(3) 模型训练。程序代码如下：

```
history = model.fit(C, F, epochs=500, verbose=False)
```

(4) 训练过程绘图。程序代码如下：

```
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['Microsoft JhengHei']
plt.rcParams['axes.unicode_minus'] = False

plt.xlabel('训练周期', fontsize=20)
plt.ylabel("损失函数", fontsize=20)
plt.plot(history.history['loss'])
```

执行结果：如图 3.11 所示。由此可见：损失函数值随着训练周期越来越小。

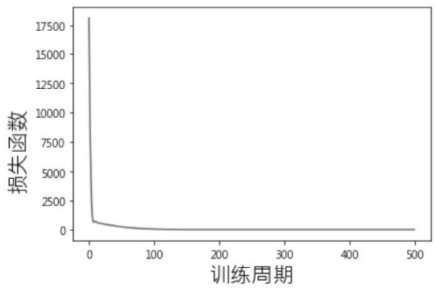

图 3.11　训练过程执行结果

(5) 测试：输入摄氏 100 度及 0 度转换为华氏温度，答案完全正确。程序代码如下：

```
y_pred = model.predict([100.0])[0][0]
print(f"华氏(F)：{y_pred:.2f} , 摄氏(C)：100")

y_pred = model.predict([0.0])[0][0]
print(f"华氏(F)：{y_pred:.2f} , 摄氏(C)：0")
```

执行结果：

华氏 (F)：212.00，摄氏 (C)：100

华氏 (F)：32.00，摄氏 (C)：0

(6) 取得模型参数 w、b。

```
w = layer1.get_weights()[0][0][0]
b = layer1.get_weights()[1][0]

print(f"w：{w:.4f} , b：{b:.4f}")
```

① 执行结果：w：1.8000，b：31.9999，近似于华氏与摄氏温度的换算公式。

② 其实换算公式也是一条回归线。

读到这里，读者应该会好奇如何使用更多的神经元和神经层，甚至更复杂的神经网络结构，下一章我们将正式迈入深度学习的殿堂，学习如何用 TensorFlow 解决各种实际的案例，并且详细剖析各个函数的用法及参数说明。