第 2 章

TensorFlow 2.0 介绍

学习目标

- 认识深度学习
- 说明 TensorFlow 2.0 的重要更新
- 认识 Eager Execution 模式
- 学习 TensorFlow 的基本运算
- 学习使用 tf.keras 搭建网络模型
- 概述 tf.data 的用途以及基本操作

2.1 什么是深度学习

人工智能（Artificial Intelligence，AI）是指赋予机器思考或学习能力，主要用来协助人类或取代那些重复性高、技能含量低的工作，让人们可以专注在更有意义的事情上。人工智能是一个综合的领域，其中包含进化计算（Evolutionary Computation）、专家系统（Expert Systems）、符号人工智能（Symbolic Artificial Intelligence）、支持向量机（Support Vector Machine）、机器学习（Machine Learning）、深度学习（Deep Learning）、强化学习（Reinforcement Learning）等众多领域，而深度学习又是机器学习的一个分支领域，如图 2-1 所示。

在深度学习中，可以将深度学习网络视为一个复杂函数，当输入一组训练数据到神经网络时，可得到一组输出预测，接着根据输出预测与标记答案的差距更新神经网络的权重（Weight），使其更加逼近于期望的输出，即标记答案。下面我们将介绍热门和流行的人工智能工具 TensorFlow。

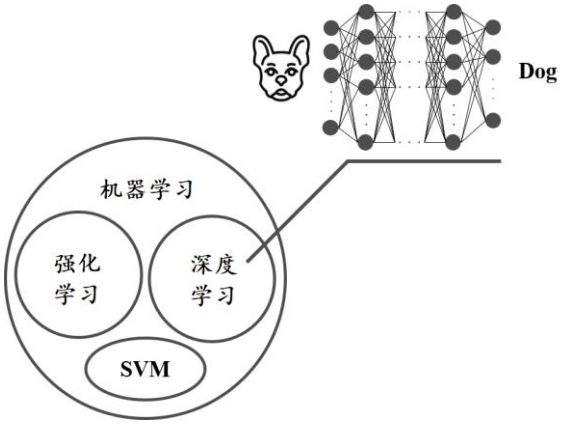


图 2-1　深度学习是目前机器学习热门的领域

2.2　建立项目

建议使用 Jupyter Notebook 来执行本章的程序代码，操作流程如下：

Step 01　启动 Jupyter Notebook。

在 Terminal（Ubuntu）或命令提示符（Windows）中输入以下指令：

```
.\tf2\Scripts\activate
jupyter notebook
```

Step 02　新建执行文件。

单击界面右上角的 New 下拉按钮，然后单击所安装的 Python 解释器（在 Jupyter 中都称为 Kernel）来启动，如图 2-2 所示，显示了 3 个不同的 Kernel，分别为：

- Python 3：本地端 Python。
- tf2：虚拟机 Python（TensorFlow-cpu 版本）。
- tf2-gpu：虚拟机 Python（TensorFlow-gpu 版本）。

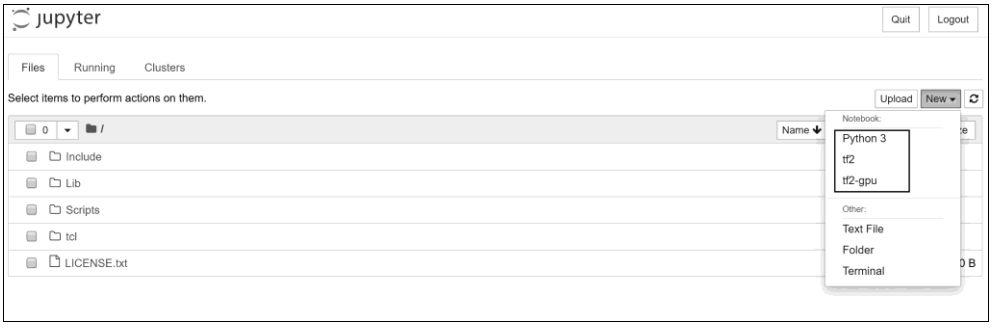


图 2-2　新建执行文件

Step 03 执行程序代码。

在方框中输入程序代码“print("Hello Jupyter Notebook")”，再按 Shift + Enter 快捷键来执行单行程序代码，执行结果会显示在该行程序代码的下方，如图 2-3 所示。

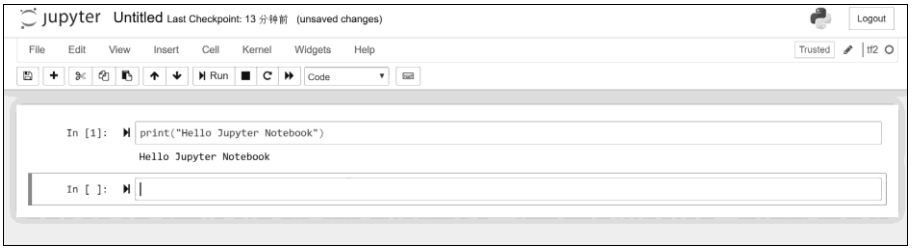


图 2-3 Jupyter 环境界面

Step 04 载入 TensorFlow 套件。

在方框中输入“import tensorflow as tf”，再按 Shift + Enter 快捷键来执行单行程序代码，即可载入 TensorFlow 套件，如图 2-4 所示。

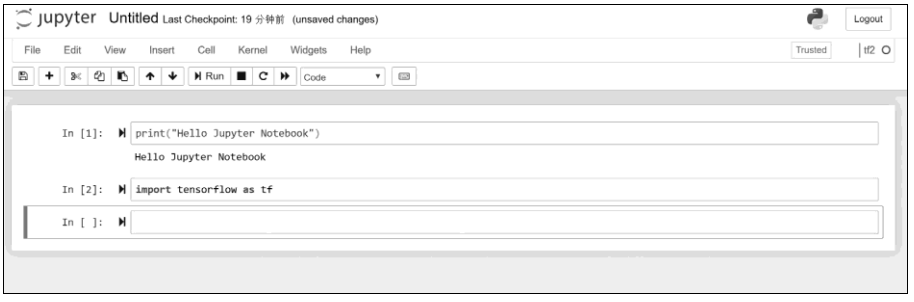


图 2-4 载入 TensorFlow 套件

接下来，本章的范例程序代码都可在 Jupyter Notebook 上执行。

2.3 TensorFlow 介绍

DistBelief 为 Google Brain 团队一开始所使用的机器学习工具，后来以 DistBelief 为基础做了改进，并开放了源代码，才有了现在我们所熟知的 TensorFlow。目前，TensorFlow 已成为流行的机器学习工具之一，而 TensorFlow 的命名也直接反映了其功能，可以将名字拆成 Tensor 和 Flow 两部分来解读。

Tensor（张量）

张量是矩阵向任意维度（Dimension）的推广，TensorFlow 的运算都是基于张量进行的。

- TensorFlow 的基本类型。

```
import tensorflow as tf
```

```
# 建立一个常数 Tensor
c = tf.constant(1)
# 建立一个变量 Tensor
v = tf.Variable(1)
print(c)  # 显示 Tensor 常数信息，Shape=()表示标量，dtype=int32 表示整数
print(v)  # 显示 Tensor 变量信息，Shape=()表示标量，dtype=int32 表示整数
```

结果如下：

```
tf.Tensor(1, shape=(), dtype=int32)
<tf.Variable 'Variable:0' shape=() dtype=int32, numpy=1>
```

- 零维张量称为“标量”。

```
x = tf.constant(4)
print(x)  # 显示 Tensor 常数信息，Shape=()表示标量，dtype=int32 表示整数
print("{} 维 Tensor".format(x.ndim))    # 显示 Tensor 的维度
```

结果如下：

```
tf.Tensor(4, shape=(), dtype=int32)
0 维 Tensor
```

- 一维张量称为“向量”。

```
x = tf.constant([1, 2, 3, 4, 5, 6])
print("{}维 Tensor ".format(x.ndim))    # 显示 Tensor 的维度
```

结果如下：

```
1 维 Tensor
```

- 二维张量称为“矩阵”。

```
x = tf.constant([[1, 2, 3], [4, 5, 6]])
print("{}维 Tensor ".format(x.ndim))    # 显示 Tensor 的维度
```

结果如下：

```
2 维 Tensor
```

Flow（流）

流解释为数据流动或计算。

TensorFlow 的运作方式是通过产生数据流图（Data Flow Graph）来进行运算，又称作“计算图”（Computation Graph）。计算图的节点（Node）用来表示数学运算，边（Edge）则表示节点间的关联性。图 2-5 所示为数学式 ReLU(XW+b)的计算图。

TensorFlow 1.x 以前的版本都是先产生计算图，再执行如上所述的运算，此架构被称为静态图。而后来的 TensorFlow 2.0 由于默认 Eager Execution 模式为执行模式，因此引入了动态图的机制，使得执行指令可以立即得到回复。

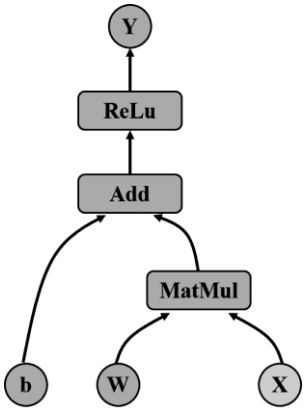


图 2-5　计算图

2.4　TensorFlow 2.0 的变化

TensorFlow 2.0 和前一版相比更简洁且容易上手。下面开始介绍在 TensorFlow 2.0 中当前主要推行的 6 个功能，其中 Eager Execution、Keras 和 tf.data 这 3 个功能将在接下来的 3 节详细介绍，而另外 3 个功能在后面的章节中再进行详细说明。

- Eager Execution

动态图模式，表示立即执行的意思，在该模式下执行运算会立即返回数值，让开发者的调试（Debug）更加便利和快速。

- Keras

TensorFlow 2.0 加入了 Keras 作为内建的高级 API 后有了更高的兼容性。内建 Keras 可通过 tf.keras 方式来调用其功能，具有指令简洁、可以自由组合且容易扩展的模块化 API 等特性，使得神经网络更容易搭建。

- tf.data

使用 tf.data 建立数据输入管道（Input Pipeline），速度更快，更简单。

- TensorFlow Hub

共享模型权重的 Library，可以从 Hub 上加载预先训练好的模型（Model），也可以将自己训练好的模型上传到 Hub，以分享给其他人。

- Distribution Strategy

新的 API 可用于更加轻松地完成在多台设备上的分布式训练，例如 CPU、GPU 或 TPU。

- SavedModel

TensorFlow 2.0 已经规范好网络模型的存储格式，使我们可以将训练好的网络模型放到想要执行的平台上，例如手机、树莓派或网页，同时也支持不同的程序设计语言，例如 C、Java、Go 或 C#等。

图 2-6 展示了当前 TensorFlow 2.0 已经能包办从训练模型到部署模型的流程，即读取数据、训练模型、保存模型以及把模型部署到各种设备的平台上。

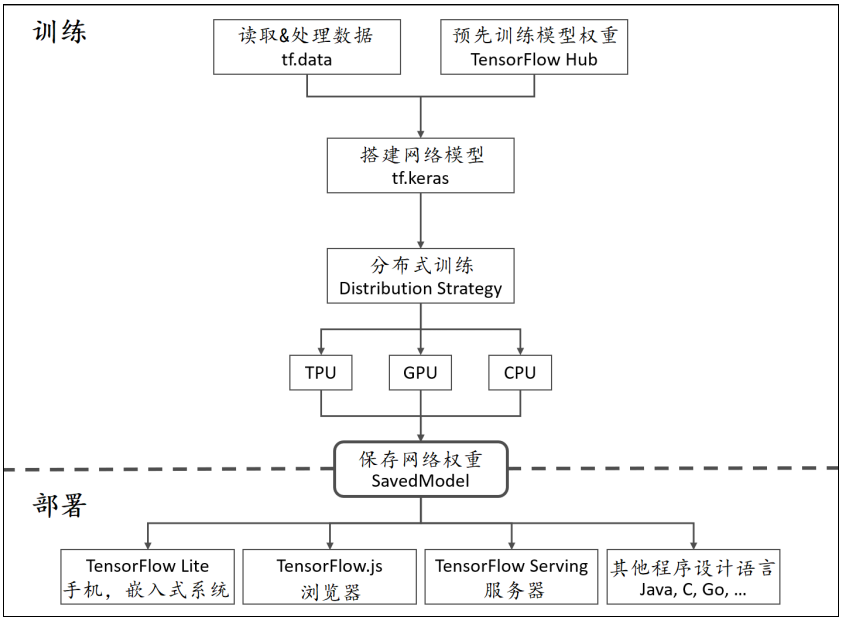


图 2-6　模型训练及部署流程所使用的 TensorFlow 模块

2.5　Eager Execution

2.5.1　Eager Execution 介绍

TensorFlow 引入了 Eager Execution 动态图模式，这个模式在 TensorFlow 2.0 中为默认的执行模式，一旦执行运算就会立刻返回数值，这有别于以往的静态图模式，需要建立计算图才能执行。如此使得 TensorFlow 更容易入门学习，也使得程序开发更为直观。

Eager Execution 模式的优点如下：

（1）立即返回数值，方便调试。
（2）不必定义计算图。
（3）不必初始化参数。
（4）无须 tf.Session.run 就可以返回运算结果。

TensorFlow 1.x 和 TensorFlow 2.0 的比较：

- TensorFlow 1.x code

tf.constant 在计算图中建立节点，并可以通过 sess.run 从中取得数值：

```
a = tf.constant(1)  # 建立一个常数 Tensor
print(a)            # 显示 Tensor 常数信息，Shape=()表示标量，dtype=int32 表示整数
# 建立一个 Session
sess = tf.Session()
# 通过 sess.run 取数值并显示出来
print("a = {}".format(sess.run(a)))
# 关闭 Session 释放资源
sess.close()
```

结果如下：

```
Tensor("Const_5:0", shape=(), dtype=int32)
a = 1
```

- TensorFlow 2.0 code

tf.constant 会直接返回数值，相对于 TensorFlow 1.x 省去了许多行程序代码：

```
a = tf.constant(1)  # 建立一个常数 Tensor
print(a)            # 显示 Tensor 常数信息，Shape=()表示标量，dtype=int32 表示整数
```

结果如下：

```
tf.Tensor(1, shape=(), dtype=int32)
```

2.5.2 TensorFlow 基本运算

Step 01 导入必要的套件。

```
import numpy as np       # 载入 NumPy 数学函数库
import tensorflow as tf  # 载入 TensorFlow 深度学习函数库
# 检查 Eager Execution 模式是否启动
print("Eager Execution 是否启动: {}".format(tf.executing_eagerly()))
```

结果如下：

```
Eager Execution 是否启动:True
```

Step 02 定义常数 Tensor。

```
a = tf.constant(3)    # 建立一个常数为 3 的 Tensor
b = tf.constant(4)    # 建立一个常数为 4 的 Tensor
# 显示 Tensor 数值(format 会直接将数值赋值给"{}"，不会有 shape、dtype 等信息)
print("a = {}".format(a))
print("b = {}".format(b))
```

结果如下：

```
a = 3
b = 4
```

Step 03 检查数据类型。

```
print(a)    # 显示 Tensor 常数信息，Shape=()表示标量，dtype=int32 表示整数
print(b)    # 显示 Tensor 常数信息，Shape=()表示标量，dtype=int32 表示整数
```

结果如下：

```
tf.Tensor(3, shape=(), dtype=int32)
tf.Tensor(4, shape=(), dtype=int32)
```

Step 04 基本运算。

```
c = a + b
print("a + b = {}".format(c))    # 显示 a+b 的结果
d = a * b
print("a * b = {}".format(d))    # 显示 a*b 的结果
```

结果如下：

```
a + b = 7
a * b = 12
```

Step 05 二维 Tensor 的运算。

在 Eager Execution 模式下，可以混合 Tensor 和 NumPy 进行运算。

```
# 建立二维张量的 Tensor，并且 dtype 为 float32
a = tf.constant([[1., 2.], [3., 4.]], dtype=tf.float32)
# 建立一个 NumPy array 数组，并且 dtype 为 float32
b = np.array([[1., 0.], [2., 3.]], dtype=np.float32)
print("a constant: {}D Tensor".format(a.ndim))

c = a + b
print("a + b = \n{}".format(c))    # 显示 a+b 的结果
# tf.matmul 为矩阵乘法
d = tf.matmul(a, b)
print("a * b = \n{}".format(d))    # 显示 a*b 的结果
```

结果如下：

```
a constant: 2D Tensor
a + b = [[2. 2.]
         [5. 7.]]
a * b = [[ 5.  6.]
         [11. 12.]]
```

Step 06 输出的结果为 Tensor 格式，可以将它转为 NumPy 格式。

```
print("NumpyArray:\n {}".format(c.numpy()))
```

结果如下：

```
NumpyArray:
 [[2. 2.]
 [5. 7.]]
```

说　明

在 TensorFlow 2.0 中，虽然有 Eager Execution 模式能够让 Tensor 格式支持 Python 基本运算（ex: for、if…else 等），但并非完全兼容，例如 OpenCV 或 Matplotlib 等套件的 API 输入格式有可能会不支持 Tensor 格式。而当遇到数据类型错误等问题时，最快的解决方法是将其转为 NumPy 格式，因为其历史悠久、高性能及受欢迎程度高，让 NumPy 格式能适用于大部分的 Python 套件。

Step 07 计算梯度：假设损失函数为 w^2。

```
w = tf.Variable([[1.0]])
# 正向传播会被记录到"tape"中
with tf.GradientTape() as tape:
    loss = w * w  # 损失函数为 w^2

# 反向传播"tape"计算梯度，∂loss^2/∂w = ∂w^2/∂w = 2w，因为 w=1，所以 grad=2

grad = tape.gradient(loss, w)
print(grad)
```

结果如下：

```
tf.Tensor([[2.]], shape=(1, 1), dtype=float32)
```

说　明

简单来说，一维的标量 x 的梯度（Gradient）就是计算 f(x)对 x 的微分，同理多维的向量 x 的梯度就是计算 f(x)对所有元素的偏微分。计算出来的梯度是有方向和大小的向量，而梯度指向的方向为局部最大值，所以在第 3 章介绍的梯度下降法就是往梯度的反方向（局部最小值）更新权重。

2.6 Keras

2.6.1 Keras 介绍

Keras 是 François Chollet 于 2014—2015 年开始编写的开源高级深度学习 API，主要用于快速搭建和训练网络模型。Keras 本身并没有运算能力，它是在 TensorFlow、CNTK 和 Theano 等深度学习开源套件上执行的，这些套件称为 Keras 的后端（Backend）。开始时 Keras 后端只支持 Theano，直到 2015 年底 TensorFlow 开源后，Keras 才搭建了 TensorFlow 后端，而今天 TensorFlow 已成为

Keras 常用的后端。总而言之，Keras 将这些深度学习套件封装为更容易使用的指令。

TensorFlow 2.0 将 Keras 收纳为内建的高级 API，因此不用额外安装 Keras 套件，直接通过 tf.keras 指令即可调用。tf.keras 和 Keras 的差别在于，tf.keras 能更为全面地支持 TensorFlow 的指令与模式，例如支持 Eager Execution、tf.data、TPU 训练等。

下面将介绍两种常用的网络搭建方法：

（1）Sequential Model（序贯模型）。

（2）Functional API（函数式模型，或称为函数式 API）。

接下来的范例会使用以下几种网络层，先简略介绍。暂时没有弄明白也没有关系，详细的理论和功能会在后面的章节讲述。以下范例展示的重点在于使用 Keras 搭建网络架构的方便性与灵活性。

（1）Dense：搭建全连接层的指令。

（2）Conv2d：搭建卷积层的指令。

（3）Flatten：将输入压平，重塑（Reshape）成一维张量，大多用于卷积层与全连接层之间。

（4）Add：将两个层的输出加在一起。

（5）Concatenate：将两个层的输出以指定的维度进行拼接（Concat），这种方法与 Add 方法相比保留了更多信息量，但计算量较大。

说　明

使用 tensorflow.keras.utils.plot_model 需要额外安装 pydot 和 graphviz 套件。

（1）在 Ubuntu 系统中的安装方法：启动 Terminal 程序并输入以下指令：

```
pip install pydot
apt-get install graphviz
```

（2）在 Windows 系统中的安装方法：

① 启动“命令提示符”程序并输入以下指令：

```
pip install pydot
```

② 到 https://graphviz.gitlab.io/_pages/Download/Download_windows.html 下载 Windows 系统下的安装包并安装，如图 2-7 所示。

图 2-7　graphviz 下载页面

说明（续）

③ 最后将 graphviz/bin 安装路径加到环境变量里。依次选择“控制面板”→“系统和安全”→“系统”→“高级系统设置”→“环境变量”，而后在“系统变量”窗格选择 Path，再单击“新建”按钮，就会出现如图 2-8 所示的界面。

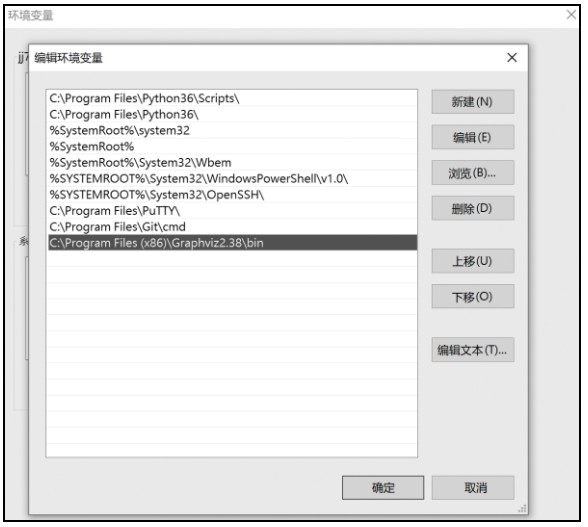


图 2-8　新建环境变量

2.6.2　序贯模型

序贯模型（Sequential Model）的搭建方法简单快速，可以解决大多数问题以及运用到各种应用中，例如手写数字识别、房价预测、评论分类等，基本上只要是回归问题或分类问题，就可以用序贯模型解决。但是，序贯模型的搭建方法有限制，必须逐层按序搭建网络，而且网络模型必须为单个输入层和单个输出层。图 2-9 所示为 Single Input and Output Model（单输入输出模型）。

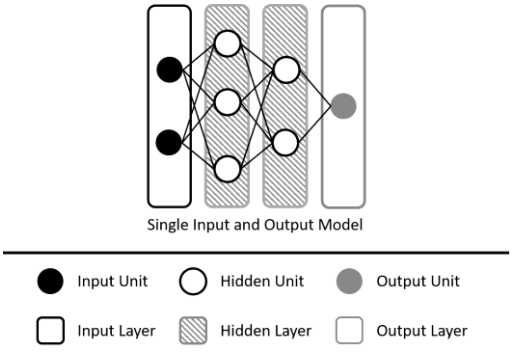


图 2-9　网络模型示意图

下面示范序贯模型的两种搭建方法，以分类问题为例：输入为 28×28 的图像并压平为 784 的一维向量，输出为 10 个元素的一维向量（分为 10 个类别），中间使用两个隐藏层，各拥有 64 个神经元，而隐藏层的激活函数为 ReLU，输出层为 Softmax。

Step 01 导入必要的套件。

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model
from IPython.display import Image
```

Step 02 建立模型（搭建方法有两种）。

- *方法一*

```
# 建立一个序贯模型
model = keras.Sequential(name='Sequential')
# 每次 model.add 会增加一层网络到模型中，第一层需要定义输入尺寸（input_shape）
# 第一个数值表示输出个数，即该层神经元的个数
model.add(layers.Dense(64, activation='relu', input_shape=(784,)))
model.add(layers.Dense(64, activation='relu'))
# 最后一层会被当作模型的输出层
model.add(layers.Dense(10, activation='softmax'))
```

- *方法二*

```
# 可以将所有网络层放到一个列表（list）中，并作为 tf.keras.Sequential 的参数
# 而这个列表同样有顺序性，第一个需要定义输入大小，最后一个为输出层
model = tf.keras.Sequential([
                    layers.Dense(64, activation='relu', input_shape=(784,)),
                    layers.Dense(64, activation='relu'),
                    layers.Dense(10, activation='softmax')])
```

Step 03 显示网络模型。

```
# 产生网络拓扑图
plot_model(model, to_file='Functional_API_Sequential_Model.png')

# 显示出网络拓扑图
Image('Functional_API_Sequential_Model.png')
```

结果如图 2-10 所示。

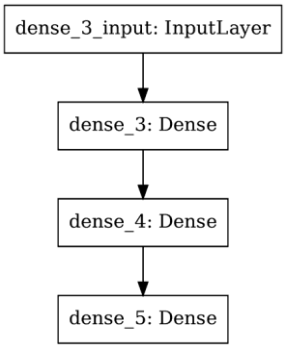


图 2-10　执行结果

2.6.3 Functional API

Keras 中的 Functional API（函数式 API）是建立网络模型的另一种方式，它提供了更多的灵活性，能够建立更复杂的模型，例如多输入单输出模型、单输入多输出模型和多输入多输出模型等，如图 2-11 所示。

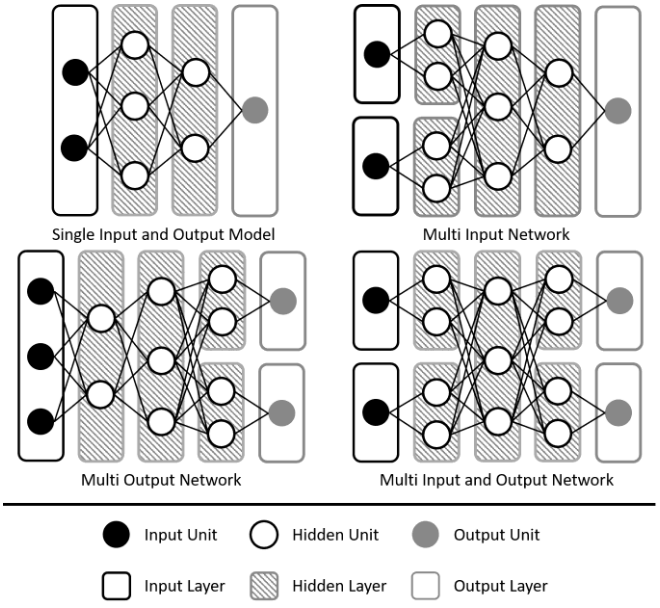


图 2-11　网络模型示意图

Functional API 在使用上非常灵活，后面介绍的高级应用都能使用 Functional API 方式搭建模型来实现。下面介绍几种高级应用。

（1）物体检测（Object Detection）。
（2）图像分割（Image Segmentation）。
（3）生成式对抗网络（Generative Adversarial Network）。

下面将按序介绍几种网络架构以及应用的场景。

（1）Single Input and Output Model：单输入单输出模型。
（2）Multi Input Model：多输入单输出模型。
（3）Multi Output Model：单输入多输出模型。
（4）Multi Input and Output Model：多输入多输出模型。

Single Input and Output Model

单输入单输出模型，例如输入为 28×28 的图像并压平为 784 的一维向量，输出为 10 个元素的一维向量（分为 10 个类别），中间使用两个隐藏层，各拥有 64 个神经元，而隐藏层的激活函数为 ReLU，输出层为 Softmax。

```
    # 不同于序贯模型和 Functional API 需要建立输入层
    inputs = keras.Input(shape=(784,), name='Input')
    h1 = layers.Dense(64, activation='relu', name='hidden1')(inputs)
    h2 = layers.Dense(64, activation='relu', name='hidden2')(h1)
    outputs = layers.Dense(10, activation='softmax', name='Output')(h2)

    # 这个 keras.Model 会自动将输入层到输出层所有经过的各层连接起来建立成网络
    model = keras.Model(inputs=inputs, outputs=outputs)

    plot_model(model, to_file='Functional_API_Single_Input_And_Output
_Model.png')
    Image('Functional_API_Single_Input_And_Output_Model.png')
```

结果如图 2-12 所示。

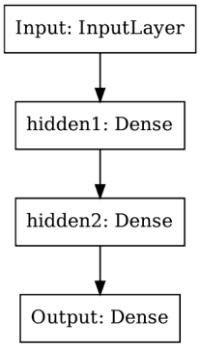


图 2-12　执行结果

Multi Input Model

多输入单输出模型，例如商品价格预测为两个输入（商品图片和商品品牌）和一个输出（价格预测），商品图片(128, 128, 3)输入经过 3 个隐藏层，商品品牌(1,)输入经过一个隐藏层，结合两个信息后再经过一个隐藏层，而输出为价格预测(1,)。

```
    # 网络模型输入层
    img_input = keras.Input(shape=(128, 128, 3), name='Image_Input')
    info_input = keras.Input(shape=(1, ), name='Information_Input')

    # 网络模型隐藏层
    h1_1 = layers.Conv2D(64, 5, strides=2, activation='relu',
                         name='hidden1_1')(img_input)
    h1_2 = layers.Conv2D(32, 5, strides=2, activation='relu',
                         name='hidden1_2')(h1_1)
    h1_2_ft = layers.Flatten()(h1_2)
    h1_3 = layers.Dense(64, activation='relu', name='hidden1_3')(info_input)
    concat = layers.Concatenate()([h1_2_ft, h1_3])
    h2 = layers.Dense(64, activation='relu', name='hidden2')(concat)

    # 网络模型输出层
    outputs = layers.Dense(1, name='Output')(h2)
```

```
# 建立网络模型
model = keras.Model(inputs=[img_input, info_input], outputs=outputs)

# 显示网络模型架构
plot_model(model, to_file='Functional_API_Multi_Input_Model.png')
Image('Functional_API_Multi_Input_Model.png')
```

结果如图 2-13 所示。

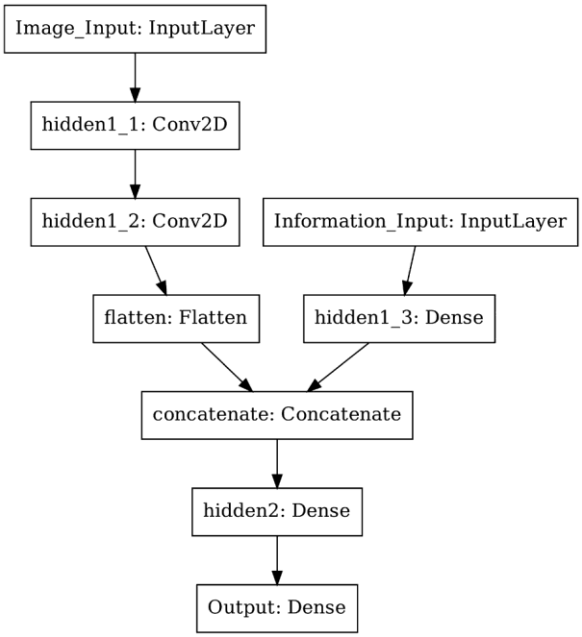


图 2-13　执行结果

Multi Output Model

单输入多输出模型，例如人像识别为一个输入（人物照片）和两个输出（年龄和性别），人物照片(128, 128, 3)输入经过 4 个隐藏层，而输出为年龄(1,)和性别(1,)两种不同的信息。

```
# 网络模型输入层
inputs = keras.Input(shape=(28, 28, 1), name='Input')

# 网络模型隐藏层
h1 = layers.Conv2D(64, 3, activation='relu', name='hidden1')(inputs)
h2 = layers.Conv2D(64, 3, strides=2, activation='relu', name='hidden2')(h1)
h3 = layers.Conv2D(64, 3, strides=2, activation='relu', name='hidden3')(h2)
flatten = layers.Flatten()(h3)

# 网络模型输出层
age_output = layers.Dense(1, name='Age_Output')(flatten)
gender_output = layers.Dense(1, name='Gender_Output')(flatten)

# 建立网络模型
```

```
model = keras.Model(inputs=inputs, outputs=[age_output, gender_output])

# 显示网络模型架构
plot_model(model, to_file='Functional_API_Multi_Output_Model.png')
Image('Functional_API_Multi_Output_Model.png')
```

结果如图 2-14 所示。

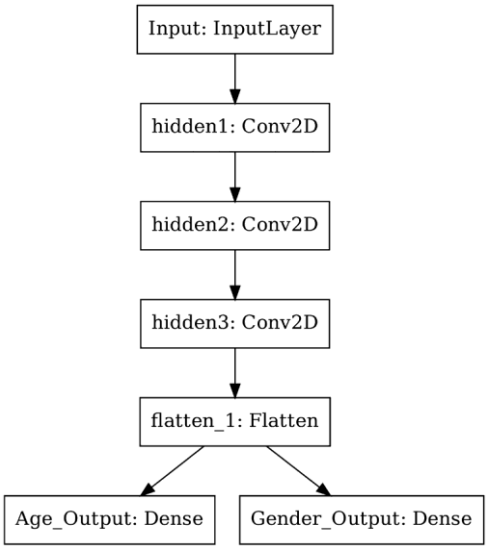


图 2-14　执行结果

Multi Input and Output Model

多输入多输出模型，例如天气预测为两个输入（卫星云图和气候信息）和 3 个输出（概率、温度和湿度），卫星云图(256×256×3)输入经过 3 个隐藏层，气候信息(10,)输入经过一个隐藏层并结合两个信息，而输出为气候信息，如降雨概率(1,)、温度(1,)和湿度(1,) 3 种不同的信息。

```
# 网络模型输入层
image_inputs = keras.Input(shape=(256, 256, 3), name='Image_Input')
info_inputs = keras.Input(shape=(10, ), name='Info_Input')

# 网络模型隐藏层(Image Input)
h1 = layers.Conv2D(64, 3, activation='relu', name='hidden1')(image_inputs)
h2 = layers.Conv2D(64, 3, strides=2, activation='relu', name='hidden2')(h1)
h3 = layers.Conv2D(64, 3, strides=2, activation='relu', name='hidden3')(h2)
flatten = layers.Flatten()(h3)
# 网络模型隐藏层(Information Input)
h4 = layers.Dense(64)(info_inputs)
concat = layers.Concatenate()([flatten, h4])  # 结合 Image 和 Information 特征

# 网络模型输出层
weather_outputs = layers.Dense(1, name='Output1')(concat)
temp_outputs = layers.Dense(1, name='Output2')(concat)
humidity_outputs = layers.Dense(1, name='Output3')(concat)
```

```
    # 建立网络模型
    model = keras.Model(inputs=[image_inputs, info_inputs],
                        outputs=[weather_outputs, temp_outputs, humidity_outputs])

    # 显示网络模型架构
    plot_model(model, to_file='Functional_API_Multi_Input_And_Output
_Model.png')
    Image('Functional_API_Multi_Input_And_Output_Model.png')
```

结果如图 2-15 所示。

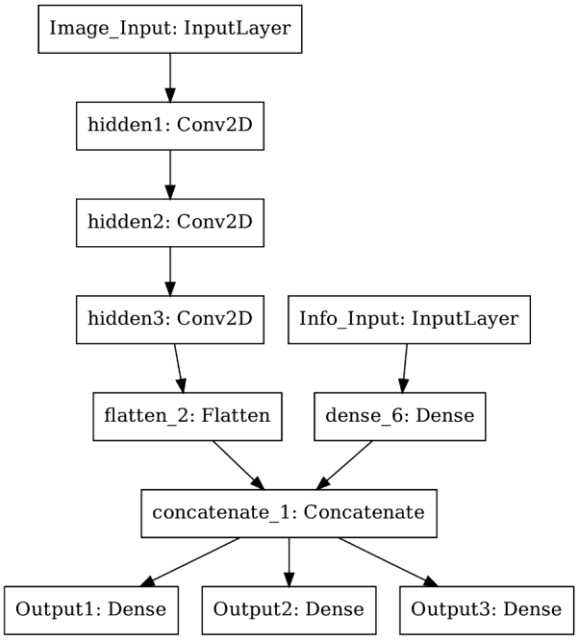


图 2-15　执行结果

2.7　tf.data

2.7.1　tf.data 介绍

从读取数据到数据传入加速设备（GPU 或 TPU）的流程被称为输入管道（Input Pipeline）。输入管道可以分为以下 3 个步骤：

Step 01　提取（Extraction）。

从存储的地方（可以是 SSD、HDD 或远程存储位置）读取数据。

Step 02　转换（Transformation）。

使用 CPU 进行数据预处理，例如对图像进行翻转、裁剪、缩放和正则化等。

Step 03　载入（Loading）。

将转换后的数据加载到机器学习模型的加速设备。

上面这 3 个步骤主要是设备读取数据和 CPU 预处理在消耗时间，如果没有妥善地分工处理，就会造成当 CPU 在准备数据时，GPU 在等待训练数据的产生（GPU 处于空闲状态）；反之，当 GPU 在训练时，CPU 则处于空闲状态，如图 2-16 所示。如此，训练时间就会增加很多。

图 2-16　一般训练时的情况

TensorFlow 提供 tf.data API，可以帮助使用者打造灵活有效的输入管道，轻松处理大量数据、不同数据格式及复杂的转换。而且通过使用 tf.data.Dataset.prefetch,一行指令就可以让生成数据与训练数据同时进行，进而提升训练效率，如图 2-17 所示。

图 2-17　CPU 和 GPU 可同步进行

倘若输入管道的执行时间远比训练时间久，将发生如图 2-18 所示的情况，造成 GPU 或 TPU 加速器无法发挥全部的运算力，通常这种情况可能是读取文件太大或数据预处理太久造成的。

图 2-18　读取与处理时间太长

上述问题可以使用 CPU 多线程来解决，只需在调用 map 方法时加入 num_parallel_calls 设置，即可启用并行处理数据的功能。通常 num_parallel_calls 会设置成计算机的核心数，图 2-19 所示为改善后的工作情况。

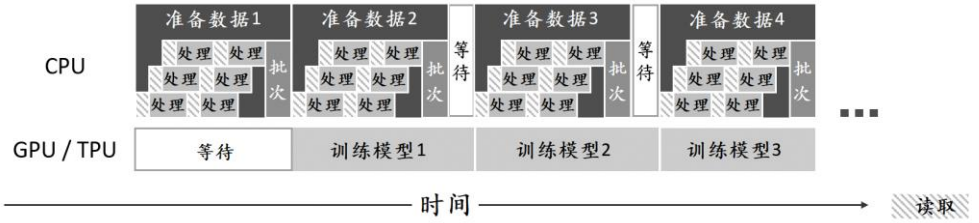


图 2-19　CPU 能以多进程方式准备数据

2.7.2　基本操作

了解了 tf.data 之后，下面将通过几个例子说明如何建立 Dataset、设置 Dataset 和提取数据（注意，下面的程序请依序执行）。

- tf.data.Dataset.from_tensors

使用 from_tensors 建立 Dataset，示例代码如下：

```
dataset = tf.data.Dataset.from_tensors(tf.constant([1, 2, 3, 4, 5, 6, 7,
          8, 9, 10], shape=(10, )))
# 显示 Dataset 信息，形状为(10, )代表一次读取的形状，types 为整数
print(dataset)
```

结果如下：

```
<TensorDataset shapes: (10,), types: tf.int32>
```

- tf.data.Dataset.from_tensor_slices

使用 from_tensor_slices 建立 Dataset，示例代码如下：

```
x_data = tf.data.Dataset.from_tensor_slices(tf.constant([0, 1, 2, 3, 4, 5, 6,
         7, 8, 9]))
# 显示 Dataset 信息，形状为()代表一次读取 1 个数值，types 为整数
print(x_data)

y_data = tf.data.Dataset.from_tensor_slices(tf.constant([0, 2, 4, 6, 8, 10,
12, 14, 16, 18]))
# 显示 Dataset 信息，形状为()代表一次读取 1 个数值，types 为整数
print(y_data)
```

结果如下：

```
<TensorSliceDataset shapes: (), types: tf.int32>
<TensorSliceDataset shapes: (), types: tf.int32>
```

- for loop

读取数据，示例代码如下：

```
for data in dataset:
    print(data)
```

结果如下：

```
tf.Tensor([ 1  2  3  4  5  6  7  8  9 10], shape=(10,), dtype=int32)
```

```
for data1, data2 in zip(x_data, y_data):
    print('x: {}, y: {}'.format(data1, data2))
```

结果如下：

```
x: 0, y: 0
x: 1, y: 2
x: 2, y: 4
x: 3, y: 6
x: 4, y: 8
x: 5, y: 10
x: 6, y: 12
x: 7, y: 14
x: 8, y: 16
x: 9, y: 18
```

- take

通过参数可以指定读取数据的数量，示例代码如下：

```
for data in dataset.take(1):
    print(data)
```

结果如下：

```
tf.Tensor([ 1  2  3  4  5  6  7  8  9 10], shape=(10,), dtype=int32)
```

```
for data1, data2 in zip(x_data.take(5), y_data.take(5)):
    print('x: {}, y: {}'.format(data1, data2))
```

结果如下：

```
x: 0, y: 0
x: 1, y: 2
x: 2, y: 4
x: 3, y: 6
x: 4, y: 8
```

如果指定读取数量超过 Dataset 所含有的数据量，就会读取出 Dataset 中所有的数据。

```
for data1, data2 in zip(x_data.take(12), y_data.take(12)):
    print('x: {}, y: {}'.format(data1, data2))
```

结果如下：

```
x: 0, y: 0
x: 1, y: 2
x: 2, y: 4
x: 3, y: 6
x: 4, y: 8
```

```
x: 5, y: 10
x: 6, y: 12
x: 7, y: 14
x: 8, y: 16
x: 9, y: 18
```

- tf.data.Dataset.zip

将多个 Dataset 打包成一个，示例代码如下：

```
dataset = tf.data.Dataset.zip((x_data, y_data))
print(dataset)
```

结果如下：

```
<ZipDataset shapes: ((), ()), types: (tf.int32, tf.int32)>
```

- map

可以使用 map 来转换数据，示例代码如下：

```
tf.data.Dataset.range(10).map(lambda x: x*2)
```

结果如下：

```
<MapDataset shapes: (), types: tf.int64>
```

- 命名

以字典方式为 elements 的组件命名，示例代码如下：

```
dataset = tf.data.Dataset.zip(
   {"x": tf.data.Dataset.range(10),
    "y": tf.data.Dataset.range(10).map(lambda x: x*2)})
print(dataset)
```

结果如下：

```
<ZipDataset shapes: {y: (), x: ()}, types: {y: tf.int64, x: tf.int64}>
```

```
for data in dataset.take(10):
    print('x: {}, y: {}'.format(data['x'], data['y']))
```

结果如下：

```
x: 0, y: 0
x: 1, y: 2
x: 2, y: 4
x: 3, y: 6
x: 4, y: 8
x: 5, y: 10
x: 6, y: 12
x: 7, y: 14
x: 8, y: 16
x: 9, y: 18
```

- 设置每一批（batch）读取的数据量

```
dataset = tf.data.Dataset.zip(
   {"x": tf.data.Dataset.range(10),
    "y": tf.data.Dataset.range(10).map(lambda x: x*2)}).batch(2)
for data in dataset.take(5):
    print('x: {}, y: {}'.format(data['x'], data['y']))
```

结果如下：

```
x: [0 1], y: [0 2]
x: [2 3], y: [4 6]
x: [4 5], y: [ 8 10]
x: [6 7], y: [12 14]
x: [8 9], y: [16 18]
```

- shuffle

Dataset 数据会被加载到缓冲区中，并从缓冲区中随机选取数据出来，取出数据产生的空位会从新的数据找替补。buffer_size 设置的是缓冲区的大小，最好的设置是大于或等于整个 Dataset 数据的个数。示例代码如下：

```
dataset = dataset.shuffle(10)
for data in dataset.take(5):
    print('x: {}, y: {}'.format(data['x'], data['y']))
```

结果如下：

```
x: [0 1], y: [0 2]
x: [6 7], y: [12 14]
x: [4 5], y: [ 8 10]
x: [8 9], y: [16 18]
x: [2 3], y: [4 6]
```

- repeat

当 Dataset 的数据读取完后，就会读取不到数据，通过设置 repeat(n)可以重复读取 Dataset n 次，示例代码如下：

```
for data in dataset.take(10):
    print('x: {}, y: {}'.format(data['x'], data['y']))

print('-' * 50)
dataset = dataset.repeat(2)
for data in dataset.take(10):
    print('x: {}, y: {}'.format(data['x'], data['y']))
```

结果如下：

```
x: [0 1], y: [0 2]
x: [6 7], y: [12 14]
x: [4 5], y: [ 8 10]
x: [8 9], y: [16 18]
```

```
x: [2 3], y: [4 6]
--------------------------------------------------
x: [6 7], y: [12 14]
x: [0 1], y: [0 2]
x: [4 5], y: [ 8 10]
x: [8 9], y: [16 18]
x: [2 3], y: [4 6]
x: [8 9], y: [16 18]
x: [4 5], y: [ 8 10]
x: [6 7], y: [12 14]
x: [0 1], y: [0 2]
x: [2 3], y: [4 6]
```

第 3 章

回归问题

学习目标

- 了解神经网络的简史、基本概念及原理
- 了解神经网络权重的更新方法
- 认识 Kaggle 平台
- 运用全连接神经网络完成房价预测系统
- 认识 TensorBoard
- 解决过拟合问题

3.1 深度神经网络

3.1.1 神经网络简史

早在 30 年前，神经网络（Neural Network）[1]-[2] 技术就已经存在，并且盛行过一段时间，不过在 1980—2000 年转而流行支持向量机（Support Vector Machine，SVM）[3-4]，主要的原因是当时能够搭建的神经网络层数不多（1~3 层），导致神经网络所能学习到的特征有限，造成模型性能不佳，这些层数不多的网络被称为浅层神经网络（Shallow Neural Network）。近年来，随着计算机各方面技术的提升，以至于能搭建更庞大的神经网络，因而有了重大的突破，这种更多层数的网络被称为深度神经网络（Deep Neural Network），在性能上远远超越以往的人工智能方法，使神经网络再度受到重视。图 3-1 给出了单隐藏层的浅层神经网络和多隐藏层的深度神经网络的网络架构。

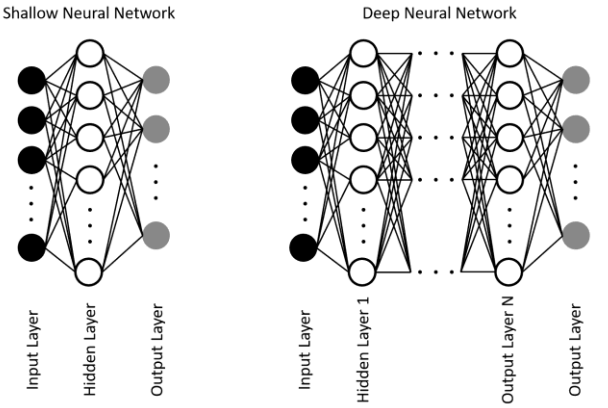


图 3-1　浅层神经网络和深度神经网络

神经网络架构能有突破性发展，归功于以下几个原因：

运算能力

以往神经网络都是使用 CPU 进行运算的，现在则多使用 GPU 或 TPU 进行运算，执行性能提高了 10~100 倍。这种进展都要归功于 2012 年深度学习之父 Geoffrey Hinton 的学生使用 NVIDIA 的 GPU 训练了 8 层的网络 AlexNet[5]，这个神经网络在当时的图像识别比赛 ImageNet[6] 中以将近 10%的领先优势拿下冠军。至此，大家才意识到 GPU 惊人的运算能力以及深度神经网络的强大。

数据数量

深度神经网络如果没有足够的训练数据是无法训练起来的，在 20~30 年前，一般不可能拥有大量的数据，原因是当时的存储设备昂贵、容量少以及凭借个人力量难以收集。而现在存储设备价格下降、网络及云端设备盛行，再加上有许多开放式的数据库供研究人员使用，故而当时人们所面临的困境不复存在。

激活函数

激活函数（Activation function）又称为活化函数，以前隐藏层所使用的激活函数大多是 Sigmoid，网络层数太深会导致梯度消失（Vanishing Gradient），使得网络难以更新，因此 X. Glorot 等人在 2011 年提出使用 ReLU[7] 函数，可以有效避免梯度消失问题。

3.1.2　神经网络原理

神经网络可以将其拆为输入层（Input Layer）、隐藏层（Hidden Layer）及输出层（Output Layer），网络中的每一层都有许多神经元（Neuron），各层之间神经元与神经元的连接都是依赖权重（Weight）和偏差（Bias）的，如图 3-2 所示的数学方程式，图中一个圆圈代表一个神经

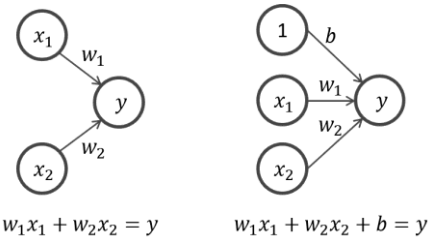


图 3-2　层与层之间的连接

元，左图和右图两边的差别在于有无偏差项。

除了权重和偏差外，网络层的输出还需要加上激活函数，如果没有加上激活函数，那么无论把网络增加多深，网络所能学习到的也只是一个线性方程式，常见的激活函数有 ReLU、Sigmoid、Tanh。激活函数通常会放在每一层的输出后方，如图 3-3 所示。

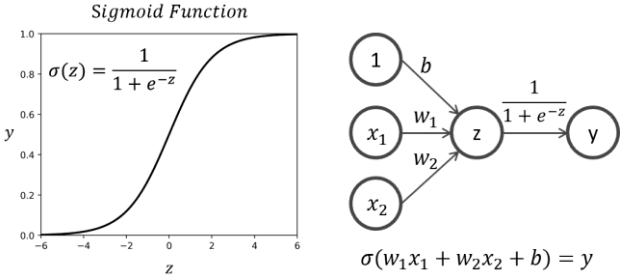


图 3-3　每层的输出后面接上 Sigmoid 激活函数

3.1.3　全连接

在神经网络中，如果每个神经元都与前面一层的所有神经元相连接，就称为全连接（Fully Connected）。图 3-4 所示为三层隐藏层的全连接神经网络，且每一层隐藏层的输出都使用 ReLU 激活函数。本章将实现一个全连接层的房价预测模型。

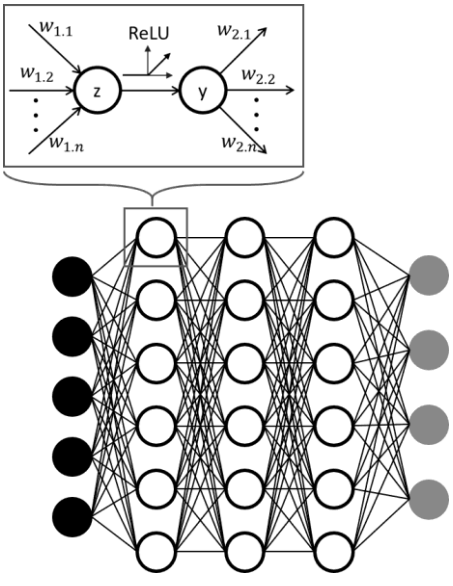


图 3-4　全连接神经网络

3.1.4　损失函数 MSE 和 MAE

损失函数（Loss Function）用来计算模型预测值与预期输出值之间的相似程度，在训练过程中需要将预测值与预期输出值的相似程度最小化。针对不同的问题可以设计合适的损失函数，机器学

习中常见的问题有回归问题（Regression Problem）、二分类问题（Binary Classification Problem）和多分类问题（Multiclass Classification Problem）。下面介绍这 3 种问题以及常用的损失函数。

- 回归问题

通常使用均方误差（Mean Squared Error，MSE）或平均绝对误差（Mean Absolute Error，MAE）来评估预期输出值与预测值的相似程度。

- 二分类问题

通常使用二分类交叉熵（Binary Cross-Entropy，BCE）来评估预期输出值与预测值的相似程度。

- 多分类问题

通常使用多分类交叉熵（Categorical Cross-Entropy，CCE）来评估预期输出值与预测值的相似程度。

本章的主题——房价预测属于回归问题，因此可以使用均方误差和平均绝对误差作为损失函数。下面将介绍这两个损失函数的公式与差异。

➢ 均方误差

$$\text{MSE}=\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}$$

➢ 平均绝对误差

$$\text{MAE}=\frac{\sum_{i=1}^{N}\left|y_i-\hat{y}_i\right|}{N}$$

y：预期输出值。

$\hat{y}$：深度学习模型的预测值。

N：一个批量的数据量。

MSE 与 MAE 都是计算模型预测值（$\hat{y}$）与预期输出值（y）的误差，不过一个将误差值取平方，另一个将误差值取绝对值，相同的预测结果经过不同损失函数的计算会得到不同的损失值，如图 3-5 所示。预测结果为-1~1 时，MAE 损失值较大，而预测值大于 1 或小于-1 时，MSE 损失值较大。采用这两种损失函数会有不同的训练结果，本章范例以 MSE 作为训练的损失函数，读者可以自行更改为 MAE 比较两者的训练差异。

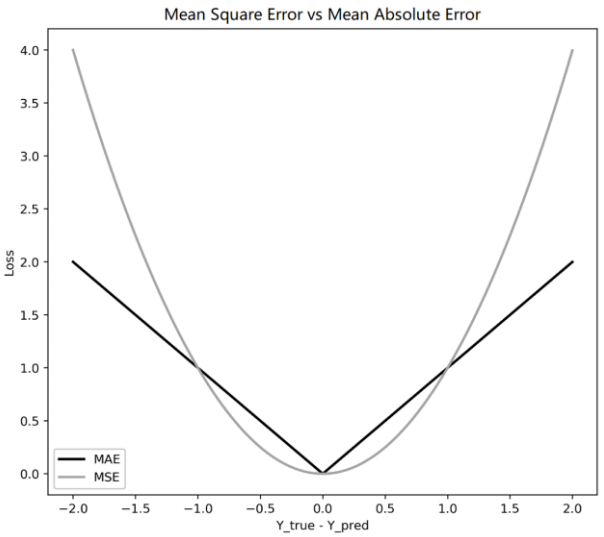


图 3-5　MSE 和 MAE 的损失函数误差比较

3.1.5　神经网络权重更新

梯度下降（Gradient Descent，GD）

神经网络的目的是找到合适的权重与偏差值使损失函数（Loss Function）的损失值（Loss）越来越小，寻找参数的过程称为“学习”，梯度下降是常用的优化算法，通过计算梯度并沿着梯度反方向移动，进而减少网络预测值与样本标注值之间的误差。该算法可以比喻为一个人在山上寻找下山的路，以人当前所处的位置为基准，通过每次寻找周边最陡峭的山路，然后朝着下坡的方向走，反复采用这个方法，每次走一段距离，直到抵达山谷，如图 3-6 所示。网络参数通过梯度下降得到优化，逐步降低损失值，以找到区域最小的损失值，使网络的预测结果值最接近预期输出值，公式如下：

$$W = W - \eta \frac{\partial L}{\partial W}$$

L：损失函数。

η：学习率。

W：权重。

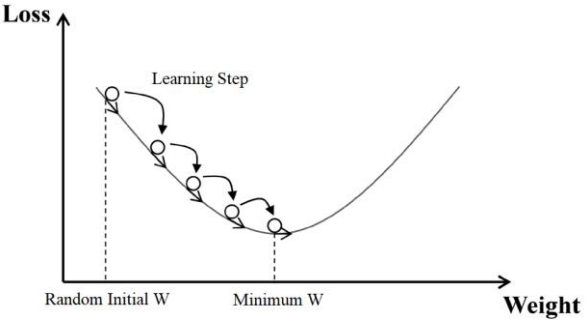


图 3-6　梯度下降示意图

事实上，梯度下降每一次的更新并不见得是朝着损失函数的最小值更新，而是朝着当前能减少最多损失函数误差的方向更新，如图 3-7 所示。因此，当我们训练到损失值无法再降低时，这时所到达的点并非全局最小值（Global Minimum），而是局部最小值（Local Minimum）。

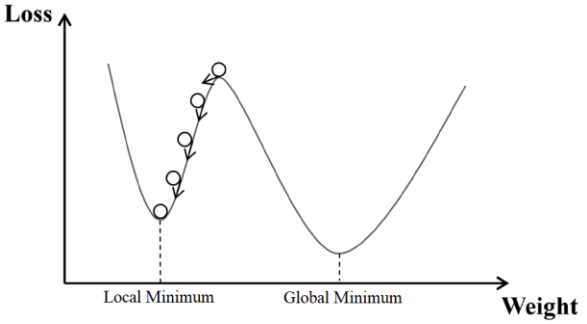
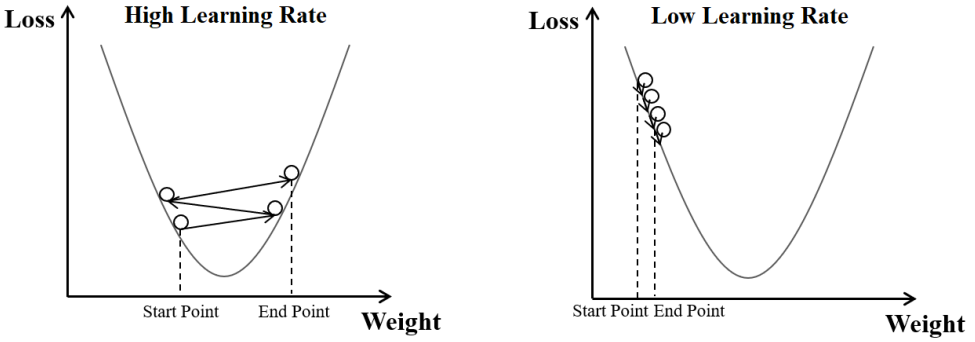


图 3-7　权重将会朝着当前能减少最多损失函数误差的方向更新

学习率（Learning Rate）

学习率的大小决定优化算法的优化步伐，即图 3-6 中的 Learning Step。如果学习率过大，每次网络参数更新的步长过大，那么很可能会跃过最佳值（Minimum，最小损失值）并且产生震荡现象，如图 3-8（a）所示；反之，如果学习率过小，那么优化的效率可能过低，经过长时间训练仍无法找到最佳值，如图 3-8（b）所示。



（a）学习率过大　　（b）学习率过小

图 3-8　不同学习率的优化过程

最后，梯度更新法也分很多种，前面提到的梯度下降是一次使用全部的训练数据计算损失函数的梯度，并更新一次权重，如果要更新 N 次，就要计算整个训练数据 N 遍，这种方法十分花费时间且没有效率。因此出现了随机梯度下降法（Stochastic Gradient Descent，SGD），一次计算一个批量（Batch）数据的梯度值并更新一次权重，例如设置一个批量大小为 64，每一次计算会从训练数据中抽出 64 笔数据计算梯度。另外，还有许多改进方法，如 Momentum[8]、AdaGrad[9]和 Adam[10]等。Momentum 加入动量的概念，AdaGrad 会根据梯度来调整学习率，Adam 可以视为 Momentum 和 AdaGrad 的结合。面对大多数问题，Adam 优化器都可以达到不错的效果。

3.1.6　神经网络训练步骤

图 3-9 所示为神经网络训练示意图，后面的练习都会围绕这张图的 5 部分实现，使读者可以深入浅出地实现具体的流程。

神经网络训练的 5 部分说明如下：

❶ 准备训练数据（Training Data），将其分为训练数据（Input X）和预期输出值的标记答案（Input Y）。

❷ 搭建神经网络模型（Model），用以预测数值（Prediction，ŷ：预测值）。

❸ 损失函数（Loss Function）计算模型预测值（Prediction，ŷ：预测值）与标记答案（Input Y，y：预期输出值）之间的误差，本章的范例将使用均方误差（Mean Squared Error）作为损失函数来训练网络模型。

❹ 优化器（Optimizer）决定学习过程如何进行，常见的方法有 SGD、Adam、RMSprop 等，本范例使用 Adam 优化器。

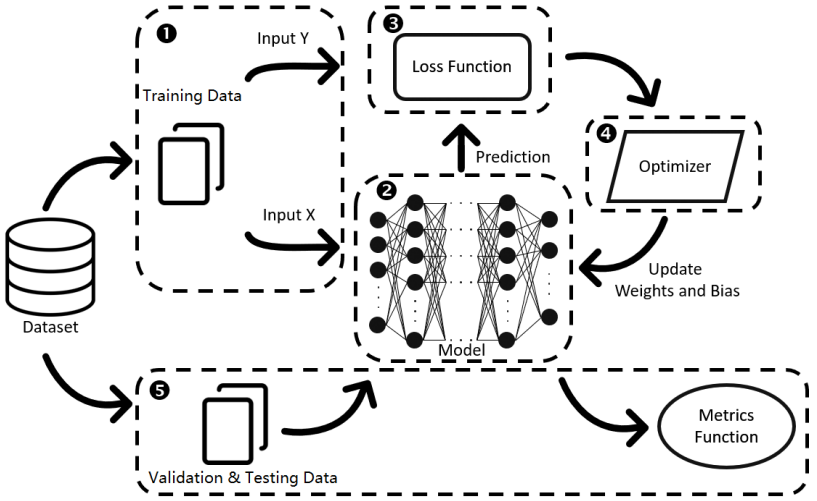


图 3-9　神经网络训练示意图

❺ 准备验证数据（Validation Data）和测试数据（Testing Data）来让网络模型进行预测，并通过评价指标函数（Metrics Function，又称为评价函数）来评价模型的好坏。

- 验证数据：模型训练过程中会调整超参数（网络层数、神经元个数等），以获得最佳模型，在调整过程中利用验证数据的性能作为调整依据。
- 测试数据：大多数比赛会公开训练数据和验证数据，而测试数据会自行保留，作为最后评估模型的主要依据。

说　明

数据通常划分为训练数据、验证数据和测试数据。

测试数据通常于评判一个神经网络算法在真实世界下的正确率和泛化能力强弱的指标。然而，若研究者直接通过测试数据的表现来对网络架构进行修改，则往往能够获得一个在训练数据及测试数据中都有良好表现的网络模型，但应用于真实世界却会表现不佳，此种情况称为元过拟合（Meta-Overfitting）[11]。为了避免研究人员针对测试指标直接进行设计而导致元过拟合，通常会从训练数据中切分出验证数据来作为架构修改的指针，如此测试数据所给出的正确率才会更加符合网络在真实数据下的表现。

说　明

评价指标函数（Metrics Function）

评价指标函数和损失值都用来评估训练模型的好坏程度，有时候评价指标函数和损失值对训练模型的评估结果一致，有时候则不一致。以二分类（Binary Classification）为例，输入两笔数据答案为[1, 0]，并使用两个网络模型进行预测，使用 MAE 计算损失值（计算预测值与标记答案值的误差）和指标值（正确预测数量，预测输出值小于 0.5 视为 0，大于 0.5 则视为 1）：

模型 1：预测输出值为[0.5, 0.5]，得到 MAE 损失值总和为 1.0，正确预测 1 笔数据。

模型 2：预测输出值为[0.7, 0.3]，得到 MAE 损失值总和为 0.6，正确预测 2 笔数据。

说　明
超参数（Hyperparameter） 超参数是指训练网络模型之前设置的参数，例如批量、学习率、训练周期、网络层数和网络节点数等，这些参数的选择会对训练结果产生影响。第 8 章将会介绍 TensorFlow 辅助调整超参数的工具。

3.2　Kaggle 介绍

Kaggle 是世界著名的竞赛平台，企业和研究学者可以在上面发布数据集或举办竞赛，无论你来自哪里，只要对比赛感兴趣就可以参加，有时候比赛还提供奖金。另外，值得注意的是，Kaggle 除了是竞赛网站之外，也是一个社群平台，参赛者之间可以互相讨论、组队或分享研究成果，而且为了鼓励参赛者多参与讨论，主办单位将奖励分享实验结果并获得最多认可的参赛者。Kaggle 竞赛页面表明当前有高达 19 个竞赛正在进行，如图 3-10 所示。

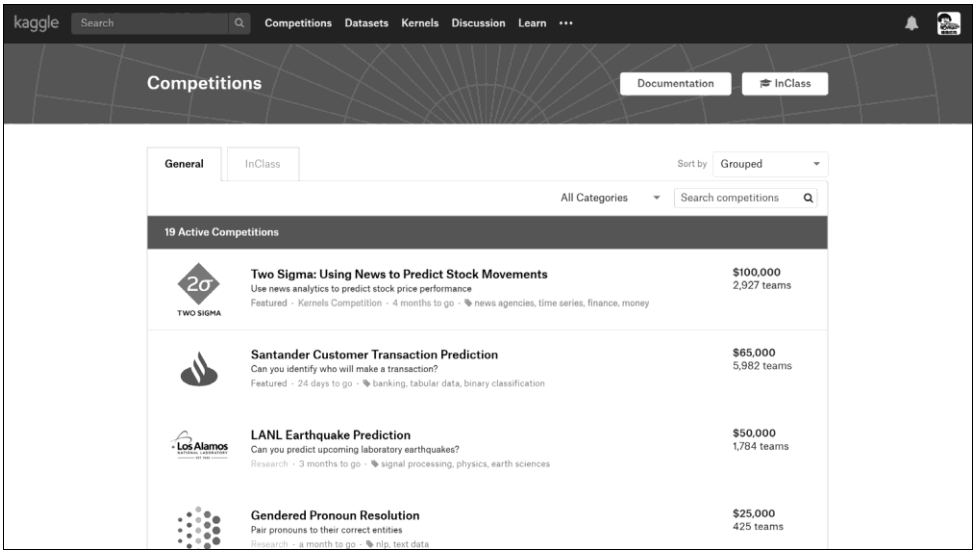


图 3-10　Kaggle 竞赛页面

Kaggle 还有一个很棒的地方是设有 Datasets 专区，里面的数据集都已经过整理并且提供下载，是一个非常好的开源数据集。有些研究员或数据科学家会在数据集下分享自己的研究成果，这使我们学习起来更加如虎添翼。本章的实验数据就是使用 Kaggle 的数据集。

3.3　实验一：房价预测模型

本节的主题为“房价预测”，希望通过神经网络模型来预测房屋的价格。网络模型的输入为房屋的信息，例如面积、楼层或房龄等信息，通过对这些输入信息的分析，网络模型会输出预测的房屋价格。网络模型训练使用均方误差作为损失函数，Adam 作为模型优化器。

3.3.1　数据集介绍

本章的范例使用 Kaggle 的“House Sales in King County, USA”数据集。访问网址：https://www.kaggle.com/harlfoxem/housesalesprediction，并单击 Download 按钮，将压缩文件下载到当前程序代码所保存的目录下，再解压缩这个文件，如图 3-11 所示。

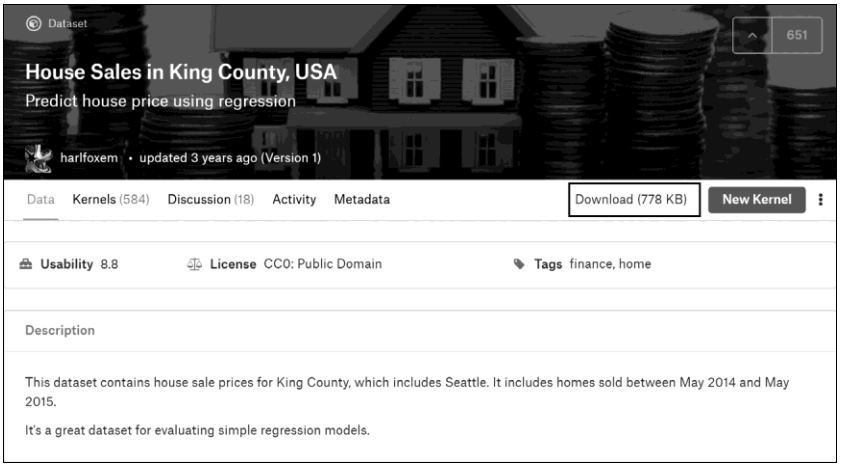


图 3-11　House Sales in King County 数据集下载

此数据集共有 21613 笔房屋数据，每一笔数据有 21 个不同的信息，分别表示以下含义：

- id：房子的标识号。
- date：房屋出售日期。
- price：房屋价格（目标）。
- bedrooms：卧室数量。
- bathrooms：浴室数量。
- sqft_living：居住的面积（平方英尺[①]）。
- sqft_lot：实际的面积（平方英尺）。

① 1 平方英尺约等于 0.0929 平方米。

- floors：房屋总楼层数。
- waterfront：海景房。
- view：房屋是否看过。
- condition：整体条件。
- grade：房屋的整体等级（根据 King County 评分系统）。
- sqft_above：除了地下室以外的面积（平方英尺）。
- sqft_basement：地下室的面积（平方英尺）。
- yr_built：房屋建造时间。
- yr_renovated：何时重新装修过（一些未重新装修过或装修记录没被记录到的数值都为 0）。
- zipcode：邮政编码。
- lat：纬度坐标。
- long：经度坐标。
- sqft_living15：2015 年记录的居住面积（可能是翻新的原因，导致 sqft_living15 与 sqft_living 不同）。
- sqft_lot15：2015 年记录的实际面积（可能是翻新的原因，导致 sqft_lot15 与 sqft_lot 不同）。

对于这 21 种不同的信息，我们需要将其分成训练数据（Input X）和预期输出的标记答案（Input Y）两类，分别说明如下：

❶ 训练数据（Input X）：date、bedrooms、bathrooms、sqft_living、sqft_lot、floors、view、waterfront、condition、grade、sqft_above、sqft_basement、yr_built、yr_renovated、lat、zipcode、long、sqft_living15、sqft_lot15。

❷ 预期输出的标记答案（Input Y）：price。

3.3.2 新建项目

建议使用 Jupyter Notebook 执行本章的程序代码，操作流程如下：

Step 01 启动 Jupyter Notebook。

在 Terminal（Ubuntu）或命令提示符（Windows）中输入如下指令：

```
.\tf2\Scripts\activate
jupyter notebook
```

Step 02 新建执行文件。

单击界面右上角的 New 下拉按钮，然后选择所安装的 Python 解释器（在 Jupyter 中都称为 Kernel）来启动它。图 3-12 显示了 3 个不同的 Kernel，分别为：

- Python 3：本地端 Python。
- tf2：虚拟机 Python（TensorFlow-cpu 版本）。
- tf2-gpu：虚拟机 Python（TensorFlow-gpu 版本）。

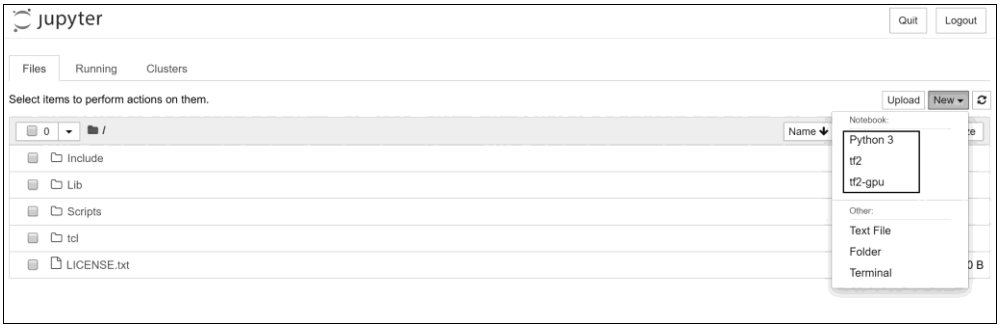


图 3-12　新建执行文件

Step 03　执行程序代码。

按 Shift + Enter 快捷键来执行单行程序代码，如图 3-13 所示。

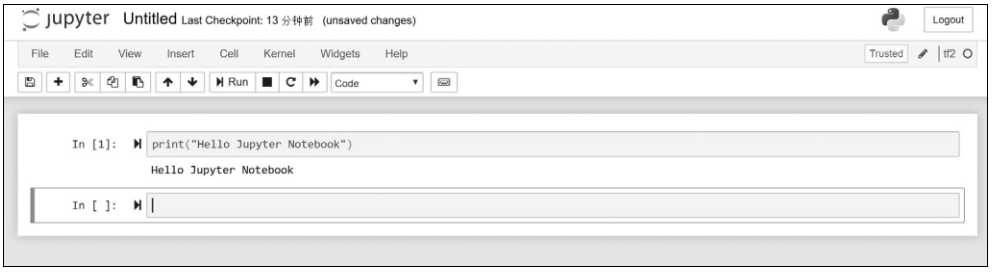


图 3-13　Jupyter 环境界面

接下来，本章后续的程序代码都可在 Jupyter Notebook 上执行。

3.3.3　程序代码

Step 01　导入必要的套件。

```
import os
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers
```

Step 02　数据读取并分析。

- *读取数据*

示例代码如下：

```
data = pd.read_csv("kc_house_data.csv")
# 显示数据集的形状，共 21613 笔数据，每一笔数据有 21 种不同信息
data.shape
```

结果如下：

```
(21613, 21)
```

- 显示数据

示例代码如下：

```
# 将显示列数设置为 25，否则会有部分数据无法显示
pd.options.display.max_columns = 25
# head 会显示前 5 行（默认）数据
data.head()
```

结果如图 3-14 所示。

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basem
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	3	7	1180	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	3	7	2170	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	3	6	770	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	5	7	1050	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	3	8	1680	

图 3-14　执行结果

- 检查数据类型

数据类型总共有 5 种：字符串（string）、布尔值（boolean）、整数（integer）、浮点数（float）和类别（categorical）。

```
data.dtypes
```

结果如下：

```
id              int64
date            object
price           float64
bedrooms        int64
bathrooms       float64
sqft_living     int64
sqft_lot        int64
floors          float64
waterfront      int64
view            int64
condition       int64
grade           int64
sqft_above      int64
sqft_basement   int64
yr_built        int64
yr_renovated    int64
zipcode         int64
lat             float64
long            float64
sqft_living15   int64
```

```
sqft_lot15      int64
dtype: object
```

Step 03 数据预处理。

- 转换数据类型

因为数据集里的日期（date）数据是字符串（string）类型的，而模型的输入只接收数值类型，所以可以通过以下程序代码将其转为数值类型，并分成年、月、日 3 种数据。

```
# 将日期拆为年、月、日并转成数值
data['year'] = pd.to_numeric(data['date'].str.slice(0, 4))
data['month'] = pd.to_numeric(data['date'].str.slice(4, 6))
data['day'] = pd.to_numeric(data['date'].str.slice(6, 8))
# 删除没有用的数据，inplace 则是将更新后的数据存回原来的地方
data.drop(['id'], axis="columns", inplace=True)
data.drop(['date'], axis="columns", inplace=True)
data.head()
```

结果如图 3-15 所示。

aterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15	year	month	day
0	0	3	7	1180	0	1955	0	98178	47.5112	-122.257	1340	5650	2014	10	13
0	0	3	7	2170	400	1951	1991	98125	47.7210	-122.319	1690	7639	2014	12	9
0	0	3	6	770	0	1933	0	98028	47.7379	-122.233	2720	8062	2015	2	25
0	0	5	7	1050	910	1965	0	98136	47.5208	-122.393	1360	5000	2014	12	9
0	0	3	8	1680	0	1987	0	98074	47.6168	-122.045	1800	7503	2015	2	18

图 3-15　执行结果

- 分割数据集

将数据集切割成 3 部分：训练数据（Training Data）、验证数据（Validation Data）和测试数据（Testing Data）。

```
data_num = data.shape[0]
# 取出一笔与 data 数量相同的随机数索引，主要用于打散数据
indexes = np.random.permutation(data_num)
# 并将随机数索引值分为 train、validation 和 test，这里的划分比例为 6:2:2
train_indexes = indexes[:int(data_num *0.6)]
val_indexes = indexes[int(data_num *0.6):int(data_num *0.8)]
test_indexes = indexes[int(data_num *0.8):]
# 通过索引值从 data 取出训练数据、验证数据和测试数据
train_data = data.loc[train_indexes]
val_data = data.loc[val_indexes]
test_data = data.loc[test_indexes]
```

Step 04 归一化（Normalization）。

归一化的主要作用是将不同的数据缩放到相同的大小，例如，房屋卧室或浴室的数量是 1~5 间，房屋居住的面积为 1500~2500m^2，由于数据量级差异甚大，因此可能会导致网络更加重视数值较大的数据而忽略数值较小的数据。为了解决此问题，我们通常会将输入数据缩放至 0~1 或-1~1，

这个过程就被称为数据归一化（Data Normalization）。

本实验使用标准分数（Standard Score，又称为 z-score）来将数据归一化，经过 z-score 归一化后的数据都会聚集在 0 附近且标准差为 1。

$$x_{norm} = \frac{(x - mean)}{std}$$

```
train_validation_data = pd.concat([train_data, val_data])
mean = train_validation_data.mean()
std = train_validation_data.std()
train_data = (train_data - mean) / std
val_data = (val_data - mean) / std
```

Step 05 建立 NumPy array 格式的训练数据。

```
x_train = np.array(train_data.drop('price', axis='columns'))
y_train = np.array(train_data['price'])
x_val = np.array(val_data.drop('price', axis='columns'))
y_val = np.array(val_data['price'])
```

训练数据共有 12967 笔，每笔有 21 种信息（代表模型输入的数据维度为 21）。

```
x_train.shape
```

结果如下：

```
(12967, 21)
```

Step 06 建立并训练网络模型。

- 搭建全连接网络模型

这里构建 3 层全连接层的网络架构，并且使用 ReLU 作为隐藏层的激活函数，由于需要得到线性输出，因此输出层不使用任何激活函数。

```
# 建立一个 Sequential 类型的模型
model = keras.Sequential(name=' model-1')
# 第 1 层全连接层设为 64 个 unit，将输入形状设置为(21, )，而实际上我们输入的数据形状为
  (batch_size, 21)
model.add(layers.Dense(64, activation='relu', input_shape=(21,)))
# 第 2 层全连接层设为 64 个 unit
model.add(layers.Dense(64, activation='relu'))
# 最后一层全连接层设为 1 个 unit
model.add(layers.Dense(1))
# 显示网络模型架构
model.summary()
```

结果如图 3-16 所示。