3

TensorFlow 基础

3.1. 概述

在这第一章中，我们将介绍三种深度学习的人工神经网络，我们将在全书中使用这些网络。这些网络是 MLP、CNN 和 RNN（在第 2 节中定义和描述），它们是本书所涉及的一些高级深度学习主题的构建模块，例如自回归网络（autoencoder、GAN 和 VAE）、深度强化学习、物体检测和分割，以及使用互信息的无监督学习。

在本章中，我们将一起讨论如何使用 Keras 库实现基于 MLP、CNN 和 RNN 的模型。更具体地说，我们将使用名为 tf.keras 的 TensorFlow Keras 库。我们将首先看看为什么 tf.keras 是我们作为工具的最佳选择。接下来，我们将挖掘三个深度学习网络内的实现细节。

本章的主要内容如下：

- 确定为什么 tf.keras 库是用于高级深度学习的最佳选择
- 介绍 MLP、CNN 和 RNN–高级深度学习模型的核心构建模块，我们将在本书中使用它们
- 提供如何使用 tf.keras 实现基于 MLP、CNN 和 RNN 模型的例子
- 在此过程中，开始介绍重要的深度学习概念，包括优化、正则化和损失函数

在本章结束时，我们将拥有使用 tf.keras 实现的基本深度学习网络。在下一章，我们将进入建立在这些基础上的高级深度学习主题。让我们在本章开始讨论 Keras 及其作为深度学习库的能力。

3.2. 深度学习库 Keras

Keras[1] 是一个流行的深度学习库，有超过 37 万名开发者在使用它–这个数字每年都在以约 35% 的速度增长。超过 800 名贡献者积极维护它。我们将在本书中使用的一些例子已经被贡献到了 Keras 的官方 GitHub 仓库。

谷歌的 TensorFlow，一个流行的开源深度学习库，使用 Keras 作为其库的高级 API。它通常被称为 tf.keras。在本书中，我们将交替使用 Keras 和 tf.keras 这个词。

tf.keras 作为深度学习库是一个受欢迎的选择，因为它高度集成到 TensorFlow 中，TensorFlow 在生产部署中以其可靠性而闻名。TensorFlow 还提供了各种工具，用于生产部署和维护、调试和

可视化，以及在嵌入式设备和浏览器上运行模型。在技术行业，Keras 被谷歌、Netflix、Uber 和英伟达使用。

我们选择 tf.keras 作为本书的首选工具，因为它是一个专门用于加速实现深度学习模型的库。这使得 Keras 成为我们想要实践和动手的理想工具，比如我们在本书中探索高级深度学习概念时。由于 Keras 旨在加速深度学习模型的开发、训练和验证，因此在有人能够最大限度地使用该库之前，学习该领域的关键概念是非常必要的。

在 tf.keras 库中，层与层之间的连接就像乐高积木的碎片一样，从而形成了一个干净、易懂的模型。模型训练很简单，只需要数据、一些训练的 epochs 和监测的指标。

最终的结果是，与其他深度学习库相比，大多数深度学习模型可以用少得多的代码行来实现如 PyTorch。通过使用 Keras，我们将通过节省代码实现的时间来提高生产力，而这些时间可以用于更关键的任务，如制定更好的深度学习算法。同样，Keras 也是快速实现深度学习模型的理想选择，比如我们在本书中要使用的模型。使用序列模型 API，典型的模型只需几行代码就可以建立。然而，不要被它的简单性所误导。

Keras 还可以使用其功能 API 和动态图的 Model 和 Layer 类来构建更高级、更复杂的模型，这些都可以通过定制来满足独特的要求。功能性 API 支持构建类似于图的模型，层的重用，以及创建行为像 Python 函数的模型。同时，模型和层类为实现不常见的或实验性的深度学习模型和层提供一个框架。

3.2.1. 安装 Keras 和 TensorFlow

Keras 不是一个独立的深度学习库。正如你在图3.1中看到的，它是建立在另一个深度学习库或后端之上的。这可能是谷歌的 TensorFlow，MILA 的 Theano，微软的 CNTK，或 Apache MXNet。然而，与本书上一版不同的是，我们将使用 TensorFlow 2.0（tf2 或简称 tf）提供的 Keras，它更被称为 tf.keras，以利用 tf2 提供的有用工具。tf.keras 也被认为是 TensorFlow 事实上的前端，它在生产环境中展示了其成熟的可靠性。此外，Keras 对 TensorFlow 之外的其他后端的支持在不久的将来将不再可用。

从 Keras 迁移到 tf.keras 一般来说就像改变一样简单。

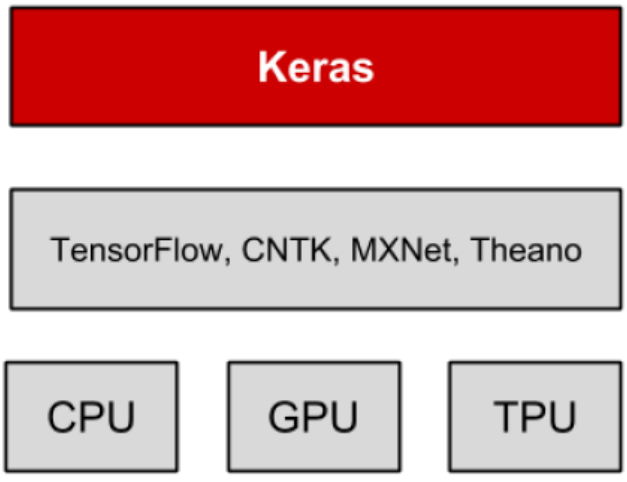


图 3.1: Keras 是一个高级别库，位于其他深度学习框架之上。Keras 被支持在 CPU、GPU 和 TPU 上。

Listing 3.1: keras 代码

```
from keras... import ...
```

变成

Listing 3.2: tensorflow.keras 代码

```
from tensorflow.keras... import ...
```

在本书中，由于对 Python 2 的支持将在 2020 年结束，所以代码实例都是用 Python 3 编写的。

在硬件方面，Keras 在 CPU、GPU 和谷歌的 TPU 上运行。在本书中，我们将在 CPU 和 NVIDIA GPU（特别是 GTX 1060、GTX 1080Ti、RTX 2080Ti、V100 和 Quadro RTX 8000 型号）上测试。

在进行本书的其他部分之前，我们需要确保 tf2 已经正确安装。有多种方法可以进行安装，其中一个例子是用 pip3 安装 tf2。

Listing 3.3: 用 pip3 安装 tf2

```
$ sudo pip3 install tensorflow
```

如果我们有一个支持 NVIDIA 的 GPU，并正确安装了驱动程序，同时有 NVIDIA CUDA 工具包和 cuDNN 深度神经网络库，强烈建议你安装支持 GPU 的版本，因为它可以加速训练和预测。

Listing 3.4: 用 pip3 安装 tensorflow

```
$ sudo pip3 install tensorflow-gpu
```

没有必要安装 Keras，因为它已经是 tf2 中的一个包了。如果你不习惯在全系统范围内安装库，强烈建议使用 Anaconda 等环境。除了拥有一个孤立的环境之外，Anaconda 发行版还安装了数据科学常用的第三方软件包，这些软件包对于深度学习来说是不可或缺的。

本书中介绍的例子将需要额外的软件包，如 pydot、pydot_ng、vizgraph、python3-tk 和 matplotlib。在进行本章以后的内容之前，我们需要安装这些包。

如果 tf2 和它的依赖项一起安装，下面的内容应该不会产生任何错误。

Listing 3.5: 用 pip3 安装 tensorflow

```
$ python3
>>> import tensorflow as tf
>>> print(tf.__version__)
2.0.0
>>> print(K.epsilon())
1e-07
```

本书并不涵盖完整的 Keras API。我们在本书中只涉及解释选定的高级深度学习主题所需的材料。[1] 在接下来的章节中，将讨论 MLP、CNN 和 RNN 的细节。这些网络将被用来使用 tf.keras 构建一个简单的分类器。

3.3. 多层感知器、卷积神经网络和循环神经网络

我们已经提到，我们将使用三个深度学习网络，它们是。

- MLP：多层感知器

[1] 关于更多信息，可以参考官方的 Keras 文档，可以在 https://keras.io 或 https://www.tensorflow.org/guide/keras/ overview。

- CNN：卷积神经网络（Convolutional Neural Network）
- RNN：循环神经网络 (Recurrent Neural Network)

这三个网络是我们将在本书中使用的。以后，你会发现它们经常被组合在一起，以便利用每个网络的优势。在本章中，我们将逐一详细讨论这些构件。在接下来的章节中，MLP 将与其他重要主题一起讨论，如损失函数、优化器和正则器。在这之后，我们将同时介绍 CNN 和 RNN。

3.3.1. MLP、CNN 和 RNN 之间的区别

MLP 是一个全连接（FC）网络。你经常会发现它在一些文献中被称为深度前馈网络或前馈神经网络。在本书中，我们将使用 MLP 这个术语。从已知的目标应用来理解这个网络，将有助于我们深入了解高级深度学习模型设计的根本原因。

MLPs 在简单的逻辑和线性回归问题中很常见。然而，MLP 并不是处理连续和多维数据模式的最佳选择。根据设计，MLP 很难记住连续数据的模式，并且需要大量的参数来处理多维数据。

对于顺序数据的输入，RNN 很受欢迎，因为内部设计允许网络在数据的历史中发现依赖性，这对预测很有用。对于像图像和视频这样的多维数据，CNN 擅长于提取特征图，用于分类、分割、生成和其他下游任务。在某些情况下，一维卷积形式的 CNN 也被用于具有连续输入数据的网络。然而，在大多数深度学习模型中，MLP 和 CNN 或 RNN 被结合起来，以充分发挥每个网络的作用。

MLP、CNN 和 RNN 并没有完成深度网络的全貌。有必要确定一个目标或损失函数，一个优化器和一个正则器。目标是在训练期间减少损失函数值，因为这种减少是一个很好的指标，表明一个模型正在学习。

为了最小化这个值，模型采用了一个优化器。这是一种算法，决定在每个训练步骤中应该如何调整权重和偏置。一个经过训练的模型不仅要在训练数据上工作，而且还要在训练环境之外的数据上工作。正则器的作用是确保训练后的模型能够泛化到新的数据。

现在，让我们来了解一下这三个网络–我们先来谈谈 MLP 网络。

在下一节中，我们将简要介绍 MNIST。

3.4. 多层感知器 (MLP)

我们要看的三个网络中的第一个是 MLP 网络。我们假设目标是创建一个神经网络，用于根据手写数字来识别数字。例如，当网络的输入是一个手写的数字 8 的图像时，相应的预测也必须是数字 8。这是一个典型的分类器网络的工作，可以用逻辑回归法来训练。为了训练和验证一个分类器网络，必须有一个足够大的手写数字数据集。修改后的美国国家标准和技术研究所数据集的数据集，简称 MNIST[2]，通常被认为是深度学习数据集中的" 你好世界"。它是一个适合手写数字分类的数据集。在我们讨论 MLP 分类器模型之前，我们必须了解 MNIST 数据集。本书中大量的例子都使用了 MNIST 数据集。MNIST 被用来解释和验证许多深度学习理论，因为它所包含的 70,000 个样本很小，但信息足够丰富。

在下一节中，我们将简要介绍 MNIST。

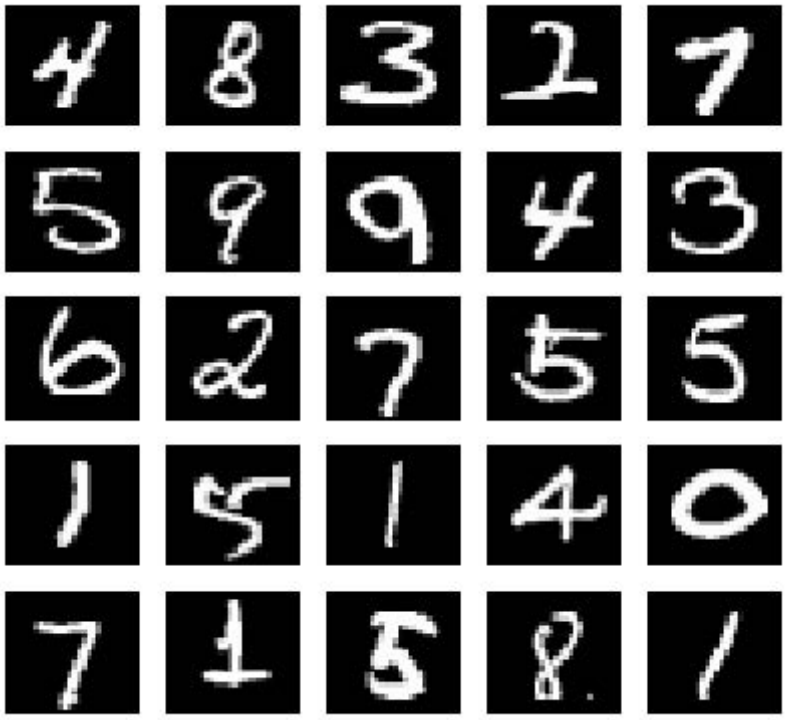


图 3.2: 来自 MNIST 数据集的示例图像。每张灰度图像是 28×28 像素的。

3.4.1. MNIST 数据集

MNIST 是一个手写数字的集合，范围从 0 到 9。它有一个由 60,000 张图像组成的训练集，以及 10,000 张被归入相应类别或标签的测试图像。在一些文献中，目标或地面真相一词也被用来指代标签。

在上图中，可以看到 MNIST 数字的样本图像，每个图像的大小为 28 x 28 - 像素，为灰度。为了在 Keras 中使用 MNIST 数据集，我们提供了一个 API 来自动下载和提取图像和标签。清单 1.3.1 演示了如何在一行中加载 MNIST 数据集，使我们能够计算训练和测试标签，然后绘制 25 张随机数字图像。

Listing 3.6: 用 pip3 安装 tensorflow

```
import numpy as np
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
# load dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# count the number of unique train labels
unique, counts = np.unique(y_train, return_counts=True)
print("Train labels: ", dict(zip(unique, counts)))
# count the number of unique test labels
unique, counts = np.unique(y_test, return_counts=True)
print("Test labels: ", dict(zip(unique, counts)))
# sample 25 mnist digits from train dataset
indexes = np.random.randint(0, x_train.shape[0], size=25)
```

```
    images = x_train[indexes]
    labels = y_train[indexes]
    # plot the 25 mnist digits
    plt.figure(figsize=(5,5))
    for i in range(len(indexes)):
        plt.subplot(5, 5, i + 1)
        image = images[i]
        plt.imshow(image, cmap='gray')
        plt.axis('off')
    plt.savefig("mnist-samples.png")
    plt.show()
    plt.close('all')
```

mnist.load_data() 方法很方便，因为不需要单独加载所有 70,000 张图片和标签并将它们存储在数组中。执行以下程序。

Listing 3.7: 用 pip3 安装 tensorflow

```
python3 mnist-sampler-1.3.1.py
```

在命令行中，代码示例打印出训练和测试数据集中的标签分布。

Train labels:{0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7: 6265, 8: 5851, 9: 5949} Test labels:{0: 980, 1: 1135, 2: 1032, 3: 1010, 4: 982, 5: 892, 6: 958, 7: 1028, 8: 974, 9: 1009}

之后，代码将绘制 25 个随机数字，如之前图 1.3.1 所示。

在讨论 MLP 分类器模型之前，必须牢记，虽然 MNIST 数据由二维张量组成，但应根据输入层的类型进行重塑。下面的图 1.3.2 显示了 3×3 灰度图像对于 MLP、CNN 和 RNN 输入层的重塑情况。

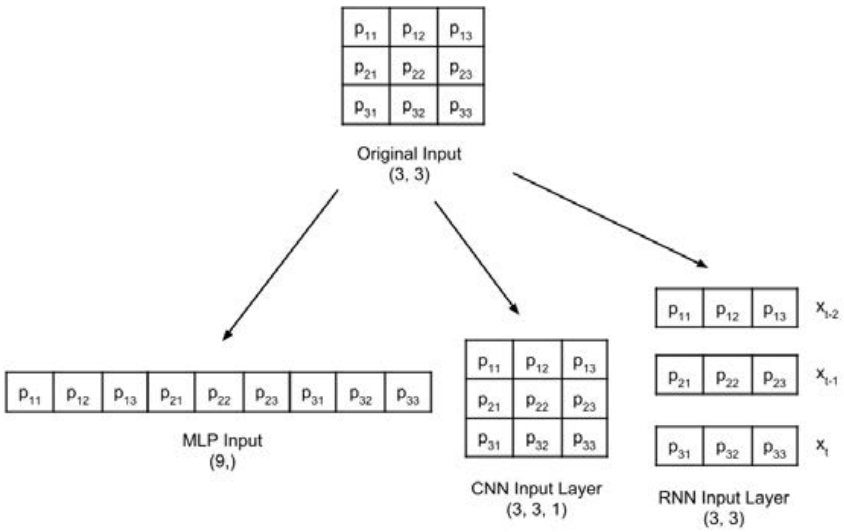


图 3.3: 类似于 MNIST 数据的输入图像根据输入层的类型进行了重塑。为简单起见，显示了 3×3 灰度图像的重塑。

在接下来的章节中，我们将介绍一个针对 MNIST 的 MLP 分类器模型。我们将演示如何使用 tf.keras 有效地建立、训练和验证该模型。

在接下来的章节中，我们将介绍一个针对 MNIST 的 MLP 分类器模型。我们将演示如何使用 tf.keras 有效地建立、训练和验证该模型。

3.4.2. MNIST 数字分类器模型

图3.4中提出的 MLP 模型可用于 MNIST 数字分类。当单元或感知器暴露时，MLP 模型是一个全连接网络，如图 1.3.4 所示。我们还将展示如何将感知器的输出是如何从输入中计算出第 n 个单元的权重 (w_i) 和偏置 (b_n) 的函数。相应的 tf.keras 的实现如下在清单 1.3.2 中。

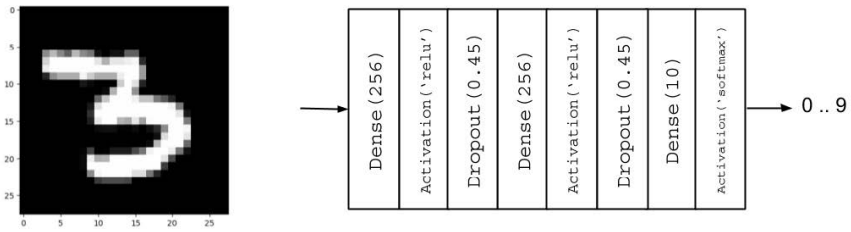


图 3.4: MLP MNIST 数字分类器模型

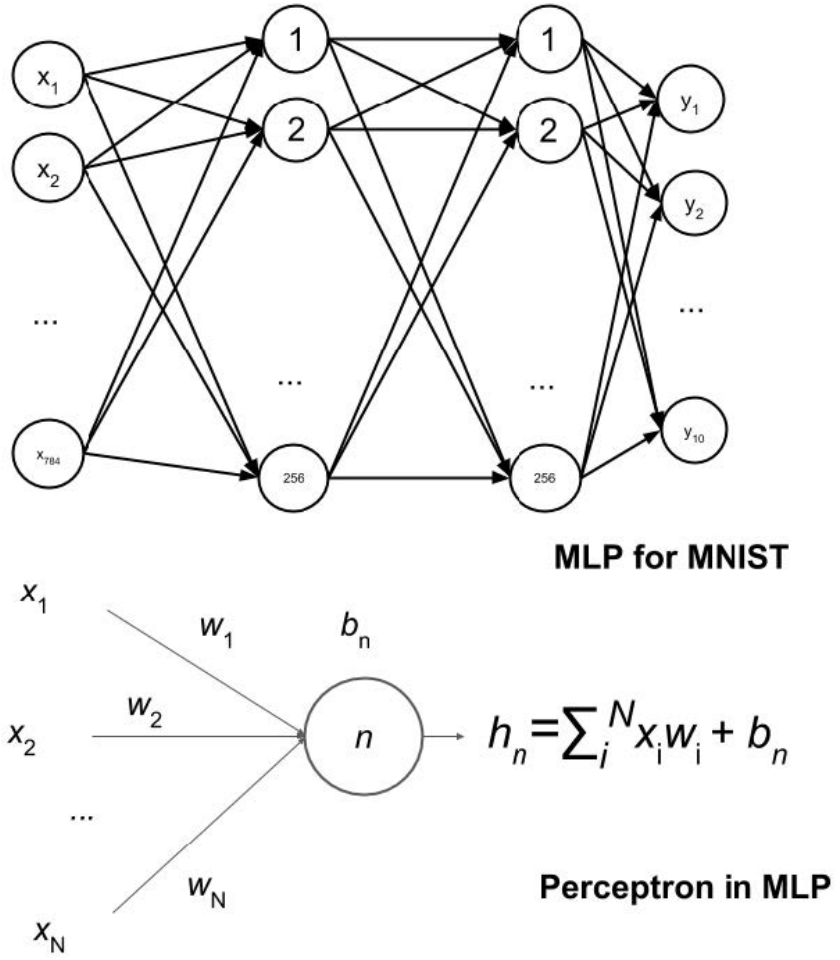


图 3.5: 图 1.3.3MLP MNIST 数字分类器是由全连接层组成。

Listing 3.8: mlp-mnist-1.3.2.py language

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras.datasets import mnist
# load mnist dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# compute the number of labels
num_labels = len(np.unique(y_train))
# convert to one-hot vector
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
# image dimensions (assumed square)
image_size = x_train.shape[1]
input_size = image_size * image_size
# resize and normalize
x_train = np.reshape(x_train, [-1, input_size])
```

```
18    x_train = x_train.astype('float32') / 255
19    x_test = np.reshape(x_test, [-1, input_size])
20    x_test = x_test.astype('float32') / 255
21    # network parameters
22    batch_size = 128
23    hidden_units = 256
24    dropout = 0.45
25    # model is a 3-layer MLP with ReLU and dropout after each layer
26    model = Sequential()
27    model.add(Dense(hidden_units, input_dim=input_size))
28    model.add(Activation('relu'))
29    model.add(Dropout(dropout))
30    model.add(Dense(hidden_units))
31    model.add(Activation('relu'))
32    model.add(Dropout(dropout))
33    model.add(Dense(num_labels))
34    # this is the output for one-hot vector
35    model.add(Activation('softmax'))
36    model.summary()
37    plot_model(model, to_file='mlp-mnist.png', show_shapes=True)
38    # loss function for one-hot vector
39    # use of adam optimizer
40    # accuracy is good metric for classification tasks
41    model.compile(loss='categorical_crossentropy',
42                  optimizer='adam',
43                  metrics=['accuracy'])
44    # train the network
45    model.fit(x_train, y_train, epochs=20, batch_size=batch_size)
46    # validate the model on test dataset to determine generalization
47    _, acc = model.evaluate(x_test,
48                            y_test,
49                            batch_size=batch_size,
50                                                    verbose=0)
51    print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

在讨论模型的实现之前，数据必须有正确的形状和格式。在加载 MNIST 数据集后，标签数量的计算方法是：

Listing 3.9: mlp-mnist-1.3.2.py

```
1    # compute the number of labels
2    num_labels = len(np.unique(y_train))
```

硬编码 $num_labels = 10$ 也是一种选择。但是，让计算机做它的工作总是一个好的做法。代码假设 y_train 的标签是 0 到 9。在这一点上，标签是数字格式，即从 0 到 9。这种标签的稀疏标量表示法不适合于每类输出概率的神经网络预测层。一个更合适的格式被称为一热向量，一个 10 维向量，所有元素都是 0，除了数字类的索引。例如，如果标签是 2，等效的单热向量是

$$0, 0, 1, 0, 0, 0, 0, 0$$

。第一个标签的索引为 0。

下面几行将每个标签转换为一个单热向量。

Listing 3.10: 单热（one-hot）向量

```
1 # convert to one-hot vector
```

```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

在深度学习中，数据被存储在张量中。张量一词适用于标量（0D 张量）、矢量（1D 张量）、矩阵（二维张量）和多维张量。从这一点来看，除非标量、矢量或矩阵能使解释更清楚，否则就使用张量这个术语。如下图所示，代码的其余部分计算了图像尺寸、第一个密集层的 input_size 值，并将每个像素值从 0 到 255 进行缩放，范围从 0.0 到 1.0。虽然可以直接使用原始像素值，但最好是将输入的数据归一化。将输入数据归一化，以避免大的梯度值导致训练困难。网络的输出也被归一化了。训练结束后，可以选择将输出张量乘以 255，把所有东西放回整数像素值。建议的模型是基于 MLP 层的。因此，预计输入是一个一维张量。因此，x_train 和 x_test 分别被重塑为 [60,000, 28 * 28] 和 [10,000, 28 * 28]。在 NumPy 中，-1 的大小意味着让库计算出正确的维度。在 x_train 的例子中，这就是 60,000。

Listing 3.11: one-hot vector

```
# image dimensions (assumed square) 400
image_size = x_train.shape[1]
input_size = image_size * image_size
# resize and normalize
x_train = np.reshape(x_train, [-1, input_size])
x_train = x_train.astype('float32') / 255
x_test = np.reshape(x_test, [-1, input_size])
x_test = x_test.astype('float32') / 255
```

在准备好数据集后，下面重点是使用 Keras 的 Sequential API 构建 MLP 分类器模型。

3.4.3. 使用 MLP 和 Keras 建立一个模型

在数据准备之后，接下来是建立模型。建议的模型由三个 MLP 层组成。在 Keras 中，一个 MLP 层被称为密集，代表密集连接层。第一个和第二个 MLP 层的性质都是一样的，各有 256 个单元，然后是整流线性单元（ReLU）的激活和放弃。选择 256 个单元是因为 128、512 和 1,024 个单元的性能指标较低。在 128 个单元时，网络会迅速收敛，但测试精度较低。512 或 1,024 的额外单元数并没有明显提高测试精度。单位数是一个超参数。它控制着网络的容量。容量是对网络可以近似的函数的复杂性的一种衡量。例如，对于多项式，度数是超参数。随着度数的增加，函数的容量也会增加。如下面几行代码所示，分类器模型是使用 Keras 的 Sequential API 实现的。如果该模型需要一个输入和一个输出由一个序列层处理，这就足够了。为了简单起见，我们现在就使用这个；然而，在第二章” 深度神经网络” 中，将介绍 Keras 的 Functional API，以实现需要更复杂结构的高级深度学习模型，如多个输入和输出。

Listing 3.12: one-hot vector

```
# model is a 3-layer MLP with ReLU and dropout after each layer
model = Sequential()
model.add(Dense(hidden_units, input_dim=input_size))
model.add(Activation('relu'))
model.add(Dropout(dropout))
model.add(Dense(hidden_units))
model.add(Activation('relu'))
model.add(Dropout(dropout))
model.add(Dense(num_labels))
# this is the output for one-hot vector model.
add(Activation('softmax'))
```

由于密集层是一个线性操作，密集层的序列只能近似于一个线性函数。问题是，MNIST 数字分类本身就是一个非线性过程。在密集层之间插入一个 relu 激活，将使 MLP 网络能够模拟非线性映射。relu 或 ReLU 是一个简单的非线性函数。它非常像一个过滤器，允许正的输入不变地通过，而把其他的东西都钳制为零。在数学上，Relu 用以下公式表示，并在图3.6中绘制。

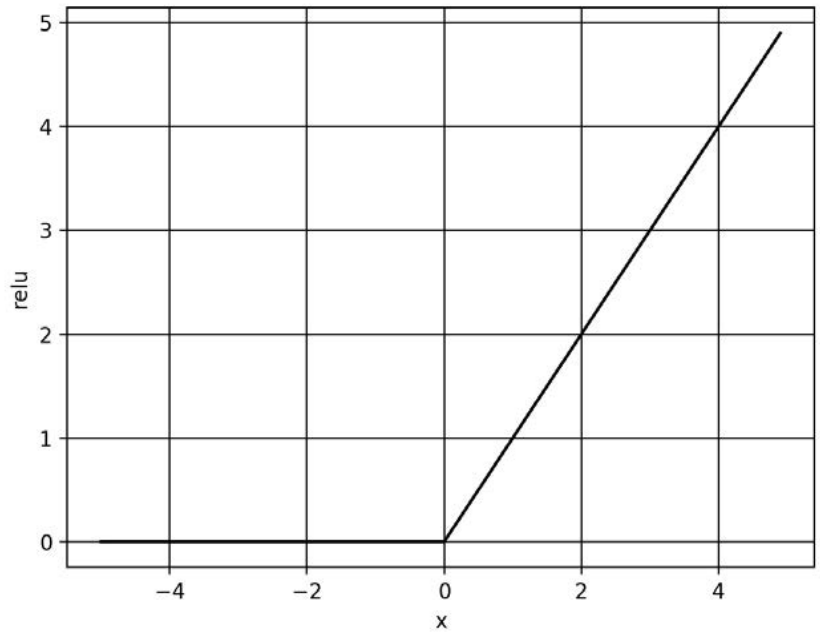


图 3.6: ReLU 函数的图谱。ReLU 函数在神经网络中引入了非线性。

还有其他可以使用的非线性函数，如 elu、selu、softplus、sigmoid 和 tanh。然而，relu 是最常用的函数，由于其简单性，计算效率高。sigmoid 和 tanh 函数被用作输出层的激活函数，将在后面介绍。表 1.3.1 显示了这些激活函数的每一个方程式。relu:

$$relu(x) = max(0, x) \tag{3.1}$$

softplus:

$$softplus(x) = \log(1 + e^x) \tag{3.2}$$

selu

$$selu(x) = k \times elu(x, a) \tag{3.3}$$

其中，$k = 1.0507009873554804934193349852946$，$a = 1.6732632423543772848170429916717$

sigmoid

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

tanh

$$tanh(x) = \frac{e^x - e{-x}}{e^x + e^{-x}} \tag{3.5}$$

虽然我们已经完成了 MLP 分类器模型的关键层，但我们还没有解决泛化的问题，也没有解决模型在训练数据集之外的能力问题。为了解决这个问题，我们将在下一节介绍正则化。

3.4.4. 正则化

一个神经网络有记忆其训练数据的倾向，特别是当它包含足够多的容量时。在这种情况下，该网络在接受测试数据时就会出现灾难性的失败。这就是网络不能泛化的典型案例。为了避免这种趋势，该模型使用了一个正则化层或函数。一个常见的正则化层是 Dropout。辍学的概念很简单。给定一个辍学率（这里设定为辍学率 =0.45），辍学层随机地将这部分单元从下一层中删除。例如，如果第一层有 256 个单位，在应用 dropout = 0.45 后，只有 $(1-0.45)\times 256$ 个单位 = 140 个单位从第一层参与第二层。

Dropout 层使神经网络对不可预见的输入数据具有鲁棒性，因为网络被训练成正确的预测，即使有些单元丢失。值得注意的是，Dropout 不用于输出层，它只在训练期间活跃。此外，在预测过程中，dropout 也不存在。除了 dropout 之外，还有一些正则器可以使用，比如 l1 或 l2。在 Keras 中，每层的偏置、权重和激活输出都可以被正则化。l1 和 l2 通过增加一个惩罚函数来支持更小的参数值。l1 和 l2 都使用参数值的绝对值（l1）或平方值（l2）之和的一小部分来执行惩罚。换句话说，惩罚函数迫使优化器找到小的参数值。具有小参数值的神经网络对输入数据中存在的噪声更加不敏感。作为一个例子，一个分数 =0.001 的 l2 权重的正则器可以实现为。

Listing 3.13: one-hot vector

```
from tensorflow.keras.regularizers import l2
model.add(Dense(hidden_units,
                kernel_regularizer=l2(0.001),
                input_dim=input_size))
```

如果使用 $l1$ 或 $l2$ 正则化，则不会增加额外的层。正则化是在 Dense 层内部施加的。对于提议的模型，dropout 仍然比 $l2$ 有更好的性能。我们几乎完成了我们的模型。下一节主要讨论输出层和损失函数。

3.4.5. 输出激活和损失函数

输出激活和损失函数输出层有 10 个单元，然后是软键激活层。这 10 个单元对应于 10 个可能的标签、类或类别。软键激活可以用数学方法表示，如下式所示。

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{N-1} e^{x_j}} \tag{3.6}$$

该方程适用于所有 $N = 10$ 个输出，x_i 为 $i = 0, 1 \cdots 9$，用于最终预测。softmax 的想法出奇地简单。它通过归一化预测将输出压成概率。这里，每个预测的输出是一个概率，即该指数是给定输入图像的正确标签。所有输出的所有概率之和为 1.0。例如，当 softmax 层产生一个预测时，它将是一个 10 维的一维张量，可能看起来像以下输出。

Listing 3.14: 输出

```
[3.57351579e-11 7.08998016e-08  2.30154569e-07  6.35787558e-07  5.57471187e-11   4.15353840e-09
    3.55973775e-16  9.99995947e-01  1.29531730e-09  3.06023480e-06]
```

预测输出张量表明，鉴于其索引具有最高的概率，输入图像将是 7。numpy.argmax() 方法可以用来确定具有最高值的元素的索引。

输出激活层还有其他选择，如线性、sigmoid 或 tanh。线性激活是一个身份函数。它将其输入复制到其输出。sigmoid 函数更具体地说就是 logistic sigmoid。如果预测张量的元素将在 0.0 和 1.0 之间独立映射，就会使用这个函数。预测张量的所有元素的总和并不像 softmax 那样被限制在

1.0 以内。例如，sigmoid 被用作情感预测（从 0.0 到 1.0，0.0 是坏的，1.0 是好的）或图像生成（0.0 被映射到像素级别 0，1.0 被映射到像素 255）的最后一层。

tanh 函数在-1.0 到 1.0 范围内映射其输入。如果输出可以在正值和负值中摆动，这就很重要。tanh 函数更常用于递归神经网络的内部层，但也被用作输出层激活。如果用 tanh 来代替 s□igmoid 的输出激活，必须对所用数据进行适当的缩放。例如，用 $= \frac{x}{255}$ 缩放每个灰度像素的范围 [0.0 1.0]，而是将其分配在 [-1.0 至 1.0] 的范围内，使用 $x = \frac{x-127.5}{127.5}$ 。

图3.7中的下图显示了 sigmoid 和 tanh 函数。在数学上，sigmoid 可以用以下公式表示。

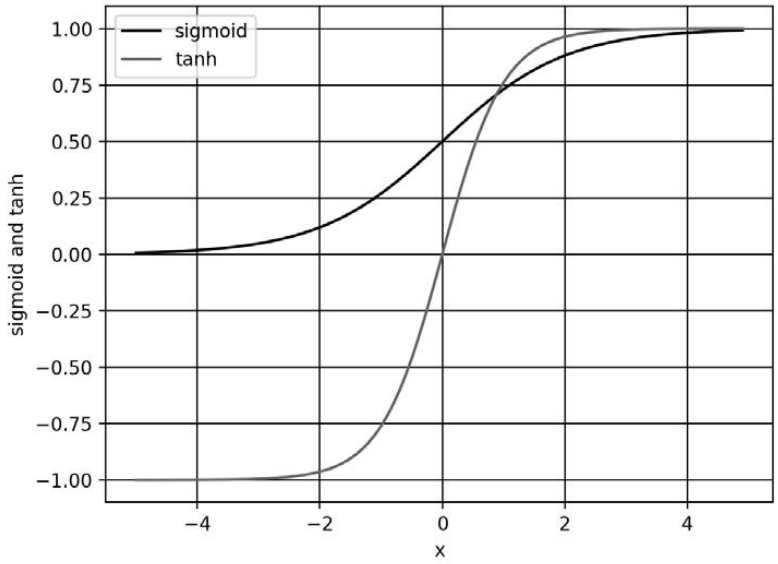


图 3.7: 二次方和 tanh 的曲线图。

$$sigmoid(x) = \omega(x) = \frac{1}{1+e^{-x}} \tag{3.7}$$

预测的张量与一击即中的地面真实向量的距离称为损失。损失函数的一种类型是平均误差（MSE），或目标或标签与预测之间差异的平方的平均值。在当前的例子中，我们使用的是 categorical_crossentropy。它是目标或标签与每个类别的预测的对数的乘积的负数。Keras 中还有其他损失函数，如 mean_ absolute_error 和 binary_crossentropy。以下总结了常见的损失函数。

损失函数：**mean_squared_error** 的数学表达

$$\frac{1}{categories}\sum_{i=1}^{categories}(y_i^{ladel} - y_i^{prediction})^2 \tag{3.8}$$

损失函数：mean_absolute_error 的数学表达

$$\frac{1}{categories}\sum_{i=1}^{categories}|y_i^{ladel} - y_i^{prediction}| \tag{3.9}$$

损失函数：categorical_crossentropy 的数学表达

$$-\sum_{i=1}^{categories} y_i^{ladel} \log y_i^{prediction} \tag{3.10}$$

损失函数：binary_crossentropy 的数学表达

$$-y_i^{ladel} \log y_i^{prediction} - (1 - y_1^{label}) \log(1 - y_1^{prediction}) \tag{3.11}$$

损失函数的选择不是任意的，而应该是模型学习的一个标准。对于按类别分类，在 softmax 激活层之后，categorical_crossentropy 或 mean_squared_error 是一个不错的选择。二元交叉熵损失函数通常用在 sigmoid 激活层之后，而 mean_squared_error 是 tanh 输出的一个选择。

在下一节，我们将讨论优化算法，以最小化我们在这里讨论的损失函数。

3.4.6. 优化

通过优化，目标是最小化损失函数。我们的想法是，如果损失被降低到一个可接受的水平，那么模型已经间接地学会了将输入映射到输出的函数。性能指标被用来确定一个模型是否已经学会了基础数据分布。Keras 的默认指标是损失。在训练、验证和测试期间，也可以包括其他指标，如准确率。准确率是指基于基础事实的正确预测的百分比，或分数。在深度学习中，还有许多其他性能指标。然而，这取决于模型的目标应用。在文献中，测试数据集上训练好的模型的性能指标被报告出来，以便与其他深度学习模型进行比较。

在 Keras 中，有几种优化器的选择。最常用的优化器是随机梯度下降（SGD）、自适应矩（Adam）和均方根传播（RMSprop）。每个优化器都有可调整的参数，如学习率、动量和衰减。Adam 和 RMSprop 是具有自适应学习率的 SGD 的变体。在拟议的分类器网络中，使用了 Adam，因为它具有最高的测试精度。

SGD 被认为是最基本的优化器。它是微积分中梯度下降的一个简单版本。在梯度下降法（GD）中，追踪一个函数的曲线下坡找到最小值，就像在山谷中走下坡路直到到达谷底一样。

图3.8是 GD 算法的图示。我们假设 x 是被调整的参数（例如，权重），以找到 y 的最小值（例如，损失函数）。从 $x = -0.5$ 的任意点开始，梯度 $\frac{dy}{dx} = -2.0$。GD 算法规定，然后将 x 更新为 $= -0.5 - \epsilon(-2.0)$。x 的新值等于旧值，加上以 ϵ 缩放的梯度的反面。小数字 ϵ 指的是学习率。如果 $\epsilon = 0.01$，那么 x 的新值 $x = -0.48$。GD 是迭代进行的。在每一步，y 会越来越接近其最小值。在 $x = 0.5$ 时，$\frac{dy}{dx}$。GD 已经找到了 $y = -1.25$ 的绝对最小值。梯度建议不再进一步改变 x。

学习率的选择是至关重要的。一个大值的 ϵ 可能找不到最小值，因为搜索将只是在最小值周围来回摆动。一方面，一个大的 ϵ 值可能需要大量的迭代才能找到最小值。在有多个最小值的情况下，搜索可能会卡在一个局部最小值上。

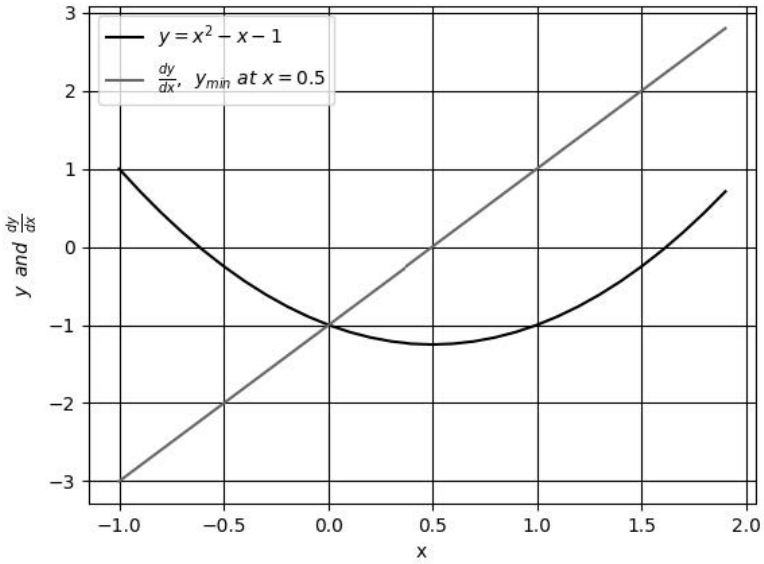


图 3.8: GD 类似于在函数曲线上走下坡路，直到到达最低点。在这幅图中，全局最低点是在 x=0.5。

图 1.3.8 中可以看到多个最小值的例子。如果由于某种原因，搜索从图的左边开始，而且学习率非常小，那么 GD 很有可能找到 $x = -1.51$ 作为 y 的最小值，GD 不会找到 $x = 1.66$ 的全局最小值。一个足够大的学习率将使 GD 能够克服 $x = 0.0$ 处的高点。

在深度学习的实践中，通常建议从较大的学习率开始（例如，0.1 到 0.001），并随着损失越来越接近最小值而逐渐减少。

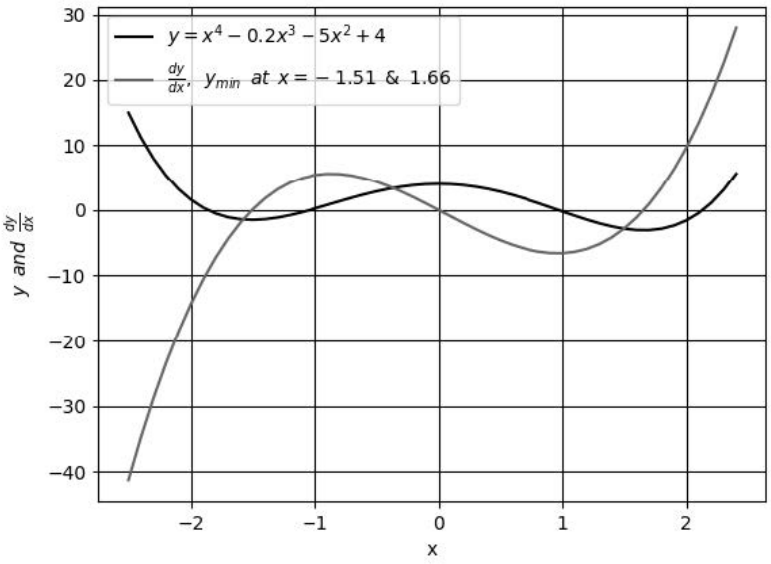


图 3.9: 一个有两个最小值的函数图，$x = -1.51$ 和 $x = 1.66$。还显示了该函数的导数。

$$\Theta \leftarrow \Theta - \epsilon g \tag{3.12}$$

在这个方程中，Θ 和 $g = \frac{m}{1}\nabla_\Theta \sum L$ 分别是损失函数的参数和梯度张量。损失函数的参数和梯度张量。g 是由损失函数的偏导数计算出来的。出于 GPU 优化的目的，建议迷你批次的大小为 2 的幂。在拟议的网络中，batch_size = 128。

公式3.12计算了最后一层的参数更新。那么，我们如何调整前面各层的参数呢？在这种情况下，应用微分的连锁规则将导数传播到下层，并相应地计算梯度。这种算法在深度学习中被称为反向传播。逆传播的细节已经超出了本书的范围。然而，在 http://neuralnetworksanddeeplearning.com，可以找到一个很好的在线参考。

由于优化是基于微分的，因此损失函数的一个重要标准是它必须是平滑或可微分的。在引入新的损失函数时，这是一个重要的约束条件，要牢记在心。

鉴于训练数据集、损失函数的选择、优化器和正则器，现在可以通过调用 fit() 函数来训练模型。

Listing 3.15: 调用 fit() 函数来训练模型

```
# loss function for one-hot vector
# use of adam optimizer
# accuracy is a good metric for classification tasks model.
compile(loss='categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
# train the network
model.fit(x_train, y_train, epochs=20, batch_size=batch_size)
```

这是 Keras 的另一个有用的功能。只需提供 x 和 y 数据、训练的历时数和批次大小，fit() 就能完成剩下的工作。在其他深度学习框架中，这意味着多项任务，如以适当的格式准备输入和输出数据、加载、监控等等。虽然所有这些都必须在 for 循环中完成，但在 Keras 中，一切都只需要一行

就可以完成。在 fit() 函数中，一个 epoch 是对整个训练数据的完整采样。batch_size 参数是每个训练步骤中要处理的输入数量的样本大小。为了完成一个 epoch，fit() 将处理等于训练数据集的大小除以批次大小加 1 的步骤数，以补偿任何小数部分。训练完模型后，我们现在可以评估其性能。

性能评估至此，MNIST 数字分类器的模型已经完成。性能评估将是下一个关键步骤，以确定拟议的训练模型是否提出了一个令人满意的解决方案。对模型进行 20 个历时的训练将足以获得可比较的性能指标。

下表，即表 1.3.3，显示了不同的网络配置和相应的性能指标。在层下，显示了第 1 至 3 层的单元数。对于每个优化器，都使用了 tf.keras 中的默认参数。可以观察到改变正则器、优化器和每层单元数的效果。表 1.3.3 中另一个重要的观察结果是，更大的网络不一定能转化为更好的性能。增加这个网络的深度在训练和测试数据集的准确性方面都没有显示出额外的好处。另一方面，较少的单元数，如 128，也会降低测试和训练的准确性。当去掉正则器，每层使用 256 个单元时，可以获得 99.93% 的最佳训练精度。然而，由于网络的过度拟合，测试精度要低得多，为 98.0%。最高的测试精度是使用 Adam 优化器和 Dropout(0.45)，达到 98.5%。从技术上讲，鉴于其训练准确率为 99.39%，仍然存在一定程度的过拟合。256-512-256、Dropout(0.45) 和 SGD 的训练和测试精度都是 98.2%。去掉 Regularizer 和 ReLU 层的结果是它的性能最差。一般来说，我们会发现 Dropout 层的性能比 l2 好。下表展示了一个典型的深度神经网络在调优过程中的表现。

这个例子表明，有必要改进网络结构。在下一节讨论了 MLP 分类器模型的总结后，我们将介绍另一个 MNIST 分类器。下一个模型是基于 CNN 的，并展示了测试准确性的明显改善。

3.4.7. 模型总结

使用 Keras 库为我们提供了一种快速的机制，通过调用以下内容来仔细检查模型描述。

model.summary()

下面的清单 1.3.3 显示了拟议网络的模型摘要。它总共需要 269,322 个参数。考虑到我们有一个对 MNIST 数字进行分类的简单任务，这是很可观的。MLPs 的参数效率不高。通过关注感知器的输出是如何计算的，可以从图 1.3.4 中计算出参数的数量。从输入到密集层。$784 \times 256 + 256 = 200,960$。从第一致密层到第二致密层：$256 \times 256 + 256 = 65792$。从第二密集层到输出层。$10 \times 256 + 10 = 2,570$。总数为 $269,322$。

Listing 3.16: 一个 MLP MNIST 数字分类器模型的总结

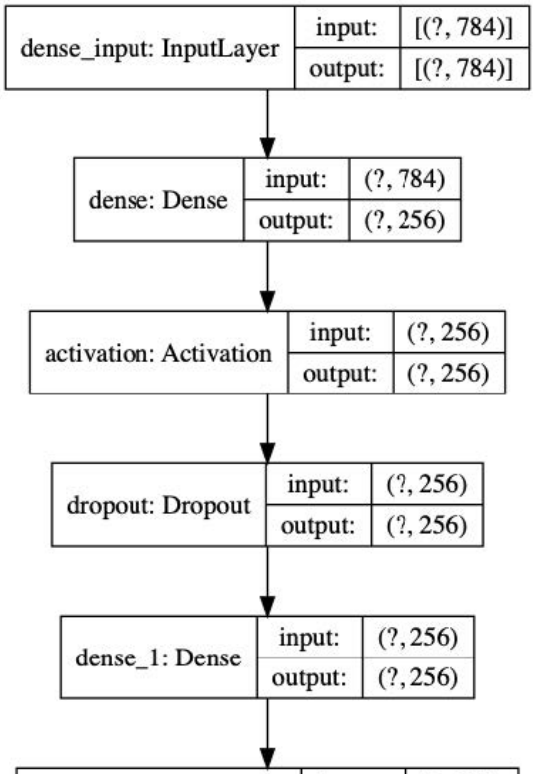
```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               200960
activation_1 (Activation)    (None, 256)               0
dense_2 (Dense)              (None, 256)               65792
activation_2 (Activation)    (None, 256)               0
activation_3 (Activation)    (None, 10)                0
=================================================================
Total params: 269,322
Trainable params: 269,322
Non-trainable params: 0
```

另一种验证网络的方法是

但以图形方式显示了每层的互连和 I/O。图3.10显示了 MLP 对 MNIST 数据集数字分类器的流程。你会发现，这与 summary() 的结果类似的结果，但用图形显示了各层的互连和 I/O。

表 **3.1:** Distribution of the workload

Layers	Regularizer	优化器	ReLU	训练准确率 (%)	测试准确率 (%)
256-256-256	None	SGD	None	93.65	92.5
256-256-256	L2(0.001)	SGD	Yes	93.35	98
256-256-256	L2(0.01)	SGD	Yes	96.90	96.7
256-256-256	None	SGD	Yes	99.93	98.0
256-256-256	Dropout(0.4)	SGD	Yes	98.23	98.1
256-256-256	Dropout(0.45)	SGD	Yes	98.07	98.1
256-256-256	Dropout(0.5)	SGD	Yes	97.68	98.1
256-256-256	Dropout(0.6)	SGD	Yes	97.11	97.9
256-512-256	Dropout(0.45)	SGD	Yes	98.21	98.2
512-512-512	Dropout(0.2)	SGD	Yes	99.45	98.3
512-512-512	Dropout(0.4)	SGD	Yes	98.95	98.3
512-1024-512	Dropout(0.45)	SGD	Yes	98.9	98.2
1024-1024-1024	Dropout(0.4)	SGD	Yes	99.37	98.3
256-256-256	Dropout(0.6)	Adam	Yes	98.64	98.2
256-256-256	Dropout(0.55)	Adam	Yes	99.01	98.3
256-256-256	Dropout(0.45)	Adam	Yes	99.39	98.5
256-256-256	Dropout(0.45)	RMSprop	Yes	98.75	98.1
128-128-128	Dropout(0.45)	Adam	Yes	98.70	97.7



在总结了我们的模型之后，我们对 MLPs 的讨论就结束了。在下一节，我们将建立一个基于 CNN 的 MNIST 数字分类器模型。

3.5. 卷积神经网络 (CNN)

我们现在要进入第二个人工神经网络，即 CNN。在这一节中，我们将解决同样的 MNIST 数字分类问题，但这次是使用 CNN。图3.11显示了我们将用于 MNIST 数字分类的 CNN 模型，而其实现则在 lstlisting1.17 中说明。为了实现 CNN 模型，需要对以前的模型进行一些改变。对于灰度 MNIST 图像来说，输入张量现在有新的维度（高度、宽度、通道）或（image_size, image_size, 1）=（28, 28, 1），而不是有一个输入向量。需要调整训练和测试图像的大小以符合这个输入形状的要求。

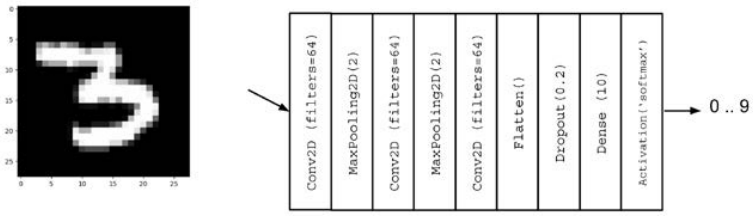


图 3.11: 用于 MNIST 数字分类的 CNN 模型

对于该流程图3.11的执行程序：

Listing 3.17: 用于 MNIST 数字分类的 CNN 模型

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense, Dropout
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.datasets import mnist
# load mnist dataset
num_labels = len(np.unique(y_train))
# convert to one-hot vector
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
# input image dimensions
x_train = np.reshape(x_train,[-1, image_size, image_size, 1])
x_test = np.reshape(x_test,[-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
# network parameters
# image is processed as is (square grayscale)
```

这里的主要变化是使用了 Conv2D 层。ReLU 激活函数已经是 Conv2D 的一个参数。当批量归一化层包含在模型中时，ReLU 函数可以作为激活层被带出来。批量归一化在深度 CNN 中使用，这样可以利用大的学习率，而不会在训练中造成不稳定。

3.5.1. 卷积

如果在 MLP 模型中，单元的数量是密集层的特征，那么内核就是 CNN 操作的特征。如图3.12所示，内核可以被看作是一个从左到右、从上到下滑过整个图像的长方形补丁或窗口。这种操作被

称为卷积。它将输入图像转化为一个特征图，这个特征图是内核从输入图像中学到的东西的代表。然后，该特征图被转化为后续层的另一个特征图，以此类推。每个 Conv2D 生成的特征图的数量是由过滤器参数控制的。

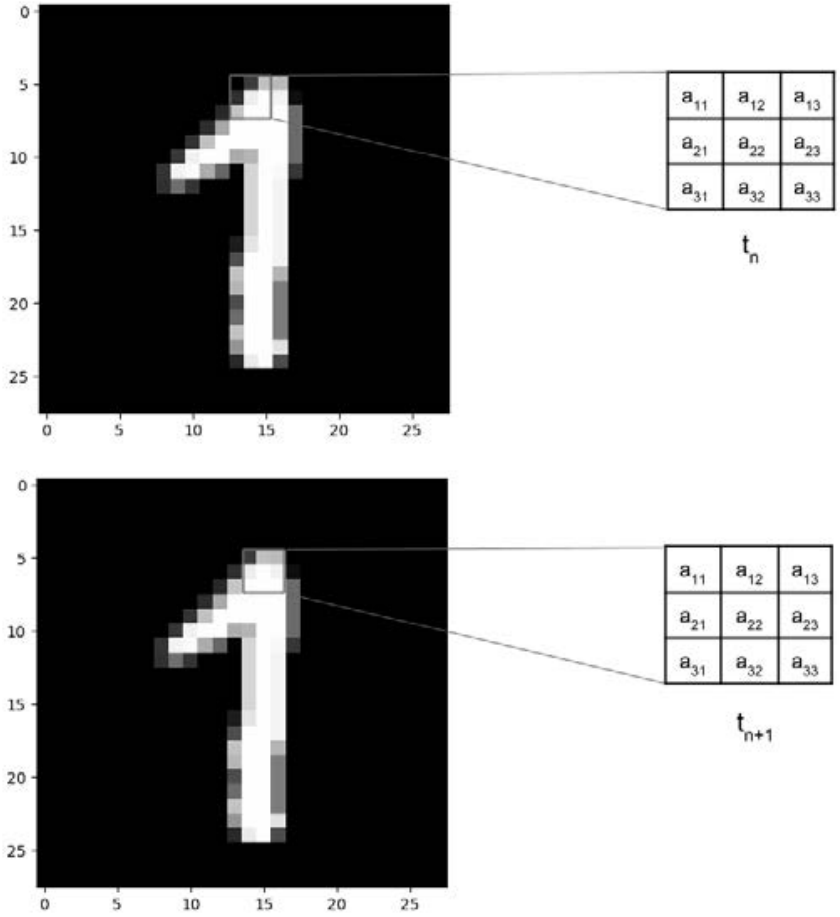


图 3.12: 一个 3×3 的运算内核对 MNIST 数字图像进行卷积。

卷积显示在步骤 t_n 和 t_{n+1} 中，其中内核向右移动了 1 个像素的跨度。图3.13显示了卷积的计算过程。

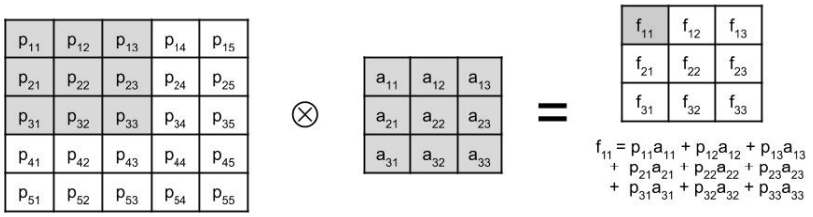


图 3.13: 卷积操作显示了特征图的一个元素是如何计算出来的

为了简单起见，图3.13中展示了一个 5×5 的输入图像（或输入特征图），其中应用了 3×3 的

核。卷积后的特征图被显示出来。特征图中的一个元素的值有阴影。你会注意到产生的特征图比原始输入图像要小，这是因为卷积只在有效元素上进行。内核不能超出图像的边界。如果输入的尺寸应与输出的特征图相同，Conv2D 接受选项 $padding =' same'$。在输入的边界处用零填充，以保持卷积后的尺寸不变。

3.5.2. 池化操作

最后一个变化是增加了一个 MaxPooling2D 层，参数 $pool_size = 2$。MaxPooling2D 压缩了每个特征图。每个大小为 $pool_size \times pool_size$ 的补丁都被压缩为 1 个特征图点。该值等于补丁内的最大特征点值。MaxPooling2D 在下图中显示了两个斑块的情况。

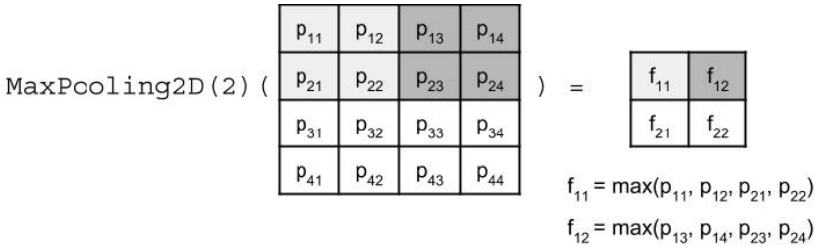


图 3.14: MaxPooling2D 执行的操作。为了简单起见，输入的特征图是 4×4，结果是 2×2 的特征图。

MaxPooling2D 的意义在于减少了特征图的大小，从而转化为增加了感受野的大小。例如，在 MaxPooling2D(2) 之后，2×2 的内核现在大约与 4×4 的补丁进行卷积。CNN 已经为不同的感受野大小学习了一套新的特征图。还有其他汇集和压缩的手段。例如，为了达到 MaxPooling2D(2) 那样的 50％的尺寸缩减，AveragePooling2D(2) 采取了补丁的平均值，而不是寻找最大值。Strided convolution, Conv2D(strides=2,...), 将在卷积过程中每两个像素跳过一次，仍然会有同样的 50% 的大小减少效果。每种缩减技术的效果都有细微的差别。在 Conv2D 和 MaxPooling2D 中，pool_size 和 kernel 都可以是非正方形。在这些情况下，必须同时指出行和列的大小。例如，$pool_size = (1, 2)$，$kernel = (3, 5)$。最后一个 MaxPooling2D 操作的输出是一个特征图的堆栈。Flatten 的作用是将堆叠的特征图转换成适合 Dropout 或 Dense 层的矢量格式，类似于 MLP 模型的输出层。在下一节，我们将评估训练好的 MNIST CNN 分类器模型的性能。

3.5.3. 性能评估和模型总结

如 Listing 1.18 所示，清单 1.17 中的 CNN 模型需要的参数数量较少，为 80226 个，而使用 MLP 层时为 269322 个。conv2d_1 层有 640 个参数，因为每个核有 3×3=9 个参数，64 个特征图中的每个都有一个核和一个偏置参数。其他卷积层的参数数可以用类似的方法计算。

Listing 3.18: CNN MNIST 数字分类器的总结

```
Layer (type)                 Output Shape              Param #
conv2d_1 (Conv2D)            (None, 26, 26, 64)        640
max_pooling2d_1 (MaxPooling2)   (None, 13, 13, 64)          0
conv2d_2 (Conv2D)            (None, 11, 11, 64)        36928
conv2d_3 (Conv2D)            (None, 3.3, 3, 64)        36928
flatten_1 (Flatten)          (None, 576)               0
activation_1 (Activation)    (None, 10)                0
=================================================================
```

```
Total params: 80,266
Trainable params: 80,266
Non-trainable params: 0
```

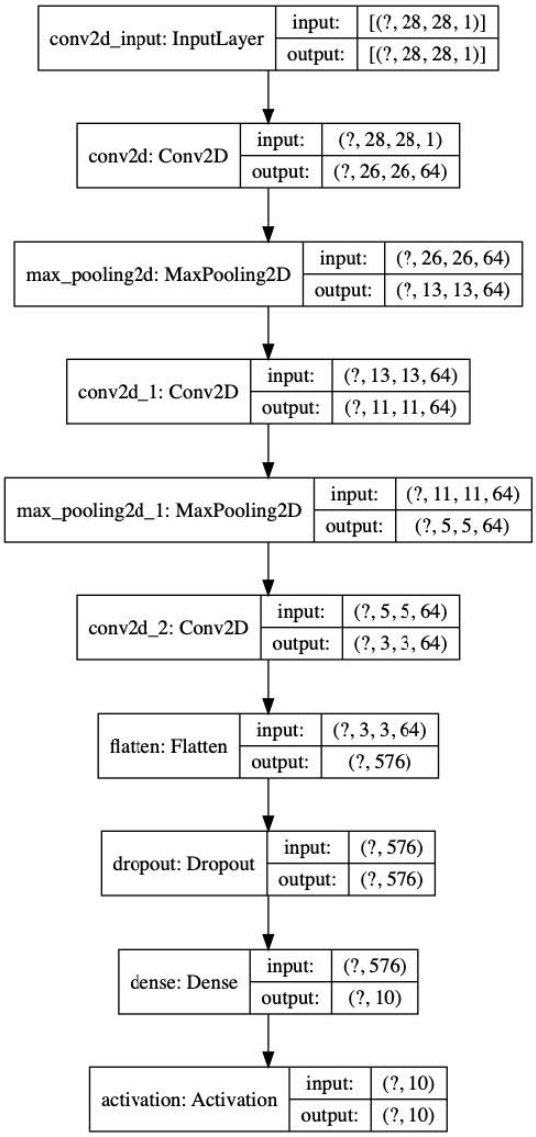


图 3.15: CNN 对于 MNIST 数字分类器的工作过程说明

表4.1显示，对于每层有 64 个特征图的 3 层网络，使用 Dropout=0.2 的 Adam 优化器可以达到 99.4% 的最大测试精度。CNNs 的参数效率更高，并且比 MLPs 有更高的准确性。同样，CNN 也适用于从连续的数据、图像和视频中学习表征。

表 3.2: 不同的 CNN 网络配置和 CNN MNIST 数字分类器的性能测量。

Layers	Optimizer	Regularizer	训练准确率 (%)	测试准确率 (%)
64-64-64	SGD	Dropout(0.2)	97.76	98.50
64-64-64	RMSprop	Dropout(0.2)	99.11	99.00
64-64-64	Adam	Dropout(0.2)	99.75	99.40
64-64-64	Adam	Dropout(0.4)	99.64	99.30

3.6. 递归神经网络（RNN）

我们现在要看的是三个人工神经网络中的最后一个，即 RNN。RNN 是一个网络家族，适用于学习序列数据的表示，如自然语言处理（NLP）中的文本或仪器仪表中的传感器数据流。虽然每个 MNIST 数据样本在本质上不是连续的，但不难想象，每张图像都可以被解释为像素的行或列的序列。因此，基于 RNN 的模型可以将每张 MNIST 图像处理为一串 28 元素的输入向量，时间步长等于 28。下面列出了图3.16中 RNN 模型的代码。

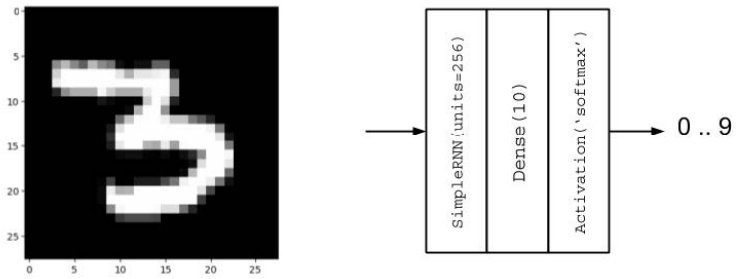


图 3.16: RNN model for MNIST digit classification

Listing 3.19: RNN 模型的代码

```
x_train = np.reshape(x_train,[-1, image_size, image_size])
x_test = np.reshape(x_test,[-1, image_size, image_size])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
# network parameters
input_shape = (image_size, image_size)
batch_size = 128
units = 256
dropout = 0.2
# model is RNN with 256 units, input is 28-dim vector 28 timesteps
model = Sequential()
model.add(SimpleRNN(units=units,
                    dropout=dropout,
                    input_shape=input_shape))
model.add(Dense(num_labels))
model.add(Activation('softmax'))
model.summary()
plot_model(model, to_file='rnn-mnist.png', show_shapes=True)
```

```
# loss function for one-hot vector
# use of sgd optimizer
# accuracy is good metric for classification tasks
 model.compile(loss='categorical_crossentropy',
               optimizer='sgd',
               metrics=['accuracy'])
# train the network
model.fit(x_train, y_train, epochs=20, batch_size=batch_size)
_, acc = model.evaluate(x_test,
                        y_test,
                        batch_size=batch_size,
                        verbose=0)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

RNN 分类器与之前的两个模型之间有两个主要区别。首先是 input_shape = (image_size, image_size)，这实际上是 input_shape = (timesteps, input_dim)，或者是一连串 timesteps 长度的 input_dim-dimension 向量。其次是使用 SimpleRNN 层来代表 $units = 256$ 的 RNN 单元。units 变量代表输出单元的数量。如果 CNN 的特点是在输入特征图上进行核的卷积，那么 RNN 的输出不仅是当前输入的函数，也是之前输出或隐藏状态的函数。由于前一个输出也是前一个输入的函数，当前输出也是前一个输出和输入的函数，以此类推。Keras 中的 SimpleRNN 层是真正 RNN 的简化版本。下面的方程式3.13描述了 SimpleRNN 的输出。

$$h_t = tanh(B + Wh_{t-1} + Ux_t) \tag{3.13}$$

在这个方程中，b 是偏置，而 W 和 U 分别称为循环核（前一个输出的权重）和核（当前输入的权重）。下标 t 用来表示序列中的位置。对于 $units = 256$ 的 SimpleRNN 层，参数总数为 $256 + 256 \times 256 + 256 \times 28 = 72,960$，对应于 b、W 和 U 的贡献。下图3.17显示了 SimpleRNN 和 RNN 用于分类任务时的图示。使 SimpleRNN 比 RNN 更简单的是在计算 softmax 函数之前没有输出值 $o_t = Vh_t + c$。

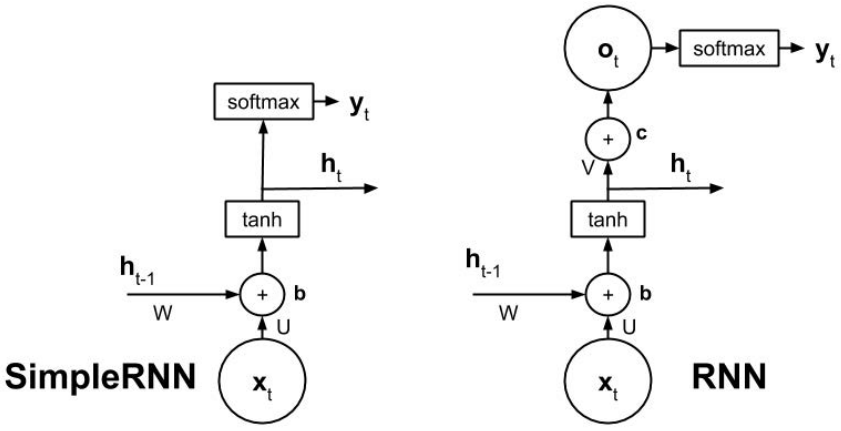


图 3.17: SimpleRNN 和 RNN 的示意图

与 MLP 或 CNN 相比，RNN 最初可能比较难理解。在 MLP 中，感知器是基本单元。一旦理解了感知器的概念，MLP 就只是一个感知器网络。在 CNN 中，内核是一个补丁或窗口，它通过特征图滑动来生成另一个特征图。在 RNN 中，最重要的是自循环的概念。事实上，只有一个单元。

出现多个单元的错觉是因为每个时间段都有一个单元存在，但实际上它只是同一个单元在重复使用，除非网络被解卷。RNNs 的底层神经网络是跨单元共享的。

Listing 1.20 中的总结表明，使用 SimpleRNN 需要的参数数量较少。

Listing 3.20: CNN MNIST 数字分类器的总结

```
1
2 Layer (type)                                 Output Shape              Param #
3 =================================================================
4 simple_rnn_1 (SimpleRNN)                (None, 256)               72960
5 activation_1 (Activation)                (None, 10)                0
6 =================================================================
7 Non-trainable params: 0
8        \label{L1.5.2}
```

图3.18是 RNN MNIST 数字分类器的图形描述。该模型非常简明。

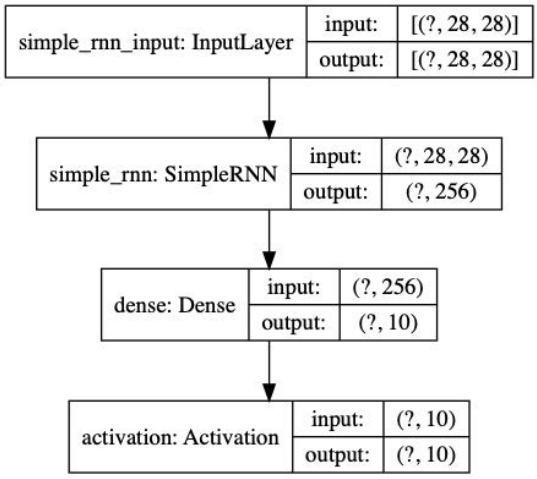


图 3.18: RNN 网络对 MNIST 数字分类器的过程

表3.3显示 SimpleRNN 在所介绍的网络中准确率最低。

表 3.3: 不同的 SimpleRNN 网络配置参数和性能指标

Layers	Optimizer	Regularizer	训练准确率 (%)	测试准确率 (%)
256	SGD	Dropout(0.2)	97.26	98.00
256	RMSprop	Dropout(0.2)	96.72	97.60
256	Adam	Dropout(0.2)	96.79	97.40
256	SGD	Dropout(0.2)	97.88	98.30

在许多深度神经网络中，RNN 家族的其他成员更常被使用。例如，长短时记忆（LSTM）已被用于机器翻译和问题回答问题中。LSTM 解决了长期依赖或记忆过去相关信息到现在输出的问题。

与 RNN 或 SimpleRNN 不同，LSTM 单元的内部结构更加复杂。图 1.5.4 是 LSTM 的示意图。LSTM 不仅使用现在的输入和过去的输出或隐藏状态，而且还引入了一个单元状态，即 st，它将信

息从一个单元带到另一个。单元状态之间的信息流由三个门控制，即 ft、it 和 qt。这三个门的作用是决定哪些信息应该被保留或替换，以及过去和当前输入的信息量应该对当前的单元状态或输出做出贡献。我们不会在本书中讨论 LSTM 单元内部结构的细节。然而，关于 LSTM 的直观指南可以在 http://colah.github.io/ posts/2015-08-Understanding-LSTMs 找到。

LSTM() 层可以作为 SimpleRNN() 的落地替换。如果 LSTM 对于手头的任务来说是多余的，可以使用一个更简单的版本，称为门控递归单元（GRU）。GRU 通过将单元状态和隐藏状态结合在一起简化了 LSTM。GRU 还可以减少一个门的数量。GRU() 函数也可以作为 SimpleRNN() 的替代品使用。

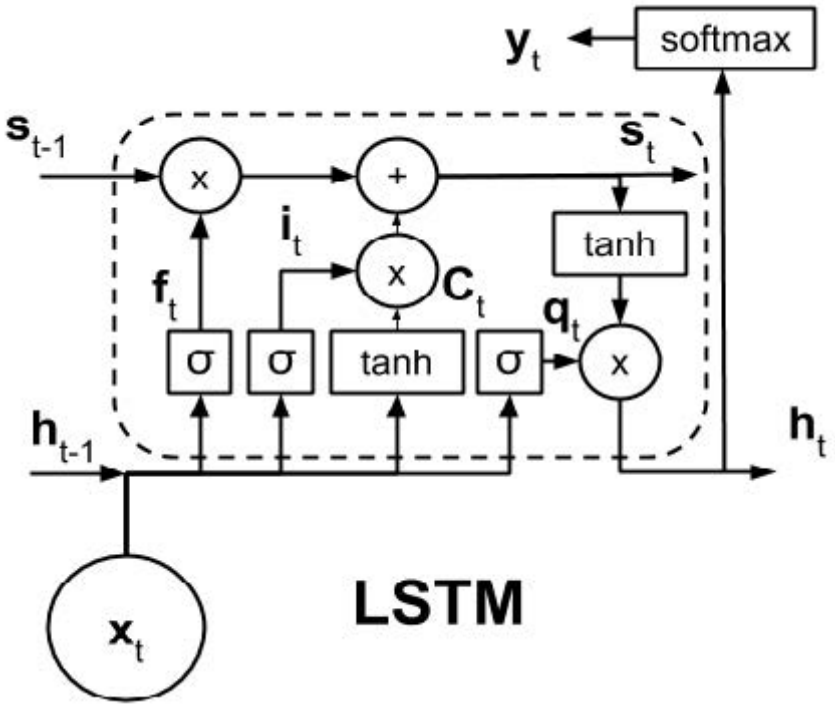


图 3.19: LSTM 工作流程示意图

还有许多其他方法来配置 RNN。一种方法是制作一个双向的 RNN 模型。默认情况下，RNN 是单向的，即当前输出只受过去状态和当前输入的影响。

在双向的 RNN 中，未来的状态也可以通过允许信息向后流动来影响现在和过去的状态。过去的输出会根据收到的新信息按需更新。RNNs 可以通过调用一个封装函数来实现双向性。例如，双向 LSTM 的实现是 Bidirectional(LSTM())。

对于所有类型的 RNN，增加单元的数量也会增加容量。然而，另一种增加容量的方法是通过堆叠 RNN 层。但应该注意的是，作为一般的经验法则，模型的容量应该只在需要时才增加。过多的容量可能会导致过度拟合，因此，可能会导致更长的训练时间和预测过程中更慢的性能。

3.7. 总结

本章概述了三种深度学习模型–MLP、RNN、CNN，还介绍了 TensorFlow 2 tf.keras，这是一个用于快速开发、训练和测试深度学习模型的库，适用于生产环境。还讨论了 Keras 的 Sequential API。在下一章，将介绍 Functional API，这将使我们能够专门为高级深度神经网络建立更复杂的模型。

本章还回顾了深度学习的重要概念，如优化、正则化和损失函数。为了便于理解，这些概念是在 MNIST 数字分类的背景下提出的。

本章还讨论了使用人工神经网络进行 MNIST 数字分类的不同解决方案，特别是 MLP、CNN 和 RNN，它们是深度神经网络的重要构件，并讨论了它们的性能指标。

在了解了深度学习的概念以及如何用 Keras 作为工具后，我们现在已经具备了分析高级深度学习模型的能力。在下一章讨论了 Functional API 之后，我们将转向流行的深度学习模型的实现。随后的章节将讨论选定的高级主题，如自回归模型（autoencoder、GAN、VAE）、深度强化学习、物体检测和分割，以及使用互信息的无监督学习。随之而来的 Keras 代码实现将在理解这些主题方面发挥重要作用。

3.8. 参考文献

1. Chollet, François. Keras (2015). https://github.com/keras-team/keras.

2. LeCun, Yann, Corinna Cortes, and C. J. Burges. MNIST handwritten digit database. ATT Labs [Online]. Available: http://yann.lecun.com/exdb/ mnist2 (2010).