# 第 1 章

# 神经网络、深度学习和大模型

学习目标

- 了解神经网络、深度学习和大模型的概念
- 知道神经网络、深度学习和大模型这三者之间的关系
- 初步了解深度学习的框架：Pytorch
- 搭建 Python 及 Pytorch 开发环境

## 1.1 神经网络和深度学习

神经网络，也称人工神经网络，是由大量节点（也称神经元）组成的非线性的数据处理系统，具有自主学习和高效求解等特性。深度学习的物质基础是深度神经网络，这里的“深度”，是指神经网络具有较深层数的数据处理节点，而深度学习的技术基础则是各种学习、训练和调优算法。

基于深度学习的求解过程一般包含两个步骤，第一是搭建（深度）神经网络模型；第二是用相关算法训练并调优该模型，构建好模型后，则可以进行图片处理等动作。事实上，Pytorch 等框架已经封装好了搭建神经网络的动作和相关算法，程序员可以用此高效地实现相关开发动作。

### 1.1.1 神经元和神经网络

神经网络是一种仿生物神经网络的结构的计算模型，可以用来近似计算函数，也可以用来处理数据、图片和视频等信息。

神经网络一般由若干神经元组成，而神经元是神经网络中的计算和存储单元，即会对输入值进行计算，完成计算后会暂存结果并传递到下一层。单个神经元模型的结构如图 1.1 所示。

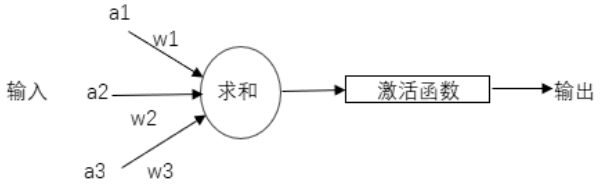


图 1.1　单个神经元模型

神经元一般由输入层、权重、输出层、神经元内部的处理函数和激活函数构成。图 1.1 给出的神经元的输出层可以接纳 3 个输入，分别是 a1、a2 和 a3，每个输入对应一个权重，分别是 w1、w2 和 w3。假设该神经元的处理函数是求和，那么该神经元会根据权重，针对输入进行求和计算。

具体得到 a1×w1+a2×w2+a3×w3 的求和结果，然后再用激活函数对该求和结果进行处理，并把处理后的结果输出到下一层。这里的激活函数一般会采用非线性的函数，引入激活函数的目的是，让神经元乃至由神经元构成的神经网络，能处理复杂的非线性的输入数据。

图 1.2 给出了由 5 个神经元组成的 2 层神经网络模型，事实上，神经网络中的不同神经元一般会包含不同的处理函数，而不同层级的神经元交互时一般也需要考虑权重因素。一般来说，层数越多、神经元个数越多的神经网络，处理数据的能力也就越强大。

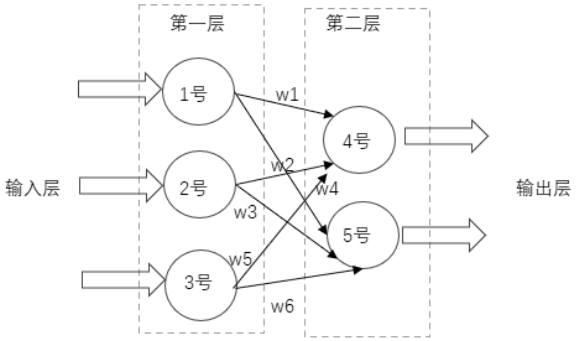


图 1.2　由 5 个神经元构成的神经网络效果图

## 1.1.2　深度神经网络与深度学习

一般来说，为了提升神经网络处理数据的性能和准确性，往往会在输入层和输出层之间搭建多个层级的神经元，这些层级对使用者来说是不透明的，所以一般也称为“隐藏层”，具体效果如图 1.3 所示。

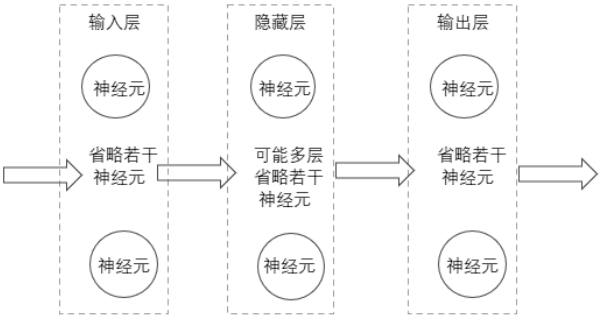


图 1.3　含隐藏层的神经网络效果图

从实践角度来看，神经网络隐藏层的层次数越多，该神经网络的功能就越强大。这里，一般会把包含多层的神经网络叫作“深度神经网络”，而针对深度神经网络的训练、分析和预测等动

作，可以称之为“深度学习”。

深度学习的过程一般可以分解成搭建、训练和预测等几个关键步骤。

（1）根据实际需求，搭建深度神经网络，比如具体可以搭建“深度卷积神经网络”或“深度对抗神经网络”等，从中可以看到，深度神经网络是深度学习的载体。

（2）预设各神经元中的算法和参数，在此基础上用训练数据训练该神经网络。训练过程中会调整各神经元中包含的算法和参数，甚至还会调整神经元的个数和层数。

（3）训练完成后，该深度神经网络会具有层次和数量都相对固定的神经元，同时各神经元里包含的算法和参数也会被训练得相对固定，此时该深度神经网络模型就可以用来预测。

（4）在预测过程中，一般还会根据结果对该深度神经网络进行调整。

上述过程非常复杂，而且会涉及一些较深的数学技能，但事实上，程序员可以通过 Pytorch 等框架，通过调用这些框架里的接口方法进行深度学习的相关操作，比如在训练好模型的基础上进行图片识别或自然语言处理。

在这个过程中，程序员如果能了解相关的神经网络和数学等相关技能，将会更高质量地使用 Pytorch 等框架进行深度学习的相关开发工作。

### 1.1.3　深度学习的应用场景

（1）深度学习可被用在图片处理、图片识别和人脸识别等场景。

（2）深度学习可被用在自然语言处理的场景，比如可以进行情感分析、文本分类和机器翻译等操作。

（3）深度学习可被用在医疗保健的场景，比如可以进行医学图片分析和病理诊断等操作。

（4）深度学习可被用在金融分析的场景，比如可以进行股票预测、信用等级分析评估和金融风险控制管理等操作。

## 1.2　深度学习和大模型

在 AI 领域，模型可以理解成基于某种 AI 结构或 AI 算法，具有特定功能的模块。比如可以是基于分类算法的股票分析模型，或者是基于神经网络的图片识别模型。

而大模型是指含数百万到数十亿参数的深度学习模型。搭建大模型的深度神经网络一般具有较深的隐藏层，即包含海量的神经元节点，同时会用海量的数据训练并调优大模型。

大模型里的参数可以是针对每个神经元的权重，在训练过程中，会通过各种优化算法（如梯度下降）来调整这些参数，从而最大程度地减小输出值和实际值的差距。

完成训练后，该大模型内部的参数就会相对固定，这样该大模型就可以根据输入的参数，进行自然语言翻译、数据预测或图片识别等工作。

从上文中可以看到，大模型和深度学习技术具有紧密联系，一方面，大模型的载体是较为复杂的深度神经网络；另一方面，大模型的训练和调优过程会用到深度学习技术。

在开发场景，程序员可以用 Pytorch 框架来搭建、训练和调优深度学习模型，并用该模型进行

图片识别等操作。在此基础上，程序员还可以用 Pytorch 框架对大模型进行开发和微调工作。

当下比较常见的大模型产品有 ChatGPT 和文心一言等，这些大模型给人们的工作和生活带来了极大的便利。

# 1.3 实现深度学习的 Pytorch 框架

Pytorch 是一个基于 Python 的深度学习框架，所谓“框架”，是指在其中封装了大量机器学习和深度学习的实现接口。在 Python 语言里，支持 Pytorch 框架的第三方包是“Torch”。

## 1.3.1 Pytorch 简介

Pytorch 框架的前身是 Torch，Torch 是用 Lua 语言实现的，被广泛应用于视觉处理和自然语言处理等各种机器学习领域，但在 2016 年之前，这个框架在 Python 里没有得到较好的支持。

2016 年，Facebook 公司在 Torch 的基础上研发出了 Pytorch 的 Alpha 版本，随后又在此基础上不断迭代，升级了各种功能。2018 年，Pytorch 的 1.0 版本正式发布，该版本发布后，被广泛应用在各种机器学习和深度学习等开发场景。

当下 Pytorch 已经升级到 2.0 版本，该版本是在 Pytorch 2022 大会上正式发布的。和之前的 1.x 版本相比，该版本包含了大量新功能，同时提升了创建和训练模型的性能。

Pytorch 框架的出现和发展，事实上降低了深度学习的门槛。经过若干年的实践和发展，当前 Pytorch 被广泛应用到数据分析领域、数据挖掘、模式识别、基因数据分析和图片识别等领域。

## 1.3.2 Pytorch 的常用模块

Pytorch 框架包含了以下几个常用功能模块：

（1）torch.Tensor 模块：封装了不同数值类型的张量，以及针对张量的各种操作。

（2）torch.nn 模块：封装了搭建神经网络的常用方法，比如封装了与搭建神经网络相关的卷积、激活和求损失函数的方法。

（3）torch.optim 模块：封装了优化器相关的方法。

（4）torch.jit 模块：承担了“即时编译器”的作用，通过该模块导出的静态图，能被 Java 等编程语言使用。

（5）torch.autograd 模块：封装了自动求导等功能，该模块被广泛应用在神经网络训练的过程中。

（6）torch.multiprocessing 模块：封装了多线程操作的相关方法，通过这种方法，程序员能提高模型的训练效率。

（7）torch.utils 模块：封装了读取数据集和训练测试数据集等方法。

（8）torch.random 模块：封装了生成随机数的相关方法。

### 1.3.3　搭建 Python 开发环境

由于 Pytorch 框架是基于 Python 语言的，为了使用这个库，首先需要搭建 Python 开发环境。搭建 Python 开发环境的步骤有 3 个：第一，下载并安装 Python 解释器；第二，下载并安装 Python 集成开发环境；第三，下载常用的包含 Pytorch 框架等的第三方库。

Python 的代码是由解释器执行，所以在搭建环境时，首先需要安装解释器，可以从官网 https://www.python.org/downloads/windows/ 下载解释器，由于本书是在 Windows 操作系统上开发 Python 代码，所以是下载 Windows 版本。同时，出于兼容性方面的考虑，本书使用的是 3.10 版本的解释器。

下载后根据提示安装解释器，安装完成后，可在安装路径里看到 python.exe，比如本机的安装路径是 C:\Users\admin\AppData\Local\Programs\Python\Python310，在其中能看到 python.exe。建议把该路径添加到环境变量 Path 里，这样就能在任何路径位置运行 python.exe。

Python 解释器安装完成后，理论上就能通过在 TXT 文本文件里编写 Python 代码，随后再通过解释器运行。但这样做的效率不高，所以一般建议在 Python 集成开发环境里开发、调试并运行 Python 代码，本书所用的集成开发环境是 PyCharm。

可以到官网 https://www.jetbrains.com.cn/pycharm/ 去下载 PyCharm 集成开发环境，这里建议下载社区版，下载完成后安装即可。

为了开发实现各种功能的 Python 代码，除了可以使用 Python 解释器所提供的默认库，还需要根据实际情况，安装各种第三方库，比如需要安装支持科学运算的 Numpy 库。安装第三方库的步骤如下：

第一步，打开一个 cmd 命令窗口，并在其中运行 cd 等命令，进入 Python 解释器所在的路径，比如本书是 C:\Users\admin\AppData\Local\Programs\Python\Python310，在该路径下，进入 Scripts 路径，可以看到用于安装第三方库的 pip3 命令。

第二步，通过运行“pip3 install 库名”命令的方式，安装第三方库，比如要安装 numpy，对应的命令是 pip3 install numpy。安装完成后，能通过运行 pip3 list 命令，确定所安装的第三方库，并能查看对应的版本。

### 1.3.4　简单安装 Pytorch 框架

可通过上文提到的 pip3 命令安装 Pytorch 框架，具体命令如下所示。

```
1  pip3 install torch torchvision torchaudio
```

顾名思义，Pytorch 是由“Py”和“torch”构成的，其中“Py”表示 Python，说明 Pytorch 框架是基于 Python 语言的，而“torch”则表示该框架在 Python 内的库名为“torch”，所以这里用 pip3 命令安装 Pytorch 框架时，输入的表示库名的参数为“torch”。

此外，torchvision 和 torchaudio 是 torch 库所必需的前置支持库，所以在这里一起安装。由于这里并没有指定相关库的版本，所以 pip3 命令会自动下载并安装在兼容环境下的最新版本。

需要注意的是，本书在后文讲解相关代码时，可能还需要安装其他第三方库，到时候依然是通过这里给出的 pip3 install 命令来安装其他第三方库的。

安装完成后，可到 Pycharm 集成开发环境里创建一个名为 chapter1 的项目，并在其中创建

一个名为 CheckVersion.py 的 Python 文件，并在其中编写如下代码。

```
1  import torch
2  print(torch.__version__)
```

这里是在第 1 行导入了 Pytorch 框架的支持库 torch，并通过第 2 行代码输出该库的版本信息。该代码运行后，会显示如下输出。

```
1  2.3.0+cpu
```

通过上述输出结果大家能看到，笔者在运行上述代码时，torch 库的最新版本是 2.3.0。大家在自己的计算机上运行时，或许会看到其他的版本号。总之，如果能看到版本号，就说明 Pytorch 框架所对应的 torch 库已在计算机上成功安装。

但是这里大家看到是 cpu 字样，这说明当下 Pytorch 框架只是支持 CPU 环境，而并没有支持 GPU 环境，下文将给出搭建支持 GPU 的 Pytorch 环境的具体做法。

# 1.4 搭建支持 GPU 的 Pytorch 环境

如果大家的计算机不支持 GPU 功能，那可以用上文给出的步骤，安装基于 CPU 的 Pytorch 框架。由于本书给出代码，其运算量并不大，所以这样的话，并不会影响学习和运行本书的代码。

但是，还是建议安装支持 GPU 的 Pytorch 框架，因为相对 CPU 版本而言，GPU 版本的计算速度会快很多。

## 1.4.1 GPU 和 CUDA

GPU（Graphics Processing Unit）是图形处理器的英文缩写，是显卡的重要组成部分，能高效地进行图形渲染和视频处理等动作。

当下一些版本的 NVIDA 显卡支持 GPU 功能。大家可以打开“任务管理器”窗口，切换到“性能”选项卡，如果能看到如图 1.4 所示的 GPU 性能数据，那么说明该台计算机支持 GPU 功能。

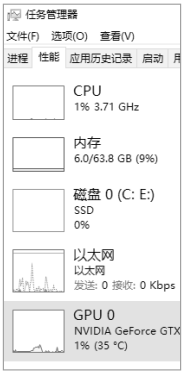


图 1.4 查看计算机是否支持 GPU 功能

CUDA（Compute Unified Device Architecture）是基于 GPU 的并行计算平台和编程模型，这种模型能利用 GPU 强大的运算能力，高效地进行图片或视频方面的运算。

为了确认自己的计算机是否支持 CUDA，大家可以打开命令行窗口，并在其中运行如下命令。

```
1  nvcc --version
```

如果支持 CUDA，该命令能运行通，并能看到 CUDA 的版本信息。比如笔者在自己的计算机上运行该命令，确认了笔者计算机上的 CUDA 版本是 12.1。

如果大家安装的 Pytorch 框架是基于 GPU 版本，那么就可以通过 CUDA 模型，利用 GPU 处理能力，提升各种复杂运算的速度。

## 1.4.2　安装基于 GPU 的 Pytorch

第一步，下载并安装 CUDA Toolkit 工具。

如果计算机支持 GPU 和 CUDA，那么可以到官网 https://developer.nvidia.com/cuda-toolkit-archive 去下载 CUDA 的 Toolkit，这里请对应地下载和自己的计算机的 CUDA 版本匹配的 .exe 版本。

下载完成后，可以按照提示逐步安装。在安装过程中，不需要修改安装路径。

第二步，下载并安装 cnDNN 工具。

cnDNN 是面向深度学习的 GPU 加速库，可到 https://developer.nvidia.cn/rdp/cudnn-archive 等处下载这个加速库。下载完成后，也可以按照提示逐步安装，这里也不需要修改安装路径。

第三步，到 https://pytorch.org/ 官网去查看安装命令，方法为在如图 1.5 所示的界面中选择 Windows、Pip 和 CUDA12.1 选项，就能在下方看到用 pip3 安装 Pytorch 的安装命令。

```
1  pip3 install torch torchvision torchaudio --index-url https://download.pytorch.
org/whl/cu121
```

如果大家要安装基于 Linux 或 Mac 的 Pytorch 框架，或者要通过 Conda 等方式安装，或者要安装基于 C++ 或 Java 的版本，则可以在如图 1.5 所示的界面中选择对应的选项，那么也能够获得对应的安装命令。

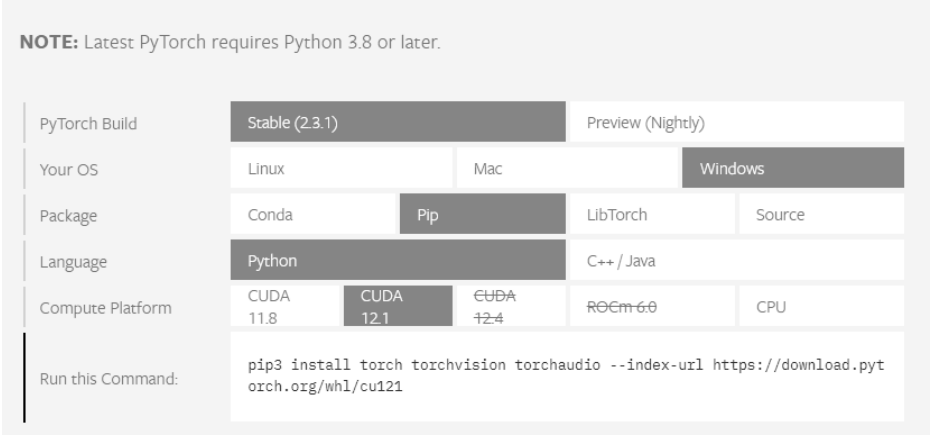


图 1.5　查看 Pytoch 安装命令的效果图

安装完成后，可在上文提到的 CheckVersion.py 里加入下列确认 cuda 的代码，如果该语句返回“gpu”，那就说明基于 GPU 的 Pytorch 框架安装成功。

```
1  print(torch.version.cuda)
```

## 1.5 小结和预告

本章主要讲解了深度学习的入门相关概念，具体讲述了神经网络、深度学习和大模型的概念，并在此基础上讲解了支持深度学习的 Pytorch 框架，以及搭建开发环境的相关步骤。

第 2 章将讲述 Pytorch 框架的基本概念——张量，并在此基础上，使用 Pytorch 框架讲述搭建和训练神经网络的基本知识点。

第 2 章

# 学习张量，搭建神经网络

学习目标

- 了解张量的概念及基本操作
- 掌握深度学习层面，常用的张量运算
- 知道神经网络的常用概念
- 掌握用 Pytorch 搭建简单神经网络的方法

## 2.1 张量的概念和基本操作

张量可以理解成是向量和矩阵的扩展，它可以具有任意维数。在 Pytorch 框架里，张量被封装成一种数据类型。

从结构上来看，Pytorch 框架里的“张量”数据类型和 Numpy 库里的数组类型很相似，但 Pytorch 框架提供了更多针对张量的运算方法。同时，基于张量的运算还可以被 GPU 设备加速。

### 2.1.1 标量、向量、矩阵和张量

标量是指只有数值没有方向的量，比如下面的赋值语句，其中 x 就是一个标量。

```
1  x=1
```

向量则是一组有序的数，通过该序列里的索引，能够找到向量中的每一个数值。比如，可以通过以下代码，用 Python 里的 Numpy 库定义一个向量，向量可以理解成是一个一维数组。

```
1  a = np.array([1,2,3,4])
```

矩阵则可以理解成二维数组，下面的代码用二维数组的形式，定义了一个 3 行 3 列的矩阵。

```
1  x=np.array([[1,2,3], [4,5,6], [7,8,9]])
```

该矩阵的样式如图 2.1 所示。

张量可以理解成是多维的数组，比如图 2.2 展示了由 3 维数组构成的 3 阶张量。

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

图 2.1　矩阵样式

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | [0,1,2] | [0,1,2] | [0,1,2] |
| 1 | [0,1,2] | [0,1,2] | [0,1,2] |
| 2 | [0,1,2] | [0,1,2] | [0,1,2] |

图 2.2　3 阶张量样式

在此基础上，大家可以想象一下 4 阶及更高阶张量的构成形式。

## 2.1.2　张量和深度学习的关系

张量是深度学习里存储数据的容器，可以用来存储待分析和预测的数据。

比如某深度神经网络想要通过分析给定的图片数据，实现图片分类的效果。假设每张图片是由 256×256 个像素构成，每个像素是如下结构的四元组构成，这 4 个数据分别表示红色分量、绿色分量、蓝色分量和透明度。

```
1   (red,green,blue,alpha)
```

那么用来存储每张图片像素数据的其实是一个三维数组，也就是 3 阶张量，大致样式如图 2.3 所示。

| | 0 | 1 | ··· | 255 |
|---|---|---|---|---|
| 0 | [255,235,123,0.5] | [155,125,143,0.7] | | [161,28,85,0.3] |
| 1 | [251,215,173,0.9] | [171,135,138,0.4] | | [161,148,185,0.2] |
| ··· | | | | |
| 255 | [223,125,73,0.8] | [28,24,147,0.6] | | [142,149,135,0.4] |

图 2.3　存储图片的 3 阶张量效果图

再扩展一下，比如该深度神经网络要分析几十万张图片，那就还需要在此基础上添加一阶，用 4 阶张量来存储这些图片的像素数据。

同样，在其他深度学习存储数据的场景里，一般也都无法仅用二维矩阵来存储数据，也需要用高阶的张量来存储数据。所以，掌握针对张量的各种操作，其实是学习深度学习相关模型和技术的必要前提。

## 2.1.3　创建张量

可以通过 torch.tensor 的方法来创建张量，在下面的 CreateTensor.py 范例中，大家能看到创建张量的基本做法。

```
1  import torch
2  # 创建一个 5x3 的张量
3  t = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]])
4  print(t)
5  print(type(t))
6  print(t.dtype)
```

该范例是在第 3 行里，通过 torch.tensor 方法，根据传入的参数，创建了一个 5 行 3 列的张

量，通过第 4 行的输出语句，大家能看到该张量的输出结果。

```
1  tensor([[ 1,  2,  3],
2          [ 4,  5,  6],
3          [ 7,  8,  9],
4          [10, 11, 12],
5          [13, 14, 15]])
```

这里是通过第 5 行代码返回了张量对象的类型，该句语句的输出结果如下，由此大家能看到张量对象的数据类型。

```
1  <class 'torch.Tensor'>
```

这里还通过第 6 行代码返回了该张量对象所存储数据的类型，该句语句的输出结果如下，由此大家能看到，该张量对象存储的是 int64 类型的数据。

```
1  torch.int64
```

### 2.1.4　张量的常见方法

创建张量后，可以通过如下 TensorUsage.py 范例中给出的方法，返回张量内包含的元素个数、张量的形状，以及张量内指定行和指定列的元素。

```
1  import torch
2  # 创建一个 5x3 的张量
3  t = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]])
4  print(t.numel()) # 返回张量的元素个数
5  print(t.shape)# 返回张量的形状
6  print(t[0]) # 返回指定索引行的数据
7  print(t[:,0]) # 返回指定索引列的数据
```

通过上述范例第 4 行给出的 numel 方法，大家能得到张量内的元素个数。通过第 5 行给出的 shape 方法，大家能看到张量的形状。通过第 6 行和第 7 行的代码，大家能看到指定行和指定列的张量数据，该范例的运行效果如下。

```
1  15
2  torch.Size([5, 3])
3  tensor([1, 2, 3])
4  tensor([ 1,  4,  7, 10, 13])
```

### 2.1.5　张量与 Numpy 数据的相互转换

在一些深度学习的开发场景，张量类型的数据会和 Numpy 类型的数据相互转换。通过下面的 TensorAndNumpy.py 范例，大家能掌握这两种数据对象相互转换的做法。

```
1  import numpy as np
2  import torch
3  n = np.ones((2, 2))
4  #Numpy 数据转换成张量
```

```
5  t = torch.from_numpy(n)
6  print(type(t))
7  n2=t.numpy()
8  print(type(n2))
```

通过上述范例第 5 行给出的 from_numpy 方法，大家可以把 Numpy 类型的对象 n 转换成张量类型，通过第 7 行给出的 numpy 方法，大家可以把张量类型的数据转换成 Numpy 类型。上述范例的运行效果如下所示，从中大家能看到相互转换后的效果。

```
1  <class 'torch.Tensor'>
2  <class 'numpy.ndarray'>
```

# 2.2 张量的常见运算

本书将讲述针对张量的常见运算，包括索引和切片运算、转换维度的运算和过滤张量数据的运算。

## 2.2.1 张量的索引操作

在下面的 TensorIndex.py 范例中，大家能看到针对张量的索引操作。

```
1  import torch
2  # 创建一个 5x3 的张量
3  t = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]])
4  print(t[0,1]) # 返回指定索引行索引列的数据
5  # 布尔索引
6  choosed = t > 5
7  choosed_items = t[choosed]
8  print(choosed)
9  print(choosed_items)
```

大家可以像第 4 行给出的代码一样，通过输入张量各维度的索引值，来访问张量内的指定数据。

此外，大家还可以像第 6 行和第 7 行代码一样，通过布尔索引的方式，获取张量内指定条件的数据。具体做法是，可以通过第 6 行的代码设置条件，并通过第 7 行的代码，把大于 5 的数据放入 choosed_items 对象中，上述代码的运行结果如下。

```
1  tensor(2)
2  tensor([[False, False, False],
3          [False, False,  True],
4          [ True,  True,  True],
5          [ True,  True,  True],
6          [ True,  True,  True]])
7  tensor([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

## 2.2.2　张量的切片操作

针对张量做切片的目的是，获取张量内指定行、指定列或指定范围内的数据。通过下面的范例 TensorSlice.py，大家能掌握相关做法。

```
1  import torch
2  # 创建一个 5x3 的张量
3  t = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15]])
4  print(t[:2]) # 返回前两行数据
5  print(t[:,0:2]) # 返回前两列数据
6  print(t[0:2,0:2]) # 返回前两行中前两列的数据
```

在张量内获取切片的关键语法要点是“:”，具体而言，针对张量做行切片的做法如第 4 行的代码所示，这里是通过“:2”的方式，返回了索引为 0 和 1（不包含 2）的行数据。针对张量做列切片的做法如第 5 行的代码所示，这里是返回了索引号是 0 和 1（不包含 2）的列数据。

如果大家要获取张量内指定行指定列的数据，可以用类似第 6 行代码给出的做法，这里是获取前两行范围内的前两列数据。上述代码的运行效果如下。

```
1  tensor([[1, 2, 3],
2          [4, 5, 6]])
3  tensor([[ 1,  2],
4          [ 4,  5],
5          [ 7,  8],
6          [10, 11],
7          [13, 14]])
8  tensor([[1, 2],
9          [4, 5]])
```

## 2.2.3　转换张量的维度

张量是多维矩阵，创建张量后，可以通过 view 和 reshape 等方法转换张量的维度。在下面的 TensorChange.py 范例中，大家能看到转换张量维度的常用技巧。

```
1  import torch
2  # 创建一个 6x2 的张量
3  t = torch.tensor([[1, 2], [3, 4], [5, 6], [7, 8], [9,  10], [11, 12]])
4  print(t.shape)# 输出张量的形状
5  # 转换成 3x4 的张量
6  print(t.view(3,4))
7  # 等同于
8  # print(t.reshape(3,4))
9  # 转换维度
10 print(t.permute(1,0))
11 # 把 2 行 1 列的张量，扩展成 2 行 3 列
12 t1 = torch.Tensor([[1], [2]])
13 print(t1.expand(2,3))
```

在上述范例中，先是通过第 3 行的代码，创建了 6 行 2 列的张量，在此基础上，用第 6 行所

示的 view 方法，把该张量转换成了 3 行 4 列的形式。

在实际操作中，也可以用第 8 行给出的 reshape 方法来转换张量的维度。这里请注意，用于转换张量维度的 view 和 reshape 方法，其参数都是待转换的张量维度值。

除此之外，还可以通过第 10 行给出的 permute 方法来转换张量的维度，这里 permute 方法的两个参数分别是 1 和 0，表示要互换第 0 维和第 1 维的数据，相当于转置。

本范例通过第 12 行和第 13 行代码，演示了张量扩展的相关做法。具体是，先通过第 12 行代码创建了 2 行 1 列的张量，在此基础上通过第 13 行的 expand 方法，把该张量扩展成 2 行 3 列的形式。

上述范例的运行结果如下，从中大家能实际感受到 view 等方法的运行结果。

```
1  torch.Size([6, 2])
2  tensor([[ 1,  2,  3,  4],
3          [ 5,  6,  7,  8],
4          [ 9, 10, 11, 12]])
5  tensor([[ 1,  3,  5,  7,  9, 11],
6          [ 2,  4,  6,  8, 10, 12]])
7  tensor([[1., 1., 1.],
8          [2., 2., 2.]])
```

## 2.2.4 过滤与条件操作

在拿到一个张量对象后，可以通过 clamp 方法限定其中数据的范围；可以通过 gather 方法获取指定其中指定索引的数据；可以通过 where 方法，对张量内指定条件的数据进行操作；也可以通过 take 方法，先把张量压缩成 1 维，然后返回指定索引位的数据。

在下面的 torchFilter.py 范例中，将演示上述方法的具体用法。

```
1  import torch
2  t = torch.tensor([[1, 2], [3, 4], [5, 6], [7, 8]])
3  #print(t)
4  # 限定张量内数据的范围
5  print(torch.clamp(t,2,5))
6  # 获取张量内指定索引条件的数据
7  print(torch.gather(t,dim=1,index=torch.tensor([[0],[1]])))
8  # 根据条件，对张量内的数据进行操作
9  print(torch.where(t>5,t*t, t))
10 # 压缩成一维，同时输出指定索引的数据
11 print(torch.take(t,torch.tensor([0, 2, 4])))
```

本范例的第 5 行代码演示了 clamp 方法的用法，该语句的输出结果如下，从中大家能看到，这里是通过 clamp 方法，把张量内的数据限制在 2 ～ 5 这个范围内。

```
1  tensor([[2, 2],
2          [3, 4],
3          [5, 5],
4          [5, 5]])
```

本范例的第 7 行演示了 gather 方法的用法，该语句的输出结果如下，这里是通过第 2 个参数

和第 3 个参数，指定了从第 2 个维度，即列的方向，截取第 1 行和第 2 行的数据。

```
tensor([[1],
        [4]])
```

本范例的第 9 行演示了 where 方法的用法，这里的含义是，针对该张量中大于 5 的数据，进行 t*t 的操作，同时不对 t<5 的数据进行操作。该方法的输出结果如下。

```
tensor([[ 1,  2],
        [ 3,  4],
        [ 5, 36],
        [49, 64]])
```

本范例的第 11 行演示了 take 方法的用法，该方法先把张量压缩成一维数组，并返回由第 2 个参数所指定的 0 号、2 号和 4 号索引位的数据，该方法的输出结果如下。

```
tensor([1, 3, 5])
```

# 2.3　搭建第一个神经网络

本节将在讲述神经网络相关知识点的基础上，给出搭建神经网络的基本步骤。通过学习本节内容，大家不仅能掌握训练集、验证集、测试集、损失函数和超参数等相关概念，而且还能通过代码，直观地掌握搭建和使用神经网络的相关技巧。

## 2.3.1　训练集、验证集和测试集

在训练神经网络过程中，一般来说，数据集的数量越多，那么训练的效果就越好。

为了更加合理地利用数据集，一般会把数据集分为训练集、验证集和测试集 3 部分，具体来说，会用训练集训练模型，用验证集评估模型，当完成训练后，再用测试数据给模型打分。

这 3 类集合的常见划分比例是，把总体数据的 60% 作为训练集，把 20% 的数据当成验证集，把剩下的 20% 数据当成测试集。这样的划分比例不仅可以确保模型能被充分训练，而且还能确保模型能有效地预测未知数据。

为了更有效地利用数据，在训练过程中一般还可以用到“交叉验证”的方法。比如，可以把样本数据划分成 a1,a2…a10 共 10 等份，在第一次训练中，把 a1 ～ a9 数据集用作训练集，把 a10 用作测试集，在第二次训练中，把 a9 用作测试集，把剩下的数据用作训练集，以此类推。

在这部分给出的范例中，为了重点讲述神经网络的相关概念，没有用到真实数据，训练和预测所用的数据都是模拟生成的，所以其中并没有包含训练集、验证集和测试集的相关代码。但是在真实的训练场景，这 3 类数据集会被广泛应用。

## 2.3.2　过拟合与欠拟合

过拟合与欠拟合的概念其实和上文提到的训练集和测试集有一定的关联。

过拟合是指在训练神经网络模型时，过多关注了训练集的数据特征，导致该模型能很好地预测训练集所用的数据，但不能很好地预测测试集数据。防止过拟合的措施有用更多的数据来训练，以及在训练时引入正则化约束项。

欠拟合是指，模型不能很好地找到训练集数据的规律，从而无法有效预测测试集里的数据，防止欠拟合的措施有增加模型的复杂度，或者减小训练所用的正则化系数。

### 2.3.3 损失函数

损失函数可以用来定量地分析神经网络等模型预测值和真实值之间的差距，换句话说，通过损失函数给出的差异值，大家能直观地看到模型质量的好坏。

损失函数在神经网络等数据预测场景里的位置如图 2.4 所示。从中大家能看到，损失函数会通过求均方差等方法，定量地计算出模型预测结果和真实结果之间的差异，这种差异结果可以直接作为训练和调参的依据。

在数据分析场景，基于均方差（MSE）的损失函数最为常见，该损失函数的具体算法如图 2.5 所示。

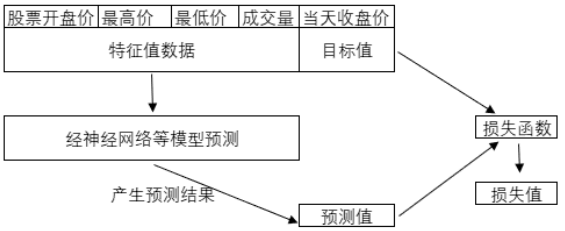


图 2.4 损失函数的位置

$$\sigma=\sqrt{\frac{\sum_{i=1}^{n}\left(x_i-\bar{x}\right)^2}{n}}$$

图 2.5 均方差算法示意图

预测值和真实值之间差的平方和的平均数叫作方差，方差的平方根就叫作均方差，预测值和真实值之间的差异越大，均方差的数值也就越大。当然，损失函数也可以用其他算法来量化预测值和真实值之间的差异。

### 2.3.4 神经网络的超参数

神经网络的超参数是指与网络结构、训练过程和优化算法相关的参数，这里讲解一些常用超参数的含义。

- 层数：该参数定义了神经网络的深度，层数越多的神经网络可以构成更为复杂的模式，但也会提升计算成本，并增加出现过拟合的风险。
- 批量大小：该参数定义了每次训练的样本数。如果设置较小的批量值，一般可以提高模型预测数据的精度，但可能导致训练过慢的情况；而如果设置较大的批量值，那么会提升训练速度，但会增大内存需求。
- 迭代次数：该参数用来定义训练集被学习的次数。一般来说，较少的迭代次数可能会导致数据未能被充分利用，而较多的迭代次数可能会导致模型出现过拟合的现象，即无法高效地预测分析未知数据。

- 学习率：该参数决定了模型在优化过程中更新权重值的步长。过高的学习率可能会导致训练过程不收敛，过低则会让训练过程过于缓慢。
- 优化器：该参数用来指定模型在学习训练过程中，用来更新参数的优化目标函数。其中 SGD 和 Adam 是当下最为常用的两种优化器。
- 正则化项：该参数用于减少模型的过拟合程度。在数据分析场景中，在损失函数里添加正则化项，可以避免出现较大的权重值，从而能优化训练的过程。

### 2.3.5　搭建神经网络的定式

学完神经网络的一些基础知识点后，下面将通过 CreateSimpleNet.py 范例，讲述搭建神经网络的基本步骤，以及用数据训练神经网络的实践要点，从中大家能看到，搭建和训练神经网络的代码其实是有定式的。

本神经网络的输入是二维平面坐标系 (−1,1) 区间内满足 y=cos(x) 关系的 100 个点，即这 100 个点是该神经网络的训练集。神经网络在接受训练后，将输出一条曲线来拟合这 100 个点的关系。

```
1  import torch
2  import matplotlib.pyplot as plt
3  import torch.nn.functional as F
4  # 模拟制作一个数据集
5  x = torch.unsqueeze(torch.linspace(-1,1,100), dim=1)
6  y = torch.acos(x) + 0.5*torch.rand(x.size())
```

本范例的前 3 行通过 import 语句引入了必需的依赖包，并通过第 5 行和第 6 行的代码，模拟生成了 100 个点，这些点满足 y=cos(x) 的关系，将用来训练神经网络。这里 x 是被分析的对象，也称特征值，y 是真实的结果，也称目标值。

```
7  class Net(torch.nn.Module):
8      # 定义神经元节点
9      def __init__(self, input_number, hidden_number, output_number):
10         super(Net, self).__init__()
11         self.hidden = torch.nn.Linear(input_number, hidden_number)
12         self.predict = torch.nn.Linear(hidden_number, output_number)
13     def forward(self, x): # 定义前向传播动作
14         x = self.hidden(x)
15         x = F.relu(x)  # 定义激活函数
16         x = self.predict(x)
17         return x
```

在本范例的第 7 ～ 17 行代码中，定义了用于创建神经网络的 Net 函数。

在定义神经网络时，首先通过 __init__ 方法里的第 9 行代码，指定调用封装在 torch 库底层的代码实现神经网络的初始化动作，通过第 11 行和第 12 行代码，用 torch.nn.Linear 方法定义了该神经网络中的两个线性层，分别为 hidden 和 predict，顾名思义，这两层内的神经元节点都是用线性函数来拟合数据。

torch.nn.Linear 有两个参数，分别表示所定义线性层的入参和出参的维度。其中名为 hidden 的线性层，其实相当于是神经网络的输入层，而 predict 线性层，则相当于输出预测结果的输出层。

随后，通过第 13 ～ 17 行的 forward 方法，指定数据前向传播的具体动作，具体为：输入的数据会被第一个线性层 hidden 处理，处理完成后的数据经 relu 激活函数处理后，再交给第二个线性层 predict 处理。

在神经网络中引入激活函数的目的是，让该模型能以非线性的方式分析预测数据，从而提升该模型处理数据的能力。

从中大家可以看到，用代码搭建神经网络时，可以在具体的类里定义 __init__ 和 forward 两个方法，在 __init__ 方法里定义各模块，在 forward 方法里定义前向传播的动作。前向传播的具体内容将在第 3 章进行介绍。

```
18 # 构建一个神经网络模型
19 model = Net(input_number=1, hidden_number=256, output_number=1)
```

根据上文给出的定义神经网络的 NET 类，这里是通过第 19 ～ 22 行代码创建一个神经网络模型，同时设置该模型所使用的优化器和损失函数等关键参数。

在第 19 行创建神经网络模型时，请大家注意其中的 3 个参数，其中，input_number 表示该神经网络接收的入参维度是 1 维，即特征值 x 是 1 维；output_number 表示该神经网络输出的目标值 y 也是 1 维，hidden_number 表示该神经网络中介于 hidden 和 predict 这两个线性层之间的隐藏层里，神经元节点的数量，这里设置成 256。该神经网络的结构如图 2.6 所示。

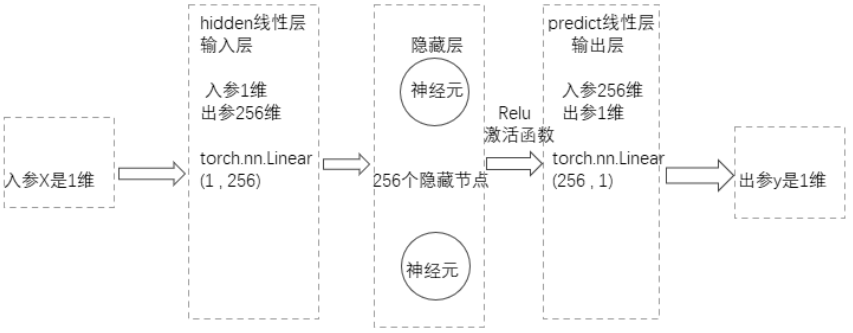


图 2.6　神经网络的结构

从中大家能看到，该神经网络的输入层接收的是 1 维数据，该层会用神经元节点把 1 维数据转换成 256 维的数据，这里数值 256 是与隐藏层里的神经元节点数量相匹配。

hidden 处理后的数据，交由隐藏层节点处理后，会把结果交激活函数处理，再传递给 predict 层，该层会把 256 维的数据转换成 1 维的输出数据，由此完成数据拟合的动作。

```
20 # 设置优化器，损失函数采用均方差函数
21 optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
22 loss_func = torch.nn.MSELoss()
```

随后，通过第 20 行代码设置了该神经网络所用的优化器，具体来说，指定了该网络在分析时，将使用梯度下降的分析方法，同时设置了学习率（也称步长）是 0.1。

在此基础上，通过第 22 行代码设置了该模型所用的损失函数，这里用到了基于均方差算法的损失函数。

```
23 # 开始训练
24 for t in range(200):
25     # 预测值
26     prediction = model(x)
27     loss = loss_func(prediction, y)
28     # 清空梯度参数
29     optimizer.zero_grad()
30     # 反向传播损失值
31     loss.backward()
32     # 更新神经网络参数
33     optimizer.step()
```

随后，通过第 24 ～ 33 行的 for 循环代码，开始训练神经网络，通过 for 循环的参数大家能看到，本次训练将进行 200 次。

大家可以把第 26 ～ 32 行的代码理解成神经网络训练的代码模板，从中大家可以看到，神经网络在训练时，一般会包含以下 5 个步骤：

（1）如第 26 行代码所示，用神经网络拟合特征数据，生成预测结果。

（2）如第 27 行代码所示，用损失函数计算真实结果和神经网络拟合结果之间的差异。

（3）如第 31 行代码所示，用类似 loss.backward 方法，从输出层向前传播损失函数的梯度，这个过程也称后向传播。

（4）如第 32 行代码所示，在后向传播的基础上，用损失函数的梯度更新神经网络的参数，这样做的目的是优化参数，提升下一次训练的准确性。

（5）本次训练结束后，用第 29 行代码清空梯度参数。

在由 for 循环指定的 200 次训练过程中，神经网络会以损失值为导向，不断优化内部参数，最终形成一个较为优化的数据分析模型。

```
34 # 可视化最后拟合出来的曲线
35 plt.figure()
36 plt.scatter(x.data.numpy(), y.data.numpy())
37 plt.plot(x.data.numpy(), prediction.data.numpy(), lw=3)
38 plt.text(0.5, 0, 'Loss is %.3f' % loss.data.numpy())
39 plt.show()
```

训练后，本范例再用第 35 ～ 39 行的代码展示了预测结果和真实数据之间的关系，具体来说，是通过第 36 行代码以散点图的方式展示了真实数据，通过第 37 行代码以曲线的方式展示了神经网络预测后的数据。

在此基础上，通过第 38 行代码显示了用于量化预测分析结果的损失值。该范例的运行结果如图 2.7 所示，从中大家能看到，基于神经网络分析的结果，能在一定程度上拟合真实数据之间的关系。而且，这里是用曲线而不是直线表示拟合结果，说明神经网络能用非线性的方式来分析并预测数据。

在第 3 章中，将详细讲述本范例用到的激活函数、损失函数和基于梯度下降的优化器等的关键实践要点，不过从本范例中，大家可以看到搭建神经网络及训练神经网络的一般定式。

（1）根据待拟合的数据集，定义神经网络的线性层和隐藏层的参数。

（2）定义神经网络的激活函数、损失函数和优化器，在定义优化器时，一般还需要定义步长等关键参数。

（3）用梯度下降等方式，训练该神经网络。训练的目的是，让该模型能根据特征值和目标值之间的关系，固化模型里的相关参数。

（4）当然，还可以用训练好的神经网络模型预测数据，不过本范例只输出了损失函数，在第3章中将以MNIST数据集为例，讲述预测数据的相关要点。

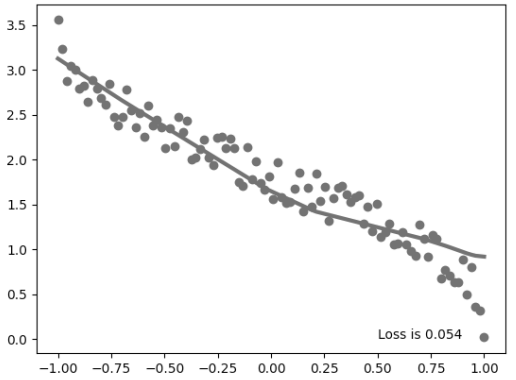


图 2.7　基于神经网络的预测效果图

## 2.4　小结和预告

本章首先讲述了张量的概念及针对张量的常见运算，请注意，张量是高维度的矩阵，也是神经网络接受参数的常见数据结构。随后，在讲述神经网络相关概念的基础上，通过代码演示了搭建和训练神经网络的相关要点，该代码具有可视化效果，从中大家能直观地看到相关参数的表现形式。

第3章将进一步用范例讲述神经网络中的重要概念，具体包含激活函数、损失函数和优化器，通过学习第3章，大家能进一步了解神经网络的工作机制。