第 5 章

常用数据集及其使用方式

数据是决定模型性能的关键因素之一，本章以图像数据集为主，对深度学习中常用的数据集的用法进行介绍。

在介绍数据集之前，需要先说明训练模型中的一些基本概念。在深度学习中，对于数据集一般会将其分为若干个 batch 依次送入模型进行训练，假设数据集全体样本数量为 N，batch_size 为 n，那么将所有数据可以分为 ceil(N/n)份，所以总共需要迭代 ceil(N/n)次才能将数据集中所有的数据训练一次，这通常被称为一个训练周期 epoch，而在每个 epoch 内使用一个 batch 的数据进行训练称为一次迭代 iteration。由以上描述容易知道 iteration 与 epoch 的关系如下：

$$\text{iteration} \times n = \text{epoch} \times N \tag{5-1}$$

每次迭代时选取 batch 的时候，通常有两种选取策略。第一种是顺序选取，当选取到最后一个 batch 时，下一次迭代又从数据集开头进行 batch 的样本选取。另一种是随机选取 batch 样本。前一种方法能够有效覆盖数据集中所有数据，后一种方法虽然有一定概率无法选取到数据集中所有数据，但是其大多数时候能提升模型的性能，由于是乱序的 batch，其能使模型避免刻意对数据样本之间的顺序关系进行记忆。

由于需要评价模型的性能，所以通常需要将全体样本数据分为训练集、验证集和测试集。为了简便起见，我们只把数据集分为训练集和测试集。此处我们不严格区分验证集与测试集，只需理解我们的出发点是将总体训练样本的一部分用于训练，另一部分用于评价模型性能。

对于有监督(有标签)任务而言，我们需要构建一种方法返回一个 batch 数据，包括训练样本及其对应的标签，即我们希望的代码如下：

```
def next_batch():
    ...
    return batch_x, batch_y
# 每一次调用 next_batch 方法时返回下一个 batch 数据
batch1 = next_batch()
batch2 = next_batch()
```

除此以外，对于数据集类 Dataset Class 来说，在创建该数据集的实例时，必要传入的参数还有数据存储路径 dataset_path、批处理大小 batch_size、取 batch 时是否乱序 shuffle、是否需要对数据做标准化与归一化处理 normalize(因为有的输入数据可能已经处理好了)及是否需要添加数据增强 augmentation 等，Dataset 类至少需要实现以下几种方法：__init__记录该实例必要的参数配置，next_batch 方法用于提取下一批数据，还有 num_examples 用于返回数据集中总体样本数，方便计算 iteration 数。因此理想中的 Dataset 类的构造函数应该包含以下结构，代码如下：

```
//ch5/data_utils/base_class.py
import abc


class Dataset(metaclass = abc.ABCMeta):
    @abc.abstractmethod
    def __init__(self,
                 dataset_path,
                 batch_size,
                 shuffle = True,
                 normalize = True,
                 augmentation = True):
        pass

    @abc.abstractmethod
    def next_batch(self, which_set):
        pass

    @abc.abstractmethod
    def num_examples(self, which_set):
        pass
```

__init__方法中传入了几个默认参数，分别将 shuffle、normalize 及 augmentation 置为 True，这么做主要是为了模型的性能提升，而 next_batch 与 num_examples 还有一个参数 which_set，用于表示对于训练集还是测试集进行操作。

5.1　IRIS 鸢尾花数据集

IRIS 数据集是常用的分类实验数据集，其是一类多变量分析的数据集。具体来说，数据集一共包含 150 个样本，总共分为 3 类，每类 50 个数据，每个数据包含 4 个属性，分别为花萼长度、花萼宽度、花瓣长度和花瓣宽度。模型的目标是通过这 4 个属性值来预测这个样本属于哪一种鸢尾花，三类鸢尾花分别 Iris Setosa(山鸢尾)、Iris Versicolour(杂色鸢尾)和 Iris Virginica(维吉尼亚鸢尾)。

通过阅读上面对数据集的叙述，不知读者能不能发现数据集的哪里需要我们来处理，下面我们就来分析需要我们处理的地方，首先最显而易见的就是模型不接收标量的类标号，我们需要将类标号 Iris Setosa、Iris Versicolour 和 Iris Virginica 转换成 one-hot 向量[1,0,0]、[0,1,0]和[0,0,1]。其次我们注意到 IRIS 是一个数据高度平衡的数据集，每一类样本的数量都相同，那么我们在划分训练集与测试集的时候应该注意这一点，划分后的数据集内部也应该基本保证类别之间的平衡。最后我们注意到每个样本有 4 个属性，需要注意的是我们应该对每个属性/特征进行单独的归一化和标准化。

IRIS 数据集总共由 3 个文件构成，文件名分别为 Index、iris. data 和 iris. names(读者可能还会看到一个名为 bezdekIris. data 的文件，此处我们不使用该文件)。其中，Index 是说明数据集中所有的文件，iris. data 是 150 个样本的具体值，iris. names 是 IRIS 数据集的描述性文字。因此 3 个文件中，我们实际上只用关注 iris. data 即可。

iris. data 本质是一个 csv 文件，文件内共有 150 行，表示 150 个训练样本，每一行的样本中包含 5 个分量，前 4 个为特征值，最后一个字符串表示其类别。因此我们可以使用 pandas 模块读取该文件并分别获得特征值与类别，代码如下：

```
//ch5/data_utils/iris.py
def read(file):
    return pd.read_csv(file, header = None, low_memory = False).values

data = list()
labels = list()

for dp in data_path:
    _data = read(dp)
    data.extend([_d[ : 4] for _d in _data])
    labels.extend([_d[ - 1] for _d in _data])
```

将所有样本分为训练集与测试集，我们默认随机选取 20%的数据作为训练数据，使用 random 模块的 sample 方法随机指定测试集样本的下标，并通过该下标将所有数据分割为训练集与测试集，代码如下：

```
//ch5/data_utils/iris.py
#计算出测试集应包含多少个样本
split_train_and_test = 0.2
num_test = int(split_train_and_test * len(__data))
#使用 random 随机选取测试集样本的下标
test_ids = random.sample(list(range(len(__data))), k = num_test)

#将全部样本通过下标分割为训练集与测试集
self.__train_data = [__data[idx]
                     for idx in range(len(__data)) if idx not in test_ids]
self.__train_labels = [__labels[idx]
```

```
                    for idx in range(len(__data)) if idx not in test_ids]

self.__test_data = [__data[idx]
                    for idx in range(len(__data)) if idx in test_ids]
self.__test_labels = [__labels[idx]
                    for idx in range(len(__data)) if idx in test_ids]
```

获得数据后，我们需要进行一些预处理，例如将字符串的类标先转换为标量再转换为one-hot向量，代码如下：

```
//ch5/data_utils/iris.py
#将训练和测试的标签由字符串转换为标量值
self.__train_labels = [self.flower_name_id_dic[n]
                       for n in self.__train_labels]
self.__test_labels = [self.flower_name_id_dic[n]
                       for n in self.__test_labels]

#将数据和标签转换为Numpy.array类型
self.__train_data = np.stack(self.__train_data, axis=0)
self.__test_data = np.stack(self.__test_data, axis=0)

#将标签转换为one-hot向量
self.__train_labels = np.eye(self.num_classes)[self.__train_labels]
self.__test_labels = np.eye(self.num_classes)[self.__test_labels]
```

还需要计算每个特征的最值(为归一化)和均值方差(为标准化)，在此直接给出类别与标量类别号及IRIS数据集的最值与均值方差，代码如下：

```
//ch5/data_utils/iris.py
#类别名称与类标之间的映射关系
flower_name_id_dic = {
    'Iris-setosa': 0,
    'Iris-versicolor': 1,
    'Iris-virginica': 2
}

#每个特征的最大值与最小值
max_val = [7.9, 4.4, 6.9, 2.5]
min_val = [4.3, 2.0, 1.0, 0.1]

#每个特征的均值与标准差
mean = [0.42870370, 0.43916666, 0.46757062, 0.45777777]
std = [0.22925036, 0.18006108, 0.29805579, 0.31692192]
```

有了数据集上的统计量后，归一化和正则化过程也十分简单。先将数据归一化到0～

1，再使用 Z-score 将其标准化，代码如下：

```
//ch5/data_utils/iris.py
def __normalization(self, data):
    data = (data - self.min_val) / (self.max_val - self.min_val)
    data = (data - self.mean) / self.std

    return data
```

特征处理完毕后，还需要将类别名称先转换为类别号，再转换为 one-hot 向量，代码如下：

```
//ch5/data_utils/iris.py
#将训练和测试的标签由字符串转换为标量值
self.__train_labels = [self.flower_name_id_dic[n]
                       for n in self.__train_labels]
self.__test_labels = [self.flower_name_id_dic[n]
                      for n in self.__test_labels]

#将数据和标签转换为 Numpy.array 类型
self.__train_data = np.stack(self.__train_data, axis=0)
self.__test_data = np.stack(self.__test_data, axis=0)

#将标签转换为 one-hot 向量
self.__train_labels = np.eye(self.num_classes)[self.__train_labels]
self.__test_labels = np.eye(self.num_classes)[self.__test_labels]
```

由于获得了训练集与测试集数据，num_example 方法也十分简洁，代码如下：

```
//ch5/data_utils/iris.py
def num_examples(self, which_set):
    if 'train' in which_set:
        return len(self.__train_data)
    elif 'test' in which_set:
        return len(self.__test_data)
```

整个数据集类的核心是 next_batch 方法，其实现相对复杂。首先需要为方法传入 which_set 参数，表示从训练集还是测试集中取数据。其次读取数据分为两种方式，一种是顺序读取，另一种是随机读取。随机读取实际上十分简单，使用 np.random.choice 在目标数据样本中随机选取 batch_size 个数据样本即可。顺序读取相对复杂，先设定一个指针 pointer，其表示选取[pointer-batch_size, pointer]中的数据，因此 pointer 的初值为 batch_size。当 pointer 超过 len(data)即所有数据时，可以使用取模运算将其重置到合法位置。可以看出 pointer 应该是一个全局变量，因此将其设置为成员变量，并且为了能在 next_batch 中改变它的值，可以直接传入引用类型而非基本数字类型，将 pointer 写成一个列表即可，即

[batch_size]。next_batch 函数的具体实现代码如下：

```
//ch5/data_utils/iris.py
def next_batch(self, which_set):
    #判断对哪一个集合进行操作
    #分别取出对应的数据、标签及当前数据位置指针
    if 'train' in which_set:
        target_data = self.__train_data
        target_label = self.__train_labels
        target_pointer = self.__train_pointer
    elif 'test' in which_set:
        target_data = self.__test_data
        target_label = self.__test_labels
        target_pointer = self.__test_pointer

    #如果需要将 batch 内数据乱序(shuffle = True),直接随机选取样本即可
    if self.shuffle:
        indices = np.random.choice(
            self.num_examples(which_set), self.batch_size)
    else:
        #否则使用指针顺序取出数据与标签,注意指针指到最后时需要将其重新指向数据开头
        indices = list(
            range(target_pointer[0] - self.batch_size, target_pointer[0])
        )

        target_pointer[0] = (target_pointer[0] + self.batch_size)
                    % self.num_examples(which_set)

    #取出 batch 数据后,使用深复制得到一个副本以方便操作,防止篡改原始数据
    batch_data = deepcopy(target_data[indices])

    #对 batch 里的数据做标准化
    if self.normalize:
        batch_data = self.__normalization(batch_data)

    return batch_data, target_label[indices]
```

IRIS 类完整代码具体可见随书代码//ch5/data_utils/iris.py。对于所有的数据集类来说，使用的流程大致相同，首先将数据从外存读取进来，再对样本与标签做必要的转换和处理，最后在 next_batch 中写取 batch 逻辑即可。容易理解，所有的数据集类中的 normalize 方法与 next_batch 方法逻辑大致相同，使用不同的数据集时只需改变其具体数据。因此，若这两种方法相较于 IRIS 类没有较大改动，之后不再单独列出这两种方法的代码。

5.2 MNIST 手写数字数据集

MNIST 是一个 0～9 的手写数字图像数据集，由 250 个不同的人手写数字构成，共包含 60000 个训练数据与 10000 个测试数据。数据分布均衡，训练集中每个数字包含 6000 张图像，测试集中每个数字包含 1000 张图像。每张图像大小为 28×28 像素，不过由于其是黑白图像，只有一个通道，因此每张图像总共可以使用 28×28×1＝784 个像素进行表示。为了简单起见，每张图像都被平展为 784 个数的一维结构。因此，容易知道训练集的形状为(60000，784)，同理测试集的形状为(10000，784)，标签以标量进行存储，标签与该样本对应的数字相同，即数字 0 的标签也为 0 等。

MNIST 数据集一共包含 4 个文件：train-images-idx3-uByte. gz、t10k-images-idx3-uByte. gz、train-labels-idx1-uByte. gz 和 t10k-labels-idx1-uByte. gz，分别代表训练集图像、测试集图像、训练集标签和测试集标签。下载完数据后，需要先将这 4 个压缩文件进行解压，得到 train-images.idx3-uByte、t10k-images.idx3-uByte、train-labels.idx1-uByte 和 t10k-labels. idx1-uByte。在此我们不深究每个文件的内部存储结构(有兴趣的读者可以移步 MNIST 官网了解数据结构)，此处只提供读取文件的方法，代码如下：

```
//ch5/data_utils/mnist.py
def __read(self, buffer, to_skip, each_size):
    objs = list()
    idx = struct.calcsize(to_skip)

    try:
        while True:
            o = struct.unpack_from(each_size, buffer, idx)
            objs.append(o)
            idx += struct.calcsize(each_size)
    except struct.error:
        return objs

def __read_zip_file(self, file_path):
    with open(file_path, 'rb') as f:
        buffer = f.read()
    return buffer

#训练集与测试集的文件名标识
train_identifier = 'train'
test_identifier = 't10k'

for imp in image_path:
    #读取训练样本,每次读取 784 个数
    if self.train_identifier in imp:
```

```
            self.__train_images.extend(self.__read(
                self.__read_zip_file(imp), '>IIII', '>784B'))
        #读取测试样本,每次读取784个数
        if self.test_identifier in imp:
            self.__test_images.extend(self.__read(
                self.__read_zip_file(imp), '>IIII', '>784B'))

    for lp in label_path:
        #读取训练标签,每次读取1个数
        if self.train_identifier in lp:
            self.__train_labels.extend(self.__read(
                self.__read_zip_file(lp), '>II', '>1B'))
        #读取测试标签,每次读取1个数
        if self.test_identifier in lp:
            self.__test_labels.extend(self.__read(
                self.__read_zip_file(lp), '>II', '>1B'))
```

由于最终的研究对象是二维图像,所以我们希望next_batch返回的数据是一个28×28的二维图像而非一维的784个像素,next_batch函数的代码如下:

```
//ch5/data_utils/mnist.py
def next_batch(self, which_set, reshape=True):
    #读取训练或测试数据
    if 'train' in which_set:
        …
    elif 'test' in which_set:
        …
    #以随机或顺序的方式读取数据
    if self.shuffle:
        …
    else:
        …
    batch_data = deepcopy(target_image[indices])

    #对输入数据进行标准化
    if self.normalize:
        batch_data = self.__normalization(batch_data)

    #将数据重整为二维图像
    if reshape:
        batch_data = np.reshape(batch_data, [-1, 28, 28, 1])

    return batch_data, target_label[indices]
```

读取数据后，可以使用 Matplotlib 将 28×28 的图像显示出来，代码如下：

```
//ch5/data_utils/mnist.py
#取出一个 batch 数据
ims, labs = mnist.next_batch('train')

#将数据重整成 Matplotlib 可以显示的形状
ims = np.squeeze(ims)

#一共随机取出 8×8 张图像
row = col = 8

random_ids = random.sample(list(range(batch_size)), k=row * col)
selected_ims = ims[random_ids]

fig, axes = plt.subplots(row, col)
for i in range(row):
    for j in range(col):
        #取出每个 axes 对图像进行显示
        axes[i][j].imshow(selected_ims[i * row + j], cmap='gray')
#总图标题
plt.suptitle('MNIST samples')
plt.show()
```

运行以上程序，可以看到从 MNIST 中随机选取的 64 张图像，如图 5-1 所示。

从图像中容易看出，MNIST 数据集中图像的特点，其是黑底白字的图像，并且其基本是二值图像，非黑即白，基本不存在介于两者之间的像素。可以看出不同人写的数字差异还是较大的，例如数字的粗细及数字 4 和 7 的不同写法。

由于图像近似为二值图像，实际上不需要对所有数据求取最大值与最小值，其最大值直接按照 255，最小值按照 0 处理即可，同时计算出归一化后的数据均值与方差，并进行归一化。下面的代码直接给出归一化后的 MNIST 数据集上的均值与标准差：

```
//ch5/data_utils/mnist.py
#归一化后的 MNIST 上的均值与标准差
mean = 0.13092535192648502
std = 0.3084485240270358

def __normalization(self, imgs, epslion=1e-5):
    imgs = imgs / 255.0
    imgs = (imgs - self.mean) / self.std

    return imgs
```

完整代码可见随书代码//ch5/data_utils/mnist.py。

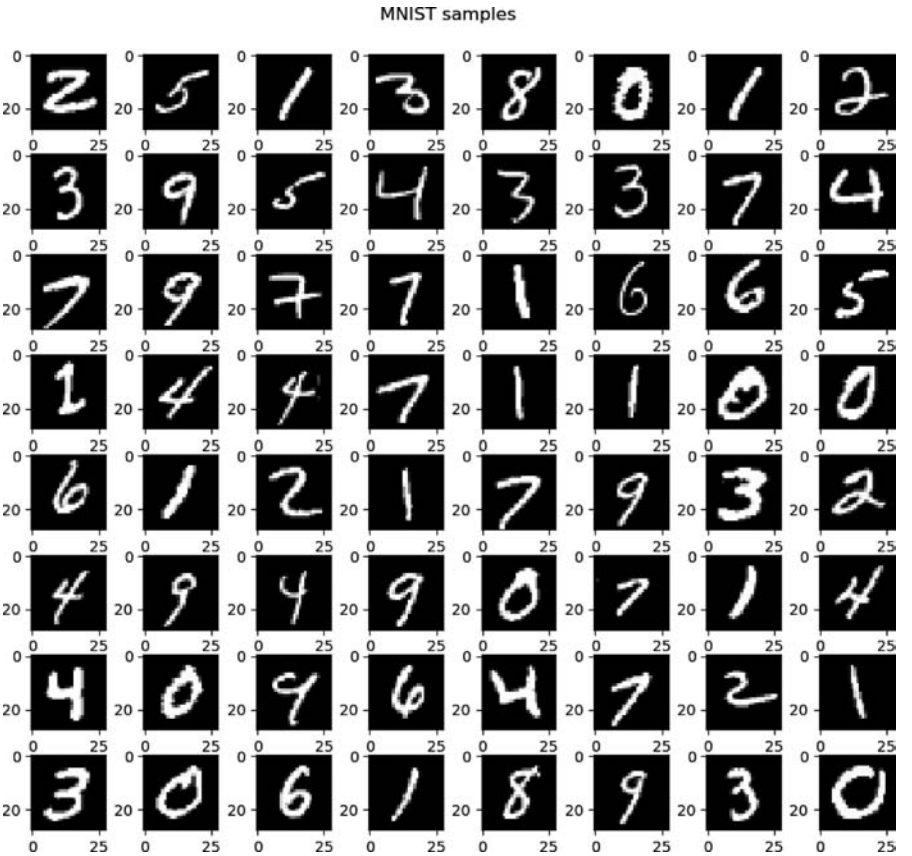

图 5-1　MNIST 中的部分图像

5.3　SVHN 数据集

与 MNIST 数据集类似，SVHN 也是一个关于数字的数据集，不同的是 SVHN 对数字 0 的标签为 10，因此使用之前需要将数字 0 的标签修改为 0。SVHN 的数字取自于街景中的门牌号，为彩色图像。

SVHN 数据集分为两部分，第一部分通常用于目标检测，即用框将需要检测的数字框出。第二部分数据集用于分类，可以认为是将检测框中的数字框出来作为单独的图像，每张图像的尺寸为 32×32×3。两者的区别如图 5-2 所示。

训练集中有 73257 个数字训练样本，测试集中有 26032 个数字训练样本。不同的是，SVHN 还有额外数据，其中包含 531131 个数字训练样本，这些样本较训练集中的数字更简单并容易识别，通常作为补充数据一起用于训练。分类数据集以 Matlab 格式进行存储，使用时直接使用 SciPy 读取即可，具体使用方法可以参考第 2 章相关内容。

(a) 用于目标检测的数据　　(b) 用于图像分类的数据

图 5-2　SVHN 中的部分图像

由于 SVHN 与 MNIST 数据集具有一定的相似性，本节只对 SVHN 数据集概况做一个基本介绍，本书仅使用 MNIST 作为数字相关数据集进行讲解。

5.4　CIFAR-10 与 CIFAR-100 数据集

CIFAR 是分类任务中常用的数据集，其为一个自然场景的彩色数据集，里面的图像大小为 32×32，因此每张图像的形状为(32，32，3)。CIFAR 数据集分为 3 个存储版本：Python 版、Matlab 版和 binary 版(为了简便，本书当然使用 Python 版本)，Python 版的数据使用 pickle 存储，官方已给出读取数据集的方法，代码如下：

```
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding = 'Bytes')
    return dict
```

读取的数据以字典形式返回，需要以传入 key 的形式从字典中获取图像数据与标签。

需要特别注意的一点是，与 MNIST 类似，CIFAR 数据集中的图像也是以一维数据的形式存储的，即每张图像以 32×32×3=3072 个像素值存储，其中第 1～1024 个像素值是 R 通道上的值，第 1025～2048 是 G 通道上的值，第 2049～3072 是 B 通道上的值。换言之，可以认为每张图像的组织方式如下所示，下标 i、j 表示像素值所在的行和列：

$$[R_{0,0},R_{0,1},\cdots,R_{0,31},\cdots,R_{31,31},G_{0,0},G_{0,1},\cdots,G_{0,31},\cdots,$$
$$G_{31,31},B_{0,0},B_{0,1},\cdots,B_{0,31},\cdots,B_{31,31}]$$

由于 MNIST 数据只有一个通道，因此对其可以直接使用 reshape 方法转换为二维图像，但是对于 CIFAR 的数据组织方式，若直接使用 np.reshape(im，[32，32，3])进行转换，

得到的结果如下：

$$[[[R_{0,0},R_{0,1},R_{0,2}],[R_{0,3},R_{0,4},R_{0,5}],\cdots,[R_{2,29},R_{2,30},R_{2,31}]],\cdots,$$
$$[\cdots,[B_{31,29},B_{31,30},B_{31,31}]]]$$

这显然不符合图像像素组织形式的要求，我们希望得到的图像像素组织形式为$[R_{i,j},G_{i,j},B_{i,j}]$，因此在 reshape 时，第一个维度应该将 3072 个像素值分为 3 个部分 RGB，再对每个通道内的像素组织成二维形式 32×32，所以需要使用下面的代码进行形状重整，代码如下：

```
new_im = np.reshape(im, [3, 32, 32])
```

此时得到的新图像格式为 CHW，即通道在前，空间维度在后，此时只需再使用转置方法将通道维度转置到空间维度后即可得到格式为 HWC 的图像，代码如下：

```
new_im = np.transpose(new_im, [1, 2, 0])
```

若附加上 batch_size 维度，整个转换过程的代码如下，其中维度为－1 值表示代码自动计算该维度的值，代码如下：

```
new_ims = np.transpose(np.reshape(ims, [-1, 3, 32, 32]), [0, 2, 3, 1])
```

CIFAR 有两个数据集版本，分别为 CIFAR-10 与 CIFAR-100，下面分别介绍这两个数据集版本。

5.4.1　CIFAR-10

CIFAR-10 数据集含有 60000 张彩色图像，总共包含 10 类图像，分别为 airplane(飞机)、automobile(汽车)、bird(鸟)、cat(猫)、deer(鹿)、dog(狗)、frog(青蛙)、horse(马)、ship(船)和 truck(卡车)。CIFAR-10 数据集高度平衡，每一类含有 6000 张图像。CIFAR-10 中部分图像如图 5-3 所示。

CIFAR-10 数据集总共含有 8 个文件，其中描述性文件有 readme.html 和 batches.meta，readme.html 是一个将请求重定向至 CIFAR 数据集官网的网页，batches.meta 中含有 CIFAR-10 数据集的元数据，其中包含 10 个类名(airplane 等)，表示数字标签到类名之间的映射关系，剩下的 6 个数据文件中，data_batch_1～data_batch_5 表示训练集数据，每个文件中包含 10000 张图像与 10000 个对应的数字标签，test_batch 表示测试集数据，其中同样包含 10000 张图像与其对应的标签。

从 CIFAR-10 的文件中读取数据时，使用键 data 取图像样本，每个文件中的样本形状为(10000, 3072)。使用 label 取图像标签，其形状为(10000,)，说明标签以数字标签进行标识，后期需要将其转换为 one-hot 形式。我们需要把 6 个数据集文件一次性读入，最终得到训练集和测试集样本的形状为(50000, 3072)和(10000, 3072)，代码如下：

图 5-3　CIFAR-10 中的部分图像

```
//ch5/data_utils/cifar.py
#定义训练集与测试集文件名标识符
self.train_identifier = 'data_batch'
self.test_identifier = 'test_batch'

#CIFAR-10 中标签的 key
label_name = b'labels'

for dp in data_path:
    data = self.unpickle(dp)
    if self.train_identifier in dp:
        self.__train_images.append(data[b'data'])
        self.__train_labels.append(data[label_name])
    if self.test_identifier in dp:
        self.__test_images.append(data[b'data'])
        self.__test_labels.append(data[label_name])
```

由于 CIFAR-10 是自然图像的数据集，像素值范围较为广泛，因此我们直接将 CIFAR-10 的图像除以 255 进行归一化。归一化后的 CIFAR-10 数据集的均值与标准差分别为

```
mean = [0.49186878, 0.48265391, 0.44717728]
std = [0.24697121, 0.24338894, 0.26159259]
```

5.4.2　CIFAR-100

CIFAR-100 数据集与 CIFAR-10 数据集十分类似，不同的是 CIFAR-100 一共有 100

类，并且其类别以层级结构进行组织，例如小类中的 bottles 和 bowls 就共同属于大类/超类 food container，其 100 个小类一共属于 20 个超类，其映射关系如表 5-1 所示。

表 5-1　CIFAR-100 上超类与子类之间的关系

类	子　类
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

CIFAR-100 的存储结构与 CIFAR-10 的存储结构类似，一共由 4 个文件构成：file.txt 为空文件，meta 为 CIFAR-100 的元数据，包含对数据集中类别的说明信息等；train 为训练数据，包含 50000 个训练样本与其对应的标签（包括子类标签与超类标签）；test 为测试数据，包含 10000 个训练样本与标签。

由于 CIFAR-100 中含有两种标签，因此我们在使用 CIFAR-100 数据集时，需要指定使用 coarse label（粗糙的标签，即超类）还是 fine label（精细的标签，即子类），同时由于 CIFAR-10 与 CIFAR-100 的数据组织方式相同，我们希望通过为构造函数传入相应的参数的方式指定使用 CIFAR-10 数据集还是 CIFAR-100 数据集，以及使用 CIFAR-100 数据集时究竟使用哪一种标签形式，定义 CIFAR 数据集类的构造函数头的代码如下：

```
//ch5/data_utils/cifar.py
class Cifar(Dataset):
    def __init__(self,
                 data_path,
```

```
                    batch_size,
                    shuffle = True,
                    normalize = True,
                    c10 = True,
                    coarse_label = False,
                    augmentation = True):
    …
```

读取 CIFAR-100 上的样本标签时较为复杂，需要根据 coarse_label 的值来决定从 pickle 读取的字典中取键为 coarse_label 或 fine_label 的标签，从 CIFAR-100 文件中读取数据的代码如下：

```
//ch5/data_utils/cifar.py
#定义训练集与测试集文件名标识符
self.train_identifier = 'train'
self.test_identifier = 'test'

#根据传入的参数判断选用超类或子类作为标签
if coarse_label:
    label_name = b'coarse_labels'
    self.num_classes = 20
else:
    label_name = b'fine_labels'
    self.num_classes = 100

for dp in data_path:
    data = self.unpickle(dp)
    if self.train_identifier in dp:
        self.__train_images.append(data[b'data'])
        self.__train_labels.append(data[label_name])
    if self.test_identifier in dp:
        self.__test_images.append(data[b'data'])
        self.__test_labels.append(data[label_name])
```

CIFAR-100 数据的标准化方法与 CIFAR-10 类似，在此给出归一化后的 CIFAR-100 数据集上的均值与标准差：

```
mean = [0.50736203, 0.48668956, 0.44108857]
std = [0.26748815, 0.2565931, 0.27630851]
```

CIFAR 相关的完整代码可以参考随书代码//ch5/data_utils/cifar.py。

5.4.3 对图像进行数据增强

在 4.6 节介绍深度学习中的技巧时曾提到可以在输入数据上进行数据增强操作以扩大

训练样本，从而提升模型的性能。相较于单通道的 MNIST 数据集，CIFAR 数据集难度大很多，因此常常需要对 CIFAR 数据集使用数据增强技术提升模型在 CIFAR 数据集上的性能。数据增强通常分为两种形式，一种是离线的数据增强，即先使用各种图像处理技术得到新的样本并将其保存下来，从而得到一个新的大数据集。另一种是在线的数据增强，即在每次取 batch 数据时对其进行数据增强，并把增强后的 batch 返回模型输入。

数据增强的核心思想是如何设计一种策略，使图像受到干扰前后数据标签不变，此时我们便认为增强后的图像可以作为新样本对模型进行训练。值得注意的是，数据增强需要引入随机性，通常会指定一个概率值表明执行这项数据增强的可能性，这样才能保证每次对于同样的样本能生成不一样的新样本。

本节介绍的数据增强技术可以应用于离线生成也可以在线应用，本节主要介绍如何将数据增强应用到 batch 数据上，即在线增强。

1. 图像翻转

图像翻转包括水平翻转与竖直翻转。通常对于自然图像来说，使用两者都不会改变其标签，而对于数字图像来说，翻转通常是不适用的，因为数字在方向上具有识别性。例如将 3 水平翻转后得到 ε，显然改变了图像的标签。将数字 6 竖直翻转得到 9 也是同样的道理。

由于我们的数据集类最终将数据转换为 np.array 类型，所以我们直接使用 OpenCV 对图像进行处理即可。在 OpenCV 中可以使用其 flip 方法对图像进行翻转，根据传入的 code 不同分别对图像进行不同的翻转操作。详细操作可以参考 2.4.2 节第 2 部分的讲解，对单张图像进行翻转的代码如下：

```
def flip_one(image, axis):
    image = cv2.flip(image, axis)
    return image
```

当对整个 batch 的数据进行翻转时，需要以一定概率对每张图像进行翻转。下面的程序说明了如何以一定的概率对 batch 中的每张图像进行水平翻转，代码如下：

```
//ch5/data_utils/augmentation.py
def horizontal_flip(batch_data, prob=0.5):
    #获取 batch 的形状
    N, H, W, C = batch_data.shape

    #为翻转后的数据创建一个形状与输入相同的占位符
    flipped_batch = np.zeros_like(batch_data)

    #对 batch 中的每张图像分别进行操作
    for i in range(N):
        #随机生成执行概率
        flip_prob = np.random.rand()
```

```
            if flip_prob < prob:
                #对 batch 中的第 i 张图像进行水平翻转操作
                flipped_batch[i] = flip_one(batch_data[i], axis = 1)

        #返回水平翻转后的 batch
        return flipped_batch
```

对整个 batch 以概率竖直翻转的代码与水平翻转类似，仅需改变翻转的 code 为 0 即可，代码如下：

```
//ch5/data_utils/augmentation.py
def vertical_flip(batch_data, prob = 0.5):
    #获取 batch 的形状
    N, H, W, C = batch_data.shape

    #为翻转后的数据创建一个形状与输入相同的占位符
    flipped_batch = np.zeros_like(batch_data)

    #对 batch 中的每张图像分别进行操作
    for i in range(N):
        #随机生成执行概率
        flip_prob = np.random.rand()

        if flip_prob < prob:
            #对 batch 中的第 i 张图像进行水平翻转操作
            flipped_batch[i] = flip_one(batch_data[i], axis = 0)

    #返回水平翻转后的 batch
    return flipped_batch
```

2. 图像裁剪

我们通常认为，从原图像中裁剪出一部分图像后，新图像的标签也是不变的。例如对于一张猫的图像，现在从其中随机裁剪出原图像 80%～90%大小的子图，那么我们认为新图像仍然能保留猫的可辨别特征，但是从数据分布上来说，其本质完全是另一张完全不同的图像。

由于神经网络通常要求输入的图像大小保持一致，因此在进行图像裁剪时，我们常用的方法有两种，一种是将原图像放大为大小为(αH，αW)的图像(α 为一个大于 1 的缩放因子)，再从新图像中随机裁剪出一个大小为(H，W)的子图。第二种方法是在原图四周用 0(或其他任意值)填充一周宽度为 d 的像素，得到大小为($H+2d$，$W+2d$)新图像，再从新图像中随机裁剪出一个大小为(H，W)的图像。两种方法没有优劣之分。两种方法的实现方式也像各自所描述的一样，进行放大或填充后再进行随机裁剪。

图像裁剪通常与水平翻转一起使用，作为最常使用的数据增强方法，其被称为 random_

crop_and_flip，其实现方式可以参考以下程序，代码如下：

```
//ch5/data_utils/augmentation.py
def random_crop_and_flip(batch_data, padding_size, resize=False):
    #获取 batch 的形状
    N, H, W, C = batch_data.shape

    #为翻转后的数据创建一个形状与输入相同的占位符
    new_batch = np.zeros_like(batch_data)

    #根据每个方向上填充的宽度计算新图像的大小
    new_H = H + 2 * padding_size
    new_W = W + 2 * padding_size

    for i in range(N):
        #生成随机裁剪的左上角坐标
        y_offset = np.random.randint(low=0, high=2 * padding_size)
        x_offset = np.random.randint(low=0, high=2 * padding_size)

        #使用缩放方式进行裁剪或使用填充方式进行裁剪
        if resize:
            image = resize_one(batch_data[i], (new_H, new_W))
        else:
            image = np.pad(batch_data[i],
                           (
                               (padding_size, padding_size),
                               (padding_size, padding_size),
                               (0, 0)
                            ),
                           mode='constant')

        #完成图像的裁剪
        new_batch[i] = image[y_offset: y_offset + H, x_offset: x_offset + W, :]

    #完成对 batch 的随机水平翻转
    new_batch = horizontal_flip(new_batch, prob=0.5)

    return new_batch
```

除了像翻转与裁剪这一类图像空间层面上的增强，还有许多图像通道/色彩上的增强方式，例如对图像进行自动对比度、减少饱和度、色彩抖动(color jitter)等。理论上来说，大多数计算机图形学操作的算法都可以作为一种数据增强的手段，没有绝对的好与坏的数据增强方法，需要根据数据集自身的特性选择最适合的数据增强方法。

在线增强需要将数据增强方法应用到每个 batch 数据上，因此需要在 next_batch 函数中进行处理，代码如下：

```
//ch5/data_utils/cifar.py
def next_batch(self, which_set):
    ...
    batch_data = deepcopy(target_image[indices])

    #在复制的 batch 副本上做随机裁剪与水平翻转操作
    if do_augment:
        batch_data = random_crop_and_flip(batch_data, padding_size=4)

    if self.normalize:
        batch_data = self.__normalization(batch_data)

    return batch_data, target_label[indices]
```

本节介绍的数据增强方法可以应用到任意图像数据集中。

5.5 Oxford Flower 数据集

不难看出，从 IRIS 到 CIFAR 数据集，我们采取的数据读取方式都是一次性将所有的数据放入内存中，再从内存中每次随机选取 batch 放入模型中。这样做的好处是减少对 IO 的操作次数，从而加快了数据读取操作的速度。但是在实际操作中，数据集的大小往往大于内存，无法一次全部将数据放入内存中。此时常采用的做法是将所有数据的路径保存起来（相当于保存字符串），在选取 batch 时，从这些路径中以顺序或随机的方式进行选取，再对选出的路径进行图像的读取，经过必要的处理后（如数据增强、标准化等）将其返回即可。虽然这样做会增加由 IO 带来的时间消耗，但是在硬件条件不足的情况下不失为一个好的选择。

Oxford Flower 是一个花卉的图像数据集，其分为大小数据集两个版本。小的数据集一共含有 17 类花卉，称为 17flowers，其可以用来作为图像分割的数据集。大的数据集一共含有 102 类花卉，称为 102flowers，可以用来作为图像分类与图像分割的数据集。在此我们只使用 102flowers 数据集。

与前面的数据集不同，102flowers 一共有 8189 张图像，更接近于从真实世界中采集到的数据集，首先它的图像大小各异，其次每一类图像的数量不均衡（最少的类别仅有 40 张图像，最多的类别含有 258 张图像），最后也是最重要的一点是，它以纯图像的形式存储，不像前面介绍的数据集有一个较好的封装过程，这也更接近于我们直接从真实世界中得到的数据。

102flowers 一共含有 3 个文件，分别是包含所有图像的压缩包 102flowers.tgz，将其解压能得到 jpg 文件夹，含有所有的 8189 张图像，图像名从 image_00001.jpg 到 image_08189.jpg。imagelabels.mat 里面有每张图像对应的标签，一共含有 8189 个数，分别表示 image_00001.jpg 到 image_08189.jpg 图像的标签，不过需要注意的是，最小的类标为 1，最大的类标为 102，为了处理成 one-hot 标签，我们需要将标签统一减 1，得到新的标签范围为

0～101。setid.mat是数据集的划分，trnid表示训练集数据的编号、valid表示验证集的编号、tstid表示测试集的编号。读取mat文件可以使用scipy.io中的readmat方法，相关用法可以参考第2章。

可以发现图像文件名中的数字都以5位数表示，若位数不够则使用前导0进行填充，因此当根据setid.mat中的数字取相应的文件名时，我们需要为该数补全前导0才能符合文件名的要求。

虽然102flowers数据大小总共也不过300MB左右，以现在的硬件水平来说，将所有的数据放入内存完全绰绰有余，在此我们还是介绍通用的数据集处理方法，即假定内存不足以放入所有的图像情况下的做法。

首先读取训练集、验证集和测试集的编号分割文件，得到每个部分的编号情况，代码如下：

```
//ch5/data_utils/oxford_flower.py
#数据集分割编号文件
image_split = 'setid.mat'

#__read方法返回scipy.io.readmat的结果
readObjs = self.__read(image_split)

#得到每个数据部分的文件id
self.__train_images = np.squeeze(readObjs['trnid'])
self.__val_images = np.squeeze(readObjs['valid'])
self.__test_images = np.squeeze(readObjs['tstid'])
```

类似地，读取数据标签的代码如下：

```
//ch5/data_utils/oxford_flower.py
#数据集标签文件
label = 'imagelabels.mat'

#通过减1将1～102的标签转换为0～101
self.__all_labels = np.squeeze(self.__read(label)['labels']) - 1

#将数字标签转换为one-hot向量
self.__all_labels = np.eye(self.num_classes)[self.__all_labels]
```

当获得图像编号后，可以通过为编号加上前导0并为其加上image_前缀即可找到编号对应的文件名，代码如下：

```
//ch5/data_utils/oxford_flower.py
def __read_images(self, img_id):
    #将传入的image_id转换为字符串，方便进行拼接
```

```
    img_id = str(img_id)

    #拼接出文件的完整路径
    img_path = os.path.join(self.image_root,
            'image_{}.jpg'.format('0' * (5 - len(img_id)) + img_id))

    #使用 OpenCV 读取图像(Numpy.array 类型)
    img = cv2.imread(img_path)

    #是否对图像进行缩放
    if self.resize:
        img = cv2.resize(img, self.resize)

    #返回读取的图像
    return img
```

读取图像的操作在 next_batch 中完成,代码如下:

```
//ch5/data_utils/oxford_flower.py
def next_batch(self, which_set):
    …
    #以乱序或顺序的方式取一个 batch 的图像编号
    if self.shuffle:
        indexes = np.random.choice(
                    self.num_examples(which_set), self.batch_size)
    else:
        indexes = list(
            range(target_pointer[0] - self.batch_size, target_pointer[0]))

        target_pointer[0] = (target_pointer[0] + self.batch_size) %
                self.num_examples(which_set)

    #根据选出的 batch 图像编号读取 batch 图像
    imgs = np.stack([self.__read_images(target_image[idx])
                        for idx in indexes], axis = 0)

    #由于编号从 1～8189,需要将其减 1 得到 0～8188 作为索引读取标签
    labels = np.stack([self.__all_labels[target_image[idx] - 1]
                        for idx in indexes], axis = 0)

    #是否进行数据增强
    if do_augment:
        imgs = random_crop_and_flip(imgs, padding_size = 16)

    if self.__normalize:
```

```
            imgs = self.__normalization(imgs)

        return imgs, labels
```

最后，给出 102flowers 数据集的归一化后的均值与标准差：

```
mean = [0.28749102, 0.37729599, 0.43510646]
std = [0.26957776, 0.24504408, 0.29615187]
```

5.6　ImageNet 数据集

ImageNet 数据集整体呈金字塔状，由上到下分别是“目录”“子目录”和“图像集”，它的标志如图 5-4 所示。

图 5-4　ImageNet 的标志

ImageNet 总共含有 1500 万张图像，每张图像都含有图像级别的标注（即图像分类任务的标注形式），每一类至少含有 500 张图像。除此之外，ImageNet 还为其中的 103 万张图像提供边界框（Bounding Box），可以作为目标检测任务的数据集。ImageNet 提供两种数据下载方式，第一种可以通过下载所有图像的 URL 文件，在需要使用图像数据时，加载其对应的 URL 并下载使用即可。第二种是直接下载所有图像文件，共有 1TB 左右，当磁盘空间允许时，可以选择直接下载图像数据。

当讨论图像分类时，我们说使用 ImageNet 数据集往往并不是指使用了 ImageNet 的全量数据集，而是使用了 Large Scale Visual Recognition Challenge（ILSVRC）比赛的数据集。它本质上是全量 ImageNet 的一个子集，其中以 2012 年比赛使用的数据集最常用，即 ILSVRC 2012 的数据集。

ILSVRC 2012 含有 1200000 张训练图像，以及含有 150000 张图像的验证集和测试集，它们来源于 Flickr 和别的搜索引擎，数据集中一共含有 1000 个大类和 1860 个小类。当作精细粒度（fine-grained）分类时，需要使用小类进行区分。

由于 ImageNet 数据过大，在此仅对其做一个概述，不介绍其具体使用方法。有兴趣的读者可以通过 http://www.image-net.org/了解 ImageNet 相关信息及通过 http://image-net.org/challenges/LSVRC/2012/了解 ILSVRC 2012 相关信息。

5.7　小结

本节介绍了几个深度学习中常用的数据集。从数据层面来说，可以分为一维数据（IRIS、MNIST）和二维数据（SVHN、CIFAR、Oxford Flower、ImageNet）。从存储方式来说，可以分为以特殊形式存储的数据（MNIST、SVHN、CIFAR）和以字面值（人可以直接读

懂)存储的数据(IRIS、Oxford Flower、ImageNet)。从数据大小来说,可以分为小型数据集(IRIS、MNIST、SVHN、CIFAR、Oxford Flower)与大型数据集(ImageNet)。类似的分类方式还有很多,可以看出本书所介绍的数据集基本涵盖了不同表达形式与存储方式的数据集。

同时本节还介绍了如何自己实现数据加载逻辑,可以认为加载数据大致分为以下几个步骤:在构造函数中读取图像数据或文件名数据,读取数据的标签信息,完成训练集、验证集和测试集的划分。在 next_batch 函数中完成读取数据与标签必要的预处理操作,例如对样本的数据进行增强操作、归一化操作,对标签的"软化"(Soft Label)操作等,最终返回预处理好的数据及其对应标签即可。

本书主要对数据读取原理及使用进行讲解,因此只使用有详细讲解的数据集作为不同类型数据集使用的演示,不涉及 ImageNet 这种大型数据集。