第 3 章 基于 TensorFlow 2 的 ANN 技术

CHAPTER 3

本章将讨论并举例说明如何使用 TensorFlow 2 进行人工神经网络(Artificial Neural Network,ANN)的创建、训练和评估,这一过程是推理应用必不可少的环节。本章不提供完整的应用程序代码,仅对单独的概念和技术进行讲解,随后几章会对以上概念和技术组合,以得到完整模型。

本章将讨论以下内容:

- 获取数据集
- ANN 层
- 激活函数
- 创建模型
- 梯度计算
- 损失函数

3.1 获取数据集

依据 Google 的建议,借助 tf.data.Dataset 对象和 tf.data.Iterator 方法组成数据管道是 TensorFlow ANN 获取数据的经典方法。tf.data.Dataset 对象由一系列元素组成,其中每个元素包含一个或多个张量对象。tf.data.Iterator 是一种遍历数据集的方法,该方法可以访问数据集中连续的单个元素。

紧接着介绍两种构造数据管道的重要方法:第一种方法使用内存中的 NumPy 数组构造;第二种方法利用逗号分隔值(Comma-Separeted Value,CSV)文件构造。最后,将讨论如何使用二进制 TFRecord 格式存取数据。

3.1.1 从 NumPy 数组获取数据

先看一些简单的示例。

创建一个 NumPy 数组：

```
import tensorflow as tf
import numpy as np

num_items = 11
num_list1 = np.arange(num_items)
num_list2 = np.arange(num_items,num_items * 2)
```

用 from_tensor_slices()方法创建数据集：

```
num_list1_dataset = tf.data.Dataset.from_tensor_slices(num_list1)
```

用 make_one_shot_iterator()方法在 num_list1_dataset 上创建一个迭代器(iterator)：

```
iterator = tf.compat.v1.data.make_one_shot_iterator(num_list1_dataset)
```

用 get_next()方法将数据集、NumPy 数组和迭代器结合起来。

```
for item in num_list1_dataset:
    num = iterator1.get_next().numpy()
    print(num)
```

注意：由于使用的是 one-shot 迭代器，这段代码在同一程序中执行两次会引发错误。

也可以用 batch 方法批量访问数据，示例如下。第一个参数是要放入每个 batch 中的元素数；第二个参数是 drop_remainder，表示在少于 batch 中应放置元素数的情况下是否删除最后一批数据，默认 False 表示不删除。

```
num_list1_dataset = tf.data.Dataset.from_tensor_slices(num_list1).batch
(3, drop_remainder = False)
iterator = tf.compat.v1.data.make_one_shot_iterator(num_list1_dataset)
for item in num_list1_dataset:
    num = iterator.get_next().numpy()
    print(num)
```

zip 方法可以将特征和标签一起呈现，形成一个新的数据集。

```
dataset1 = [1,2,3,4,5]
dataset2 = ['a','e','i','o','u']
dataset1 = tf.data.Dataset.from_tensor_slices(dataset1)
dataset2 = tf.data.Dataset.from_tensor_slices(dataset2)
zipped_datasets = tf.data.Dataset.zip((dataset1, dataset2))
iterator = tf.compat.v1.data.make_one_shot_iterator(zipped_datasets)
for item in zipped_datasets:
    num = iterator.get_next()
    print(num)
```

可以用 concatenate 方法连接两个数据集，代码如下：

```
ds1 = tf.data.Dataset.from_tensor_slices([1,2,3,5,7,11,13,17])
ds2 = tf.data.Dataset.from_tensor_slices([19,23,29,31,37,41])
ds3 = ds1.concatenate(ds2)
print(ds3)
iterator = tf.compat.v1.data.make_one_shot_iterator(ds3)
for i in range(14):
    num = iterator.get_next()
    print(num)
```

还可以用如下代码取代迭代器：

```
epochs = 2
for e in range(epochs):
    for item in ds3:
      print(item)
```

注意：此处的外循环不会引发错误，因此在大多数情况下，该方法应作为首选方法。

3.1.2 从 CSV 文件获取数据

CSV 文件是一种非常普遍的数据存储方法。TensorFlow 2 处理 CSV 文件的方法很灵活，主要方法为 tf.data.experimental.CsvDataset。

1. CSV 示例 1

示例 1 从 CSV 文件的每行提取两项组成数据集，这两项均为浮点数。提取时，忽略文件的第 1 行，获取文件的第 1 列和第 2 列(列编号从 0 开始)。

```
filename = ["./size_1000.csv"]
record_defaults = [tf.float32] * 2      # 两个 float 类型的列
dataset = tf.data.experimental.CsvDataset(filename, record_defaults,
header = True, select_cols = [1,2])
for item in dataset:
   print(item)
```

2. CSV 示例 2

通过以下代码获取数据，数据集由一个必填浮点数、一个默认值为 0.0 的可选浮点数和一个 int 型整数组成，其中 CSV 文件中没有表头，只导入第 1、2 和 3 列。

```
# 见文件 Chapter_2.ipynb
filename = "mycsvfile.txt"
record_defaults = [tf.float32, tf.constant([0.0], dtype = tf.float32),
tf.int32,]
dataset = tf.data.experimental.CsvDataset(filename, record_defaults,
header = False, select_cols = [1,2,3])
for item in dataset:
    print(item)
```

3. CSV 示例 3

数据集由两个必填浮点数和一个必填字符串组成，CSV 文件有一个 header 变量。

```
filename = "file1.txt"
record_defaults = [tf.float32, tf.float32, tf.string ,]
dataset = tf.data.experimental.CsvDataset(filename, record_defaults,
header = False)
for item in dataset:
    print(item[0].numpy(), item[1].numpy(),item[2].numpy().decode() )
                                                    # 解码为二进制字符串
```

3.1.3　使用 TFRecords 存取数据

另一种普遍的数据存储方式是二进制文件格式 TFRecord。对于大型文件，最好选择二进制文件格式进行存取，其占用的磁盘空间更小，复制所需的时间更少，并且读取磁盘的效率更高。以上特点会影响数据管道的效率，从而影响模型的训练时间。TFRecord 格式还用多种方式进行了优化，

以便与 TensorFlow 一起使用。需要注意的是，数据必须在存储之前转换成二进制格式，并在读取时进行解码，该格式的使用相对复杂。

1. TFRecord 示例 1

本示例将演示 TFRecords 存取数据的基本原理（见文件 TFRecords.ipynb）。

TFRecord 文件是二进制字符串序列，因此必须在保存之前指定其结构，以便可以对其进行正确的写入和后续的读取。TensorFlow 支持两种结构：tf.train.Example 和 tf.train.SequenceExample。用户仅需将每个数据样本存储在其中一个结构中，然后将其序列化，使用 tf.python_io.TFRecordWriter 将其保存到磁盘即可。

下面的示例中，浮点数组 data 被转换为二进制格式，保存到磁盘。其中，feature 是一个字典，包含了在序列化和保存之前传递给 tf.train.Example 的数据。更详细的示例可参见 TFRecord 示例 2。

注意：TFRecords 支持的字节数据类型为 FloatList、Int64List 和 BytesList。

```
#文件: TFRecords.ipynb
import tensorflow as tf
import numpy as np

data = np.array([10.,11.,12.,13.,14.,15.])

def npy_to_tfrecords(fname,data):
    writer = tf.io.TFRecordWriter(fname)
    feature = {}
    feature['data'] =
tf.train.Feature(float_list = tf.train.FloatList(value = data))
     example = tf.train.Example(features = tf.train.Features(feature =
feature))
    serialized = example.SerializeToString()
    writer.write(serialized)
    writer.close()

npy_to_tfrecords("./myfile.tfrecords",data)
```

构造一个 parse_function 函数，该函数对从文件中读取的数据集进行解码，解码时需要一个与保存的数据具有相同名称和结构的字典（keys_to_features），读取记录的代码如下：

```
dataset = tf.data.TFRecordDataset("./myfile.tfrecords")

def parse_function(example_proto):
    keys_to_features = {'data':tf.io.FixedLenSequenceFeature([], dtype =
tf.float32, allow_missing = True) }
    parsed_features = tf.io.parse_single_example(serialized = example_
proto, features = keys_to_features)
    return parsed_features['data']
dataset = dataset.map(parse_function)
iterator = tf.compat.v1.data.make_one_shot_iterator(dataset)
#将数组作为一个 item 进行检索
item = iterator.get_next()
print(item)
print(item.numpy())
print(item[2].numpy())
```

2. TFRecord 示例 2

本示例将展示由 data 字典给出的更复杂的记录结构，如下所示：

```
filename = './students.tfrecords'
data = {
            'ID': 61553,
            'Name': ['Jones', 'Felicity'],
            'Scores': [45.6, 97.2]
        }
```

结合该记录结构，用 Feature()方法构造 tf.train.Example 类。注意观察如何对字符串进行编码。

```
ID = tf.train.Feature(int64_list = tf.train.Int64List(value = [data['ID']]))

Name = tf.train.Feature(bytes_list = tf.train.BytesList(value = [n.encode
('utf-8') for n in data['Name']]))

Scores = tf.train.Feature(float_list = tf.train.FloatList(value = data
['Scores']))

example = tf.train.Example(features = tf.train.Features(feature = {'ID': ID,
'Name': Name, 'Scores': Scores }))
```

将此记录序列化，并将其写入磁盘，步骤与 TFRecord 示例 1 相同：

```
writer = tf.io.TFRecordWriter(filename)
writer.write(example.SerializeToString())
writer.close()
```

若要解析该记录，只需要构造一个 parse_function 函数来反映记录的结构即可：

```
dataset = tf.data.TFRecordDataset("./students.tfrecords")

def parse_function(example_proto):
    keys_to_features = {'ID':tf.io.FixedLenFeature([], dtype = tf.int64),
                        'Name':tf.io.VarLenFeature(dtype = tf.string),
                        'Scores':tf.io.VarLenFeature(dtype = tf.float32)
                        }
    parsed_features = tf.io.parse_single_example(serialized = example_
proto, features = keys_to_features)
    return parsed_features["ID"],
parsed_features["Name"],parsed_features["Scores"]
```

下一步操作与 TFRecord 示例 1 相同：

```
dataset = dataset.map(parse_function)

iterator = tf.compat.v1.data.make_one_shot_iterator(dataset)
item = iterator.get_next()
#将记录作为一个 item 进行检索
print(item)
```

输出结果如下：

```
(<tf.Tensor: id = 264, shape = (), dtype = int64, numpy = 61553>, <tensorflow.
python.framework.sparse_tensor.SparseTensor object at 0x7f1bfc7567b8>,
<tensorflow.python.framework.sparse_tensor.SparseTensor object at 0x7f1bfc
771e80>)
```

现在，可以从 item 中提取数据了（请注意，字符串必须从字节中解码，其在 Python 3 的默认格式为 utf8）。还要注意，字符串和浮点数数组以稀疏数组返回，为了从记录中提取它们，此处使用了稀疏数组 values 方法：

```
print("ID: ",item[0].numpy())
name = item[1].values.numpy()
name1 = name[0].decode()returned
```

```
name2 = name[1].decode('utf8')
print("Name:",name1,",",name2)
print("Scores: ",item[2].values.numpy())
```

3.1.4　使用独热编码处理数据

独热编码(One-hot encoding,OHE)是从数据标签构成的张量,每个编码中只有一个1对应于标签值,其他位置均为0,即张量中仅有一位是hot(1)。

1. OHE 示例 1

在本示例中,用 tf.one_hot()方法将十进制数5转换为独热编码值0000100000。

```
y = 5
y_train_ohe = tf.one_hot(y, depth = 10).numpy()
print(y, "is ",y_train_ohe,"when one - hot encoded with a depth of 10")
#若独热编码位数为10,则5可表示为00000100000
```

2. OHE 示例 2

本示例使用从 fashion MNIST 时装数据集导入的示例代码来演示OHE方法。

原始标签是0～9的整数,进行独热编码后,标签2变为0010000000。但要注意索引与索引中存储的标签之间的区别。

```
import tensorflow as tf
from tensorflow.python.keras.datasets import fashion_mnist
tf.enable_eager_execution()
width, height, = 28,28
n_classes = 10

#加载数据集
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
split = 50000
#将特征训练集拆分为训练集和验证集
(y_train, y_valid) = y_train[:split], y_train[split:]

#使用独热编码处理标签
#然后转换回 numpy 进行显示
```

```
y_train_ohe = tf.one_hot(y_train, depth = n_classes).numpy()
y_valid_ohe = tf.one_hot(y_valid, depth = n_classes).numpy()
y_test_ohe = tf.one_hot(y_test, depth = n_classes).numpy()

#原始标签和独热编码标签之间的差异

i = 5
print(y_train[i])      #索引 i = 5 处标签的"原始"数值为 2
# 2
#请注意索引 5 和该索引处标签 2 之间的差异
print(y_train_ohe[i])
# 0. 0. 1. 0. 0.0 .0 .0. 0. 0.
```

接下来将研究神经网络的基本数据结构：神经层(本书中,有时会简称为层)。

3.2 ANN 层

ANN(Artificial Neural Network,人工神经网络)的基本数据结构是神经层(layer),许多相互连接的层构成了一个完整的 ANN。虽然人类大脑神经元和组成一个层的人造神经元之间只有很少的对应关系,使用“神经元”这个词可能会产生些误导,但是这不妨碍将一个层想象成一组神经元。请记住两者的不同,接下来将不加区分地使用术语“神经元”(neuron)。与任何计算机处理单元一样,神经元的特征在于其输入和输出,通常,一个神经元有多个输入和一个输出,每个输入连接都有一个权重 w_i。

图 3-1 展示了一个神经元。需要注意的是,除了普通的 ANN 之外,其他神经网络的激活函数 f 都是非线性的。神经网络中的一个普通神经元接收来自其他神经元的输入,每个神经元都有一个权重 w_i。神经网络通过调整这些权重进行学习,来使输入产生所需的输出。

输入乘以权重,再加上偏置,应用激活函数,可以得到神经元的输出(见图 3-2)。

图 3-2 显示了如何通过配置人工神经元和层来创建 ANN。

一个神经层的输出由式(3-1)给出：

$$\text{输出} = f\left(\sum_{1}^{n} \boldsymbol{W} \cdot \boldsymbol{X} + \text{bias}\right) \tag{3-1}$$

其中,$\boldsymbol{W}$ 是输入权重,$\boldsymbol{X}$ 是输入向量,f 为非线性激活函数,bias 表示偏置值。

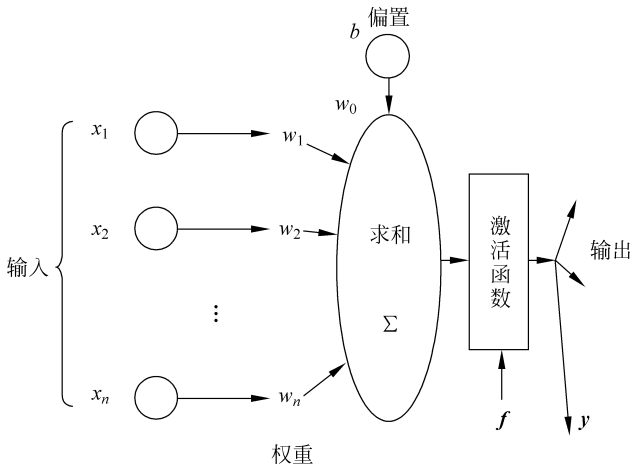

图 3-1　人工神经元

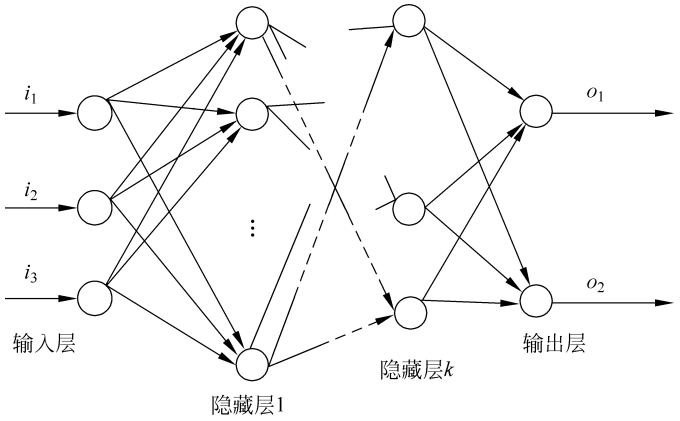

图 3-2　人工神经网络

神经层的类型很多，支持多种 ANN 模型结构。详细列表可参见 https://www.tensorflow.org/api_docs/python/tf/keras/layers。

接下来介绍一些更流行的神经层，并说明 TensorFlow 如何实现它们。

3.2.1　全连接层

全连接层(Dense Layer)是完全连接的神经层，上一层的所有神经元都与下一层的所有神经元相连。在一个全连接网络中，所有神经层都是全连接的(如果一个网络具有三个或更多隐藏层，则该网络为深度网络)。

一个全连接层由代码行 layer＝tf. keras. layers. Dense(n)构造，其中 n 是输出单元的数量。

注意：全连接层是一维的。详情请参阅 2.6 节部分内容。

3.2.2 卷积层

卷积层(Convolutional Layer)是一个神经层，该层的神经元被过滤器(通常是方形的)分组为若干个小块(patch)。过滤器(filter)在该层逐步滑动创建卷积层。过滤器对每个小块进行乘法和求和计算的过程，称为卷积。卷积网络(Convolutional nets，ConvNets)在图像识别和处理方面很有优势。

对于图像，卷积层的部分签名如下：

```
tf.keras.layers.Conv2D(filters, kernel_size, strides = 1, padding = 'valid')
```

在下面示例中，网络第一层有一个大小为(1，1)的过滤器，该层填充(padding)值为 valid，padding 参数的另一个取值为 same。

当 padding 设置为 same 时，需要在图层周围进行填充(通常用 0 进行填充)，以便在进行卷积之后，输出大小与原始图层大小相同。当 padding 值为 valid 时，则无须进行填充。如果步长(stride)和过滤器大小的组合不能完全适配该神经层，则该层将被截断，输出层大小小于该卷积层。

```
seqtial_Net = tf.keras.Sequential([tf.keras.layers.Conv2D( 1, (1, 1),
strides = 1, padding = 'valid')
```

3.2.3 最大池化层

当窗口在神经层上滑动时，最大池化层(Max Pooling Layer)在窗口内取最大值，该操作与卷积操作基本相同。

空间数据(即图像)最大池化的签名如下：

```
tf.keras.layers.MaxPooling2D(pool_size = (2, 2), strides = None, padding =
'valid', data_format = None)
```

要使用默认值，只需执行以下操作：

```
layer = tf.keras.maxPooling2D()
```

3.2.4　批标准化层和 Dropout 层

批标准化层(Batch Normalization)的输入和输出尺寸相同,激活项的平均值为 0,方差为 1,该方式有助于学习。批标准化规范了激活项,使输出既不会变得很小,也不会爆炸性地增大,而这两种情况都会阻止神经网络的学习。

批标准化层的方法签名如下:

```
tf.keras.layers.BatchNormalization(axis = -1, momentum = 0.99, epsilon =
0.001, center = True, scale = True, beta_initializer = 'zeros', gamma_
initializer = 'ones', moving_mean_initializer = 'zeros', moving_variance_
initializer = 'ones', beta_regularizer = None, gamma_regularizer = None, beta_
constraint = None, gamma_constraint = None)
```

要使用默认值,只需执行如下命令:

```
layer = tf.keras.layers.BatchNormalization()
```

Dropout 是指在训练过程中(而不是在推理过程中)随机关闭一定比例的神经元。该操作降低了网络对单个神经元的依赖,增强了网络的泛化能力。

Dropout 层的签名如下:

```
tf.keras.layers.Dropout(rate, noise_shape = None, seed = None)
```

其中,参数 rate 是关闭的神经元的比例。

可使用如下方式设置:

```
layer = tf.keras.layers.Dropout(rate = 0.5)
```

即随机选择 50%的神经元关闭。

3.2.5　Softmax 层

Softmax 层中,每个输出单元的激活值对应于该输出单元与已知标签匹配的概率。因此,具有最高激活值的输出单元就是网络的预测项。Softmax 层要求所预测的类互斥,这种情况下,该神经层输出的概率总和为 1。

Softmax 作为全连接层上的激活函数发挥作用。

调用示例如下：

```
model2.add(tf.keras.layers.Dense(10,activation = tf.nn.softmax))
```

以上代码增加了一个具有 10 个神经元的全连接 Softmax 层，神经元的激活值总和为 1。

接下来进一步讨论激活函数。

3.3 激活函数

神经网络具有非线性激活函数，所谓激活函数是指应用于神经元加权输入和的函数。在一般的神经网络模型中，线性激活单元无法将输入层映射到输出层。

常用的激活函数有许多，包括 sigmoid、tanh、ReLU 和带泄漏的 ReLU。关于这些激活函数的更加详尽的总结和图表，可参见 https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6。

3.4 创建模型

用 Keras 创建 ANN 模型的方法有如下四种。

- 方法 1：将参数传递给 tf.keras.Sequential。
- 方法 2：用 tf.keras.Sequential 的.add 方法。
- 方法 3：用 Keras 函数 API。
- 方法 4：子类化 tf.keras.Model 对象。

有关这四种方法的详细信息，请参阅第 2 章。

3.5 梯度计算

梯度下降法是大多数机器学习模型的重要组成部分，TenorFlow 的一大优势就在于它能够自动计算梯度。TensorFlow 提供了多种用于梯度计算的方法。

若启用动态图机制，有四种自动计算梯度的方法（这些方法也可以在计算图模式下工作）。

(1) tf.GradientTape：记录所有在上下文中的操作，并且通过调用

tf.gradient()获得任何上下文中计算得出的张量的梯度。

(2) tfe.gradients_function()：输入一个函数(如 f())并返回一个梯度函数(如 fg())，该梯度函数可以计算 fg()输出相对于 f()的参数或其子集的梯度。

(3) tfe.implicit_gradients()：与方法 2 类似，不同之处在于 fg()计算 f()的输出相对于这些输出依赖的所有可训练变量的梯度。

(4) tfe.implicit_value_and_gradients()：同方法 3 几乎一样，不同之处在于 fg()会同时返回函数 f()的输出。

最常用的方法是 tf.GradientTape。在其上下文中，随着计算的进行，会对这些计算进行记录(生成一个 tape)，以便可以使用 tf.gradient()对 tape 进行重放，实现自动微分。

在下面的代码中，当用 sum 计算时，tape 会将计算结果记录在 tf.GradientTape()的上下文中，以便通过调用 tape.gradient()进行自动微分。

注意观察在本例[weight1_grad]=tape.gradient(sum,[weight1])中列表如何使用。

默认情况下，tape.gradient()只能调用一次。

```
#默认情况下,在同一个 GradientTape 上下文中,tape.gradient 方法只能调用一次
weight1 = tf.Variable(2.0)
def weighted_sum(x1):
   return weight1 * x1
with tf.GradientTape() as tape:
   sum = weighted_sum(7.)
   [weight1_grad] = tape.gradient(sum, [weight1])
print(weight1_grad.numpy())
```

在接下来的示例中，将传递给 tf.GradientTape()方法的参数 persistent 设置为 True。该操作允许 tape.gradient()被多次调用。同样，可在 tf.GradientTape 上下文中计算加权和，并调用 tape.gradient()计算每项相对于其权值变量的梯度。

```
#如果需要,多次调用 tape.gradient()
#使用 GradientTape(persistent = True)
weight1 = tf.Variable(2.0)
weight2 = tf.Variable(3.0)
weight3 = tf.Variable(5.0)

def weighted_sum(x1, x2, x3):
    return weight1 * x1 + weight2 * x2 + weight3 * x3
```

```
with tf.GradientTape(persistent = True) as tape:
    sum = weighted_sum(7.,5.,6.)

[weight1_grad] = tape.gradient(sum, [weight1])
[weight2_grad] = tape.gradient(sum, [weight2])
[weight3_grad] = tape.gradient(sum, [weight3])

print(weight1_grad.numpy())   #7.0
print(weight2_grad.numpy())   #5.0
print(weight3_grad.numpy())   #6.0
```

接下来将研究损失函数，损失函数可在神经网络模型训练期间对模型进行优化。

3.6 损失函数

损失函数（即误差度量）是神经网络训练的必要部分，它是对网络在训练期间的计算输出与正确输出之间的差异程度的度量。通过对损失函数微分，可以找到一个调整各层之间连接权重的值，从而使神经网络的计算输出接近正确的输出。

最简单的损失函数是均方误差，如式(3-2)所示。

$$(1/n)\sum_{n}(y_n - \hat{y}_n)^2 \tag{3-2}$$

其中，y 是正确的标签值，$\hat{y}$ 是神经网络预测的标签值。

需要特别注意的是分类交叉熵损失函数，由式(3-3)给出。

$$-\sum_{n}(y_n \log(\hat{y}_n) + (1 - y_n)\log(1 - \hat{y}_n)) \tag{3-3}$$

当所有可能的类别中只有一个类别正确时，以及当 softmax 函数用作 ANN 最后一层的输出时，要用到该损失函数。

注意：根据反向传播的需要，这两个函数可以很好地进行区分。

3.7 小结

本章研究了一些支持神经网络创建和应用的技术；讨论了人工神经网络的数据表示、神经层、模型创建、梯度计算函数、损失函数以及模型的保存和恢复。以上内容是后续章节开发神经网络模型时所提及概念和技术的重要基础。

第 4 章将探索一些有监督的学习场景（包括线性回归、逻辑回归和 k 近邻）来进一步学习 TensorFlow 的使用。