第 3 章 TensorFlow 进阶

本章学习目标

- 了解 TensorFlow 的计算模型；
- 掌握 TensorFlow 嵌入层的知识；
- 掌握使用损失函数的方法；
- 掌握通过 TensorFlow 实现反向传播的方法；
- 掌握实现随机训练和批量训练的方法；
- 掌握创建分类器的方法；
- 掌握模型评估方法。

大家已经在第 2 章中学习了 TensorFlow 中张量、占位符、会话等基础知识，本章将把第 2 章所学的概念组合成计算图，帮助大家进一步了解 TensorFlow 的工作原理。最后将介绍一种简单的模型评估方法。

3.1 TensorFlow 的计算模型

TensorFlow 是一个通过计算图的形式来表述计算的编程系统，其中的每一个计算都是计算图上的一个节点，而节点之间的边描述了计算之间的依赖关系。TensorFlow 程序的执行一般分为两个阶段：定义计算图所有的计算和在会话中执行计算。

3.1.1 计算图的工作原理

接下来给出一段简单的代码，将之前所学的有关 TensorFlow 的对象表示成计算图，其中包含了创建数据集和计算图的简单操作，代码如下所示。

```
import tensorflow as tf
import numpy as np
sess = tf.Session()
x_input = np.array([1.0,2.0,3.0,4.0,5.0])
x_output = tf.placeholder(tf.float32)
y = tf.constant(3.0)
z = tf.add(x_output, y)
for x in x_input:
    print(sess.run(z, feed_dict = {x_output:x}))
```

输出结果如下所示。

```
4.0
5.0
6.0
7.0
8.0
```

上述代码在 TensorFlow 中的计算图如图 3.1 所示。

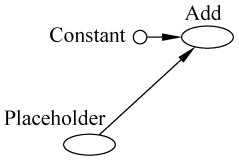

图 3.1　TensorFlow 中的工作原理

3.1.2　计算图的使用

一般情况下，TensorFlow 会自动维护一个默认的计算图，通过 tf.get_default_graph()函数可以获取该计算图。以下代码展示了如何查看一个运算所属的计算图的方法。

```
1  import tensorflow as tf
2  import numpy as np
3  sess = tf.Session()
4  x_input = np.array([1.0,2.0,3.0,4.0,5.0])
5  x_output = tf.placeholder(tf.float32)
6  y = tf.constant(3.0)
7  z = tf.add(x_output, y)
8  for x in x_input:
9      print(sess.run(z, feed_dict = {x_output:x}))
10 print(z.graph is tf.get_default_graph())
```

输出如下所示。

```
4.0
5.0
6.0
7.0
8.0
True
```

上述代码中最后一行通过 tf.get_default_graph()函数查看了张量 z 所属的计算图。由于代码中没有为 z.graph 指定特定的计算图，因此默认将张量 z 归入默认计算图中，因此张量 z 所属的计算图等于默认计算图，输出为 True。

TensorFlow 同样提供了自定义生成新的计算图的方法，通过 tf.Graph()函数可以生成新的计算图，不同的计算图之间张量和运算会相互隔离。在不同的计算图上定义和使用张量进行计算的方法如下所示。

```
import tensorflow as tf
graph_1 = tf.Graph()
with graph_1.as_default():
    #在计算图 graph_1 中定义一个变量"v",初始化变量"v"为维度为[1,3] 的张量[1,2,3]
    v = tf.get_variable("v", initializer =
tf.constant_initializer([1,2,3])(shape = [1,3]))
graph_2 = tf.Graph()
with graph_2.as_default():
    #在计算图 graph_2 中定义一个变量"v",初始化变量"v"为维度为[2,3]的零张量
    v = tf.get_variable("v", initializer =
                            tf.zeros_initializer()(shape = [2,3]))
with tf.Session(graph = graph_1) as sess:
    tf.global_variables_initializer().run()
    with tf.variable_scope("", reuse = True):
        print(sess.run(tf.get_variable("v")))
with tf.Session(graph = graph_2) as sess:
    tf.global_variables_initializer().run()
    with tf.variable_scope("", reuse = True):
        print(sess.run(tf.get_variable("v")))
```

输出如下所示。

```
[[1. 2. 3.]]
[[0. 0. 0.]
 [0. 0. 0.]]
```

从输出结果不难看出,上述代码中生成的两个计算图在运行时变量“v”的值是相互隔离的。

TensorFlow 中的计算图不仅可以用来隔离张量和计算,它还提供了管理张量和计算的机制。计算图可以通过 tf.Graph.device()函数来指定运行计算的设备。例如通过下列代码可以在 GPU 上执行简单的算术计算。

```
import tensorflow as tf
graph_1 = tf.Graph()
x = tf.constant([1.0,2.0,3.0])
y = tf.ones([1,3])
result = x + y
with graph_1.device("/gpu:0"):
    with tf.Session() as sess:
        print(sess.run(result))
```

输出如下所示。

```
[[2. 3. 4.]]
```

使用 GPU 加速可以显著提升 TensorFlow 的计算速度,后续章节将会进一步讲解使用 GPU 加速 TensorFlow 的方法。

在 TensorFlow 中，可以通过集合来管理一个计算图中不同类型的资源，例如通过 tf.add_to_collection 函数可以将各类资源（张量、变量或者运行程序所需的队列资源等）加入一个或多个集合中，然后通过 tf.get_collection 获得一个集合里面的所有资源。TensorFlow 自动管理了一些常用的集合，具体如表 3.1 所示。

表 3.1　常见的符号类型

集合名称	集合内容	使　用
tf.GraphKeys.VARIABLES	所有变量	持久化 TensorFlow 模型
tf.GraphKeys.TRAINABLE_VARIABLES	可学习的变量（一般指神经网络中的参数）	模型训练、生成模型可视化内容
tf.GraphKeys.SUMMARIES	日志生成相关的张量	计算可视化
tf.GraphKeys.QUEUE_RUNNERS	处理输入的 QueueRunner	输入处理
tf.GraphKeys.MOVING_AVERAGE_VARIABLES	所有计算了滑动平均值的变量	计算变量的滑动平均值

3.2　TensorFlow 的嵌入层

本节将介绍如何在一个计算图中进行多个乘法操作的方法。

首先，创建一个矩阵与占位符。

```
import numpy as np
import tensorflow as tf
x = np.array([[1.0, 2.0, 3.0],
              [4.0, 5.0, 6.0],
              [7.0, 8.0, 9.0]])
x_vals = np.array([x, x + 1])
x_data = tf.placeholder(tf.float32, shape = (3, 3))
```

接下来，创建将在矩阵乘法和加法中使用的常量矩阵。

```
m_1 = tf.constant([[1.0], [2.0], [3.0]])
m_2 = tf.constant([[5.0]])
a_1 = tf.constant([[6.0]])
```

然后，进行声明操作，将矩阵乘法和矩阵加法表示成计算图。

```
prod_1 = tf.matmul(x_data, m_1)
prod_2 = tf.matmul(prod_1, m_2)
add_1 = tf.add(prod_2, a_1)
```

最后，输出两矩阵经过相乘操作后的和。

```
with tf.Session() as sess:
    for x_val in x_vals:
        print(sess.run(add_1, feed_dict = {x_data:x_val}))
```

输出结果如下所示。

```
[[ 76.]
 [166.]
 [256.]]
[[106.]
 [196.]
 [286.]]
```

从上述代码编写的流程中可以看到，在 TensorFlow 中，想要通过计算图运行数据，需要先声明数据形状，预估经过相应操作后返回值的形状来创建占位符。但是，实际情况中很难预先准确知晓返回值的维度，或者遇到返回值的维度总是在变化的情况。此时，需要将变化的维度，或者事先不知道的维度设为“None”。示例代码如下。

```
x_data = tf.placeholder(tf.float32, shape = (None, None))
```

将本节前面示例代码中占位符的维度参数替换后输出如下所示。

```
[[ 76.]
 [166.]
 [256.]]
[[106.]
 [196.]
 [286.]]
```

可以看到，输出结果与原代码一致。

3.3 TensorFlow 的多层

深度神经网络中神经层的数量往往是大于 1 的，3.2 节已经介绍了如何在一个计算图中进行多项操作的方法，接下来将介绍如何在多个层之间连接并传播数据，以及自定义 Layer 的方法。有时，计算图会因为过大而无法完整查看，此时需要通过对各层 Layer 和各项操作进行层级命名管理。

TensorFlow 中的图像函数是处理四维图片的，图片的 4 个维度分别为图片数量、图片高度、图片宽度以及颜色通道。首先，确定图片的思维参数，然后通过 numpy 中的 np.random.uniform()函数创建一个像素为 3×3 的二维图片，并创建用于传入图片的占位符。代码如下所示。

```
x_shape = [1, 3, 3, 1]
x = np.random.uniform(size = x_shape)
x_input = tf.placeholder(tf.float32)
```

上述代码中 x_shape 的 4 个参数分别为“1，3，3，1”，其中第一个“1”表示图片数量为 1，第一个“3”表示图片高度为 3，第二个“3”表示图片宽度为 3，第二个“1”表示该 2D 图片是

单颜色通道的。

然后，通过 TensorFlow 中的 conv2d()函数卷积大小为 2×2 的常量窗口来创建一个用来过滤刚才创建的 3×3 像素的图片。其中 conv2d()函数需要传入滑动窗口、过滤器和步长。在此，选用 2 作为各个方向步长值(本例中由于该窗口需要在 4 个方向上移动，因此需要在 4 个方向上指定对应的步长值)，选用均值滤波器作为过滤器(滤波器)。padding 为是否进行扩展，这里选用“SAME”。创建一个常量为 0.25 的向量与 2×2 的窗口卷积。

```
my_filter = tf.constant(0.25, shape = [2, 2, 1, 1])
my_strides = [1, 2, 2, 1]
mov_avg_layer = tf.nn.conv2d(x_input, my_filter, my_strides, padding = "SAME", name =
"Moving_Avg_Window")
```

通过自定义 Layer 对第一层的输出数据进行“y＝wx＋b”的操作。

```
def custom_layer(input_matrix):
    input_matrix_sqeezed = tf.squeeze(input_matrix)
    a = tf.constant([[1.0, 2.0], [ - 1.0, 3.0]])
    b = tf.constant(1.0, shape = [2,2])
    temp1 = tf.matmul(a, input_matrix_sqeezed)
    temp = tf.add(temp1, b)
    return(tf.sigmoid(temp))
```

将刚刚定义的 Layer 导入计算图中，并通过 tf.name_scope()函数来对该 Layer 进行命名。

```
with tf.name_scope("Custom_layer") as scope:
custom_layer1 = custom_layer(mov_avg_layer)
```

最后通过 sess.run()执行计算图，并输出结果，完整代码如下所示。

```
1  import numpy as np
2  import tensorflow as tf
3  #将 size 设为[1, 3, 3, 1]是因为 tf 中图像函数是处理四维图片的
4  #这四维依次是: 图片数量,高度,宽度,颜色通道
5  x_shape = [1,3,3,1]
6  x = np.random.uniform(size = x_shape)
7  #tf.nn.conv2d 中 name 表明该 layer 命名为"Moving_Avg_Window"
8  #卷积核为[[0.25,0.25],[0.25,0.25]]
9  x_input = tf.placeholder(tf.float32, shape = x_shape)
10 my_filter = tf.constant(0.25, shape = [2, 2, 1, 1])
11 my_strides = [1, 2, 2, 1]
12 mov_avg_layer = tf.nn.conv2d(x_input, my_filter, my_strides, padding = "SAME", name =
   "Moving_Avg_Window")
13 #自定义 layer,对卷积操作之后的输出做操作
14 def custom_layer(input_matrix):
15     input_matrix_sqeezed = tf.squeeze(input_matrix)
```

```
    a = tf.constant([[1.0, 2.0], [-1.0, 3.0]])
    b = tf.constant(1.0, shape= [2,2])
    temp1 = tf.matmul(a, input_matrix_sqeezed)
    temp = tf.add(temp1, b)
    return(tf.sigmoid(temp))
#把刚刚自定义的 layer 加入到计算图中,并给予自定义的命名(利用 tf.name_scope())
with tf.name_scope("Custom_layer") as scope:
    with tf.Session() as sess:
        custom_layer1 = custom_layer(mov_avg_layer)
        #为占位符传入 3×3 图片,并执行计算图
        print(sess.run(custom_layer1, feed_dict = {x_input: x}))
```

输出如下所示。

```
[[0.90566695 0.7627088 ]
 [0.668813   0.74542195]]
```

在实际应用中往往需要应对多层级的复杂情况,此时通过折叠和展开已命名的自定义层级 Layer,可以让已命名的层级 Layer 和操作的可视化图更加清晰。

3.4 TensorFlow 实现损失函数

损失函数(loss function)用于衡量模型的输出值与预测值之间的差距,对神经网络进行优化的目标便是根据损失函数来定义的。本节将介绍如何在 TensorFlow 中实现适用于分类问题和回归问题的各种经典损失函数,并通过具体示例来介绍如何根据具体问题定义损失函数以及不同的损失函数对训练结果的影响。

3.4.1 损失函数

为了优化学习算法,通常需要对模型的训练结果进行评估,TensorFlow 通过损失函数来对比模型实际预测结果与期望结果之间的差距。本节主要对分类问题和回归问题中常见的经典损失函数进行讲解。分类问题与回归问题同属于监督学习的范畴,分类问题主要方法为将不同的样本划分到事先设定好的标签类别下,例如图像识别中分辨照片中的人物性别的问题。回归问题主要解决对具体数值的预测,例如根据车主往年的保险理赔情况预测车主下一年的保费。

首先,对回归算法的损失函数进行讲解。回归算法常用于预测连续因变量。创建预测序列和目标序列作为张量,预测序列为 −1 到 1 之间的等差数列,目标值为 0,代码如下所示。

```
import tensorflow as tf
x = tf.linspace(-1.0,1.0,500)
target = tf.constant(0.0)
```

L^1 正则损失函数,即绝对值损失函数。其大小等于预测值与目标值差值的绝对值之

和。由于 L^1 正则损失函数在目标值附近不平滑，在实际应用中容易出现无法收敛的情况。示例代码如下所示。

```
l1_y = tf.abs(target - x)
l1_y_output = sess.run(l1_y)
```

L^2 正则损失函数，又被称为欧几里得损失函数。其大小等于预测值与目标值差值的平方和。与 L^1 正则损失函数不同的是，L^2 正则损失函数在目标值附近具有良好的曲度，这有利于算法在目标值附近收敛，离目标值越近收敛速度越慢。示例代码如下所示。

```
l2_y = tf.square(target - x)
l2_y_output = sess.run(l2_y)
```

Pseudo-Huber 损失函数是 Huber 损失函数的连续、平滑近似，保证函数各阶可导。该损失函数通过 L^1 和 L^2 正则改善极值附近的平滑程度，使得目标值附近连续。该损失函数的表达式依赖于参数 delta 的值，后面将会通过绘图的方式反映 delta 取值差异对损失函数造成的影响。示例代码如下所示。

```
delta1 = tf.constant(1.0)
phuber1_y = tf.multiply(tf.square(delta1),tf.sqrt(1.0 + tf.square((target - x)/delta1)) -
1.0)
phuber1_y_output = sess.run(phuber1_y)
delta2 = tf.constant(5.0)
phuber2_y = tf.multiply(tf.square(delta2),tf.sqrt(1.0 + tf.square((target - x)/delta2)) -
1.0)
phuber2_y_output = sess.run(phuber2_y)
```

分类损失函数主要用来评估预测分类结果的准确性的度量，接下来将对常见的几类分类损失函数进行介绍。

(1) Hinge 损失函数也称作铰链损失函数，它主要用于支持向量机(SVM)，部分情况下也会被用来评估神经网络算法。示例代码如下所示。

```
hinge_y = tf.maximum(0.0, 1.0 - tf.multiply(target, x))
hinge_y_output = sess.run(hinge_y)
```

上述代码中实际上计算了两个目标类(−1,1)之间的损失值，代码中设定的目标值为1，此时预测值越靠近 1 则损失值越小。

(2) 交叉熵损失函数(cross-entropy loss function)主要用来判定预测值与目标值的接近程度，预测值与目标值的距离越近，交叉熵的值越小。交叉熵损失函数可以缓解方差损失函数更新权重过慢的问题。由于交叉熵损失函数具有非负性，优化目标可以简化为求得函数极小值；预测值距离目标值越近损失值越接近于 0。示例代码如下所示。

```
xentropy_y = -tf.multiply(target, tf.log(x)) - tf.multiply((1.0 - target), tf.log(1.0 - x))
xentropy_y_output = sess.run(xentropy_y)
```

(3) Sigmoid 交叉熵损失函数(Sigmoid cross entropy loss function),该损失函数在计算交叉熵损失函数之前先将预测值进行了 Sigmoid 函数转换。Sigmoid 函数能够把输入的连续实值“压缩”到 0 和 1 之间,如图 3.2 所示。

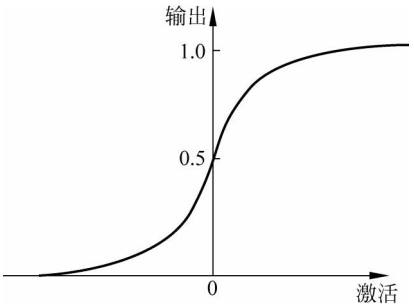

图 3.2 Sigmoid 函数

从图 3.2 可以看到,经过 Sigmoid 函数转换的数据被“压缩”到了(0,1)的区间内,经过这样处理的数据将提升计算损失值的效率。示例代码如下所示。

```
xentropy_sigmoid_y = tf.nn.sigmoid_cross_entropy_with_logits(logits = y, labels =
targets)
xentropy_sigmoid_y_output = sess.run(xentropy_sigmoid_y)
```

(4) 加权交叉熵损失函数(Weighted cross entropy loss),实际应用中难免遇到正负样本数量差距悬殊的情况,在负样本远大于正样本的情况下,如果不对正负样本的损失进行权重调整,那么学习模型就会把所有结果都预测成负样本(因为正样本的数目很少,所以都预测为负样本的话,总的损失也会很小),显然这样的模型是没有意义的。此时,需要给正样本的损失加上一定的权重,当正样本分类错误的时候,乘以一个非常大的权重,使得对正样本的预测错误不至于被模型忽视。当负样本分类错的时候,乘以一个较小的权重,由于负样本总数众多,此时负样本的损失也不会与实际情况出现较大的偏差。示例代码如下。

```
weight = tf.constant(0.5)
xentropy_weighted_y = tf.nn.weighted_cross_entropy_with_logits(logits = x, targets =
targets, pos_weight = weight)
xentropy_weighted_y_output = sess.run(xentropy_weighted_y)
```

(5) Softmax 交叉熵损失函数(Softmax cross-entropy loss)是作用于非归一化的输出结果,指针对单个目标分类的计算损失。通过 Softmax 函数将输出结果转化成概率分布,然后计算真值概率分布的损失,示例代码如下。

```
1  import tensorflow as tf
2  with tf.Session() as sess:
3    unscaled_logits = tf.constant([[1.0,2.0,3.0,4.0]])
4    target = tf.constant([[0.5,1.5,2.0,2.5]])
5    softmax_xentropy = tf.nn.softmax_cross_entropy_with_logits(logits = unscaled_logits,
     labels = target)
6    print(sess.run(softmax_xentropy))
```

输出如下所示。

```
[9.361233]
```


(6) 在只有一个正确答案的分类问题中，TensorFlow 提供了一种更加高效的算法，即稀疏 Softmax 交叉熵损失函数(Sparse Softmax cross-entropy loss)，该损失函数将分类为 True 的目标值转化成指数，而 Softmax 交叉熵损失函数将目标转化成概率分布，示例代码如下所示。

```
import tensorflow as tf
with tf.Session() as sess:
    unscaled_logits = tf.constant([[1.0,2.0,3.0]])
    sparse_target = tf.constant([2])
    sparse_xentropy = tf.nn.sparse_softmax_cross_entropy_with
            _logits(logits = unscaled_logits, labels = sparse_target)
    print(sess.run(sparse_xentropy))
```

输出如下所示。

```
[0.40760595]
```

在深度学习中，图片一般是用非稀疏的标签进行表示的，此时使用 tf. nn. sparse_softmax_cross_entropy_with_logits()函数的效率比使用 tf. nn. softmax_cross_entropy_with_logits()函数的效率更高。

3.4.2 损失函数工作原理及实现

为了方便大家直观地了解个损失函数之间的区别，在此通过 matplotlib 分别绘制 3.4.1 节中讲到的回归算法损失函数和分类算法损失函数的曲线图。

首先，用 matplotlib 绘制回归算法的损失函数。

```
import tensorflow as tf
import matplotlib.pyplot as plt
x = tf.linspace(-1.0,1.0,500)
target = tf.constant(0.0)
l1_y = tf.abs(target - x)
l2_y = tf.square(target - x)
delta1 = tf.constant(.1)
phuber1_y = tf.multiply(tf.square(delta1),tf.sqrt(1.0 + tf.square((target - x)/
delta1))-1.0)
delta2 = tf.constant(3.0)
phuber2_y = tf.multiply(tf.square(delta2), tf.sqrt(1.0 + tf.square((target - x)/
delta2))-1.0)
with tf.Session() as sess:
    x_array = sess.run(x)
    l1_y_output = sess.run(l1_y)
```

```
    l2_y_output = sess.run(l2_y)
    phuber1_y_output = sess.run(phuber1_y)
    phuber2_y_output = sess.run(phuber2_y)
    plt.plot(x_array, l1_y_output, "g:", label = "L1 Loss")
    plt.plot(x_array, l2_y_output, "r-", label = "L2 Loss")
    plt.plot(x_array, phuber1_y_output, "k--", label = "P-Huber Loss(1.0)")
    plt.plot(x_array, phuber2_y_output,"b-.", label = "P-Huber Loss(5.0)")
    # 设置 y 轴刻度的范围,从 0 到 1
    plt.ylim(0, 1)
    #设置图例处于图标上部中心位置,字号为 11
    plt.legend(loc = " upper center ", prop = {"size": 11})
    # 输出图形
    plt.show()
```

输出如图 3.3 所示。

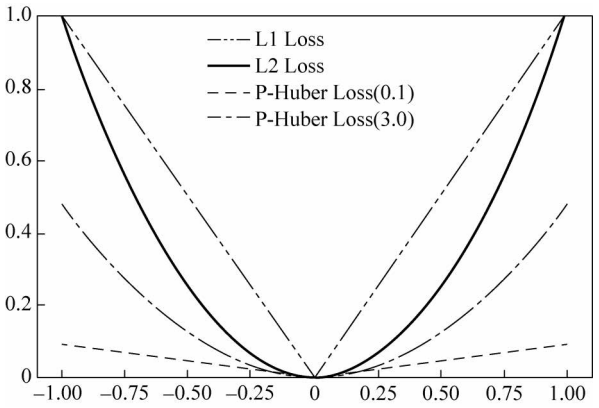

图 3.3　回归算法损失函数曲线图

对比图 3.3 中 L^1 正则损失函数曲线和 L^2 正则损失函数曲线，可以看到，L^1 正则损失函数曲线在极值点附近的梯度变化非常大，在极值点附近，很小的损失值也会产生非常大的误差，这一特性是不利于模型收敛的；而 L^2 正则损失函数曲线则相对平滑许多，梯度随着损失函数的减小而减小，这一特性使得它在最后的训练过程中能得到更精确的结果，即使在学习率固定的情况下也能收敛。

由于均方误差在误差较大点时的损失远大于平均绝对误差，这使得模型对异常值赋予更大的权重值，并花费更多的资源减小异常值造成的误差，这会让模型的整体表现下降。因此，当训练数据中含有较多的异常值时，采用 L^1 正则损失函数可能更加高效。在需要对所有观测值进行处理的情况下，如果利用 L^2 正则损失函数进行优化会得到所有观测的均值，而使用 L^1 正则损失函数则得到所有观测的中值。在应对异常值时，取观测数据的中值比取均值的鲁棒性更好。

在实际操作中，如果需要检测异常值，建议采用均方误差；如果异常值极有可能是坏点，采用平均绝对误差的效果可能更好。L^1 正则损失函数在处理异常值时具有更好的鲁棒性，但它的导数不连续，优化效率很低；相比较而言，L^2 正则损失函数对于异常值非常敏感，但在优化中表现稳定且更加精确。

上文比较了 L^1 正则损失函数和 L^2 正则损失函数的差异和优缺点，然而在现实中存在着许多情况是上述两种损失函数都很难处理的。例如某个任务中 95%的数据为正样本，而其余的 5%数据为负样本。那么利用 L^1 正则损失函数优化的模型的预测结果总是为正样本的值，这样会完全忽视剩下那 5%的负样本；对于 L^2 正则损失函数而言，由于异常值会带来很大的损失，将使得模型的预测值倾向于 5%的负样本的值。这两种结果在实际的业务场景中都会使模型失去意义。此时可以采用之前介绍的 Pseudo-Huber 损失函数。

与 L^2 正则损失函数相比，Pseudo-Huber 损失函数相对异常值敏感度低，保持了函数曲线的平滑性，这样便于收敛。从图 3.3 中可以看到，参数值为“3”时函数曲线的两侧梯度大于参数为“0,1”时的曲线，Pseudo-Huber 损失函数的参数值越大，则两边的线性部分越陡峭。参数的选择对该损失函数至关重要，因为它决定了模型处理异常值的行为。当残差大于设定的参数值时使用 L^1 正则损失函数，残差小于设定的参数值时则使用更为平滑的 L^2 正则损失函数。

Pseudo-Huber 损失函数综合了 L^1 正则损失函数和 L^2 正则损失函数的特性，不仅可以保持损失函数连续可导，利用 L^2 正则损失函数的梯度随误差减小的特性让优化更易于收敛，优化结果也更加精确，并且在应对异常值时具有优秀的鲁棒性。

了解了常见的回归算法损失函数的工作原理及特性，接下来绘制 3.4.1 节中介绍的常见分类算法损失函数的图像。

```
import tensorflow as tf
import matplotlib.pyplot as plt
x = tf.linspace(-3.0,5.0,500)
target = tf.constant(1.0)
targets = tf.fill([500,], 1.0)
hinge_y = tf.maximum(0.0, 1.0-tf.multiply(target, x))
xentropy_y = -tf.multiply(target, tf.log(x)) - tf.multiply((1.0 - target), tf.log(1.0 - x))
xentropy_sigmoid_y = tf.nn.sigmoid_cross_entropy_with_logits(logits = x, labels =
targets)
weight = tf.constant(0.5)
xentropy_weighted_y = tf.nn.weighted_cross_entropy_with_logits(logits = x, targets =
targets, pos_weight = weight)
with tf.Session() as sess:
    x_array = sess.run(x)
    hinge_y_output = sess.run(hinge_y)
    xentropy_y_output = sess.run(xentropy_y)
    xentropy_sigmoid_y_output = sess.run(xentropy_sigmoid_y)
    xentropy_weighted_y_output = sess.run(xentropy_weighted_y)
    plt.plot(x_array, hinge_y_output, "b-", label = "Hinge Loss")
    plt.plot(x_array, xentropy_y_output, "r--", label = "Cross Entropy Loss")
    plt.plot(x_array, xentropy_sigmoid_y_output, "k-.", label = "Cross Entropy Sigmoid
    Loss")
    plt.plot(x_array, xentropy_weighted_y_output, "g:", label = "Weighted Cross Entropy
    Sigmoid Loss(x0.5)")
    # 设置 y 轴刻度的范围为 -1.5～3
    plt.ylim(-1.5, 3)
```

```
    #设置图例位置为图标下部靠右侧,字号为 11
    plt.legend(loc = "lower right", prop = {"size":11})
    #输出图形
    plt.show()
```

输出如图 3.4 所示。

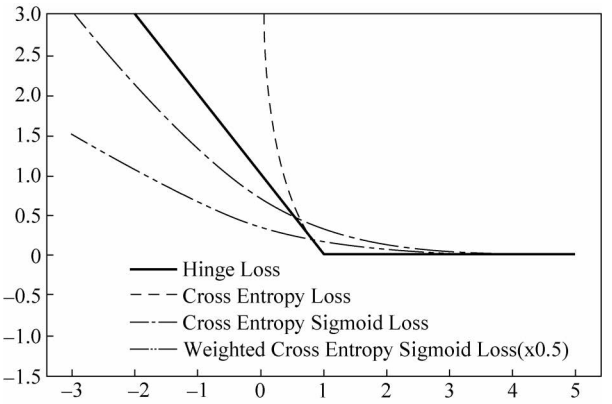

图 3.4　分类算法损失函数曲线图

从图 3.4 可以看出 Hinge 损失函数是典型的凸函数，即并不鼓励分类器过度自信，让某个可以正确分类的样本距离分割线的距离超过 1 并不会有任何奖励。从而使得分类器可以更专注整体的分类误差。

3.5　TensorFlow 实现反向传播

TensroFlow 可以在维持操作状态的同时，基于反向传播算法自动对模型的数据进行更新。使用监督学习的方式设置神经网络参数需要贴好标签的训练数据集。以判断学生成绩是否及格为例，训练数据集就是学生的成绩（其中包括及格的学生成绩和不及格的学生成绩）。在监督学习中，通常会要求模型对已知答案的数据集的预测值尽可能地接近目标值。通过调整神经网络中的参数对训练数据进行拟合，使得模型具备对未知样本进行准确预测的能力。

3.5.1　反向传播算法

反向传播算法（Backpropagation）是目前最常见的神经网络优化算法，它是利用链式法则递归计算表达式的梯度的方法，在多层神经网络的训练中具有重要的意义，理解反向传播算法对于设计、构建和优化神经网络非常关键。反向传播经常被误解为神经网络的整个学习算法，事实上，在神经网络在领域它只用于梯度计算，该算法适用于大多数简化多变量复合求导的过程。

图 3.5 展示了使用反向传播算法对神经网络进行优化的过程。通过 TensorFlow 实现反向传播的过程。

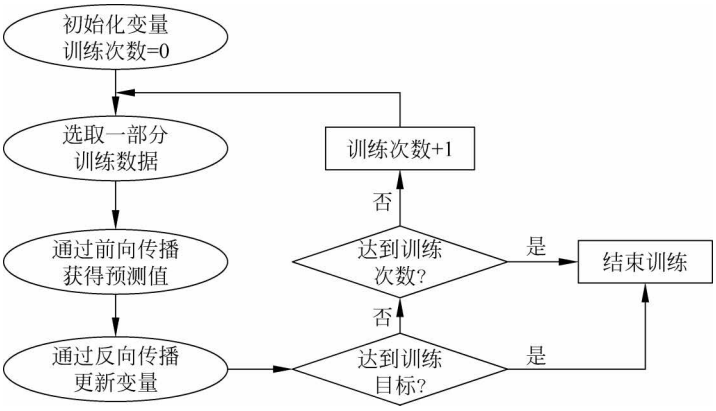

图 3.5　反向传播算法优化神经网络的流程

从图 3.5 中可以看出，反向传播算法对神经网络进行优化的过程实际上是一个迭代的过程。在每次迭代开始前，先选取一小部分训练数据（这部分数据集被称作 batch），通过前向传播算法，根据 batch 数据得到神经网络模型的预测值。根据预测值与目标值的误差评估模型的学习效果。然后，通过反向传播算法，根据预测值与目标值之间的差值，更新神经网络中的各参数值，使得神经网络在对应 batch 上的预测结果靠近目标值。

3.5.2　反向传播算法的工作原理及实现

之前已经介绍了关于在 TensorFlow 中创建对象和操作对象，以及度量预测值与目标值之间差值的损失函数的方法，接下来将介绍如何通过计算图来更新变量和最小化损失函数，从而反向传播预测值与目标值之间的差值。通过声明函数，TensorFlow 在计算图中会根据输入数据，通过损失函数来调节神经网络中相应的参数。

在 TensorFlow 中，实现反向传播算法首先需要使用 TensorFlow 表达一个 batch 数据。使用常量来表达一个 batch 数据的方法如下所示。

```
x = tf.constant([[1.0,2.0,3.0]])
```

值得注意的是，总是用常量来表示迭代中所选取的数据会极大地提升 TensorFlow 的计算图的节点数量。通常情况下训练一个神经网络需要的迭代次数在百万次以上，如果使用常量表示所选取的数据，计算图会变得非常庞大，这对资源的利用率极低。为了降低迭代中生成的节点数量，提升学习效率，需要用到之前所讲的占位符机制。通过占位符对数据进行“占位”，所占位置的数据值在程序运行时再指定。程序只需要将数据通过占位符传入 TensorFlow 计算图，这样便解决了需要生成大量常量来提供输入数据的问题。

本节通过回归算法的例子来演示 TensorFlow 中反向传播算法的实现方法。从均值为 5、标准差为 0.1 的正态分布中随机抽样 100 个数，然后乘以变量 A，指定损失函数为 delta 值为 0.25 的 Pseudo-Huber 损失函数。上述过程可以理解成实现函数 $x \times A = \text{target}$，$x$ 为均值为 5 的 100 个随机数，target 为 10，由此可得 A 的目标值为 2。

(1) 首先,导入相应的模块,代码如下所示。

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

(2) 生成数据 100 个随机数以及 100 个目标数,并声明占位符和变量 A。代码如下所示。

```
x = np.random.normal(5, 0.1, 100)
y = np.repeat(10.0, 100)
x_data = tf.placeholder(shape = [1], dtype = tf.float32)
y_target = tf.placeholder(shape = [1], dtype = tf.float32)
A = tf.Variable(tf.random_normal(shape = [1]))
```

(3) 加入乘法操作,实现函数 $x \times A =$ target。

```
my_output = tf.multiply(x_data, A)
```

(4) 指定损失函数为 delta 值为 0.25 的 Pseudo-Huber 损失函数。

```
deltal = tf.constant(0.25)
loss = tf.multiply(tf.square(deltal), tf.sqrt(1.0 + tf.square((my_output - y_target)/
deltal)) - 1.0)
```

(5) 初始化所有变量。

```
init = tf.global_variables_initializer()
sess.run(init)
```

(6) 声明变量的优化器,为优化器算指定学习率,优化算法的步长值取决于所设定的学习率,在此设置学习率为 0.05。学习率的选择对学习效果影响重大,一般来说,较小的学习率意味着更精确的预测结果,但是收敛速度较慢,通常在算法表现不够稳定时采用较小的学习率;较大的学习率意味着更快的收敛速度,但是预测的精度相应降低,通常在算法收敛速度过慢的情况下增大学习率从而提升收敛速度。

```
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(loss)
```

(7) 创建一个损失值 batch 数据。

```
loss_batch = []
```

(8) 开始训练算法。在调用 sess.run 时,需要使用 feed_dict 来设定 x 的取值。在得到一个 batch 的前向传播结果之后,需要定义一个损失函数来刻画当前的预测值和真实答案之间的差距。然后通过反向传播算法来调整神经网络参数的取值缩小预测值与目标值之间

的距离。


```
for i in range(100) :
    rand_index = np.random.choice(100)
    rand_x = [x[rand_index]]
    rand_y = [y[rand_index]]
    sess.run(train_step, feed_dict = {x_data: rand_x, y_target: rand_y})
    print("第" + str(i+1) + "次 A = " + str(sess.run(A)))
    print("损失值为" + str(sess.run(loss, feed_dict = {x_data: rand_x, y_target: rand_y})))
```

上述代码中,设定迭代次数为 100 次,每次选定一组随机的 x 和 y(取值为 1～100 的随机数)导入计算图中并计算相应的损失值。TensorFlow 将自动完成对损失值的计算并在 100 次迭代中调整 A 的值来降低损失值。

输出结果如下所示。

```
第 1 次 A = [0.6222887]
损失值为 [1.6599011]
第 2 次 A = [0.6843269]
损失值为 [1.58904]
第 3 次 A = [0.74623096]
损失值为 [1.5141836]
第 4 次 A = [0.8099702]
损失值为 [1.4054521]
第 5 次 A = [0.87185335]
损失值为 [1.3588741]
.
.
.
损失值为 [0.00031849]
第 95 次 A = [2.020464]
损失值为 [6.4484775e-05]
第 96 次 A = [2.0156667]
损失值为 [9.596348e-06]
第 97 次 A = [1.9776129]
损失值为 [6.9886446e-06]
第 98 次 A = [1.9755591]
损失值为 [2.577901e-06]
第 99 次 A = [2.0272558]
损失值为 [0.01105778]
第 100 次 A = [1.9916381]
损失值为 [1.9565225e-05]
```

从上述结果可以看到,经过不断的优化,算法最后求得的 A 的预测值与目标值已经非常接近了。加载的 matplotlib 图如图 3.6 所示。

从图 3.6 中可以看出,算法在 100 次的迭代中大幅降低了损失值。通常算法的训练需要经过大量的迭代,每次如果将结果一一显示会占用大量不必要的篇幅,后续将通过以下方式缩减输出的结果数量。

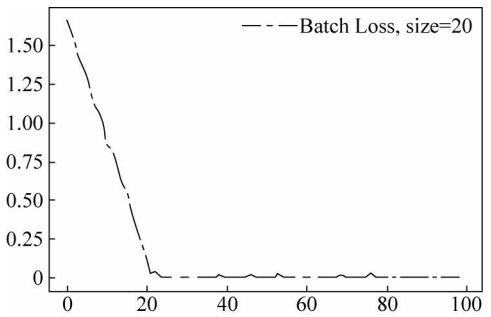

图 3.6　预测值与目标值的距离

```
if (i+1) % 10 == 0:
    print("第" + str(i+1) + "次 A = " + str(sess.run(A)))
    print("损失值为" + str(sess.run(loss, feed_dict = {x_data: rand_x, y_target: rand_y})))
```

上述代码只输出迭代次数能将 10 整除的数据，这可以大幅减少输出结果所占的篇幅。

接下来，对上述示例中的代码编辑流程进行简要概括。

- 生成数据；
- 初始化占位符和变量；
- 创建损失函数；
- 定义优化算法；
- 通过随机数据进行迭代，更新变量 A 的值。

从图 3.6 及输出结果中可以看到在某些时刻，这种标准的梯度下降算法可能会出现卡顿或者收敛速度变缓的现象，尤其是在极值点附近时这种情况尤为明显。下面推荐几种解决这一问题的思路。

(1) 调整优化的步长值，对于变化程度小的变量可以使用较大步长值，对于变化程度大的变量使用较小的步长值。通过 TensorFlow 中的 Adagrad 优化器可以实现这一方法。该优化器所采用的的算法会遍历整个历史迭代的变量梯度，一般将学习率设为 0.01。不过，这个算法可能会导致梯度迅速变为 0。通过 TensorFlow 中的 Adadelta 优化器计算指数加权平均值，旨在消除梯度下降中的摆动。某一维度的导数较大，则指数加权平均值就大；某一维度的导数较小，则其指数加权平均值就小。这样就保证了各维度导数都在一个量级，进而减少了摆动。允许使用一个更大的学习率，从而减少所需要的迭代次数。

(2) 为算法添加“势能”，通过 TensorFlow 中的 Momentum 优化器为损失函数增加势能。该算法的原理是将上一次迭代过程的梯度下降值的导数作为势能加载到下一次迭代过程中。

TensorFlow 实现反向传播的示例完整代码如下所示。

```
1  import numpy as np
2  import tensorflow as tf
3  import matplotlib.pyplot as plt
4  #创建计算图
5  sess = tf.Session()
```

```
#生成数据,100 个均值为 5,标准差为 0.1 的随机数 x 和 100 个值为 10 的目标数 y
x = np.random.normal(5, 0.1, 100)
y = np.repeat(10.0, 100)
#声明占位符
x_data = tf.placeholder(shape = [1], dtype = tf.float32)
y_target = tf.placeholder(shape = [1], dtype = tf.float32)
#声明变量 A
A = tf.Variable(tf.random_normal(shape = [1]))
#创建函数 x×A = target
my_output = tf.multiply(x_data, A)
#指定损失函数为 delta 参数为 0.25 的 Pseudo-Huber 函数,并初始化所有变量
deltal = tf.constant(0.25)
loss = tf.multiply(tf.square(deltal), tf.sqrt(1.0 + tf.square((my_output - y_target)/
deltal)) - 1.0)
init = tf.global_variables_initializer()
sess.run(init)
#声明变量的优化器
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(loss)
#将损失值导入创建的 batch
loss_batch = []
#开始训练算法并将每次迭代数据加载到 matplotlib 图中
for i in range(100) :
    rand_index = np.random.choice(100)
    rand_x = [x[rand_index]]
    rand_y = [y[rand_index]]
    sess.run(train_step, feed_dict = {x_data: rand_x, y_target: rand_y})
    print("第" + str(i+1) + "次 A = " + str(sess.run(A)))
    print("损失值为 " + str(sess.run(loss, feed_dict = {x_data: rand_x, y_target: rand_
    y})))

    temp_loss = sess.run(loss, feed_dict = {x_data: rand_x, y_target: rand_y})
    loss_batch.append(temp_loss)
plt.plot(loss_batch, "g-.", label = "Batch Loss, size = 20")
plt.legend(loc = "upper right", prop = {"size": 11})
plt.show()
```

3.6 TensorFlow 实现随机训练和批量训练

TensorFlow 根据反向传播算法来更新模型变量,在更新过程中可一次只操作一个数据点,也可以一次操作大量数据。如果只针对单个训练样本进行训练,那么所学习的效果可能并不理想;而对大批量样本的训练通常需要高昂的成本。因此,选用合适的训练类型对机器学习算法的收敛至关重要。

3.5.1 节已经对反向传播算法的流程进行了介绍,反向传播必须对一个或多个样本的损失值进行度量,然后将损失值反向传回,本节将要介绍的随机训练可以一次随机抽取一定数量的训练样本和目标数据进行训练。而批量训练则可以一次对指定批量大小的训练数据

和目标数据进行训练，然后取多次训练的平均损失值来计算模型的梯度。

随机训练可以有效降低优化算法陷入局部极小值的情况，但是通常需要多次迭代才能收敛；批量训练的收敛速度快，但是迭代需要消耗大量的计算资源。

接下来将通过随机训练和批量训练两种方法来演示如何实现之前讲过的回归算法。

(1) 首先，导入相应的模块。

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

(2) 声明通过计算图传入的训练数据数量，即对“批量”进行声明。

```
batch_size = 20
```

(3) 声明模型的数据 x、y，占位符和变量 A。

```
x = np.random.normal(1, 0.1, 200)
y = np.repeat(15.0, 200)
x_data = tf.placeholder(shape = [None, 1], dtype = tf.float32)
y_target = tf.placeholder(shape = [None, 1], dtype = tf.float32)
A = tf.Variable(tf.random_normal(shape = [1, 1]))
```

从上述代码中可以看到，占位符具有两个维度，第一个维度为 None，第二个维度表示批量训练中数据量。

(4) 接下来在计算图中添加矩阵乘法操作(矩阵相乘不具有交换律，因此要注意前后矩阵的顺序)。

```
my_output = tf.matmul(x_data, A)
```

(5) 设定损失函数为每个数据点的 L^2 损失平均值，通过 tf.reduce_mean()函数实现求均值。同时初始化所有变量。

```
loss = tf.reduce_mean(tf.square(my_output - y_target))
init = tf.global_variables_initializer()
```

(6) 声明优化器。

```
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
```

(7) 通过循环迭代对算法进行优化，迭代次数为 200 次，每批迭代的数据批量为 20。

```
for i in range(200):
   rand_index = np.random.choice(200, size = batch_size)
   rand_x = np.transpose([x[rand_index]])
   rand_y = np.transpose([y[rand_index]])
   sess.run(train_step, feed_dict = {x_data: rand_x, y_target:rand_y})
```

(8) 为了节省篇幅,此处只输出迭代次数可以整除 20 的结果。

```
if(i+1)%20 == 0:
   print("批量训练 第" + str(i+1) + "次迭代 A 的值为" + str(sess.run(A)))
   temp_loss = sess.run(loss, feed_dict = {x_data: rand_x, y_target: rand_y})
   print("损失值为" + str(temp_loss))
   loss_batch.append(temp_loss)
```

与批量训练相比,随机训练更容易脱离局部极小值,但是收敛速度较慢。接下来演示随机训练的方法,代码如下所示。

```
#随机训练和批量训练
#导入相应模块并开始一个计算图会话
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.python.framework import ops
sess = tf.Session()
#批量训练
#声明批量大小为 20
batch_size = 20
#声明数据并创建占位符
x = np.random.normal(1, 0.1, 200)
y = np.repeat(15.0, 200)
x_data = tf.placeholder(shape = [None, 1], dtype = tf.float32)
y_target = tf.placeholder(shape = [None, 1], dtype = tf.float32)
# 声明变量 A
A = tf.Variable(tf.random_normal(shape = [1, 1]))
#在计算图中添加矩阵乘法操作
my_output = tf.matmul(x_data, A)
#设定损失函数为每个数据的 L2 损失的平均值
loss = tf.reduce_mean(tf.square(my_output - y_target))
#初始化全部变量
init = tf.global_variables_initializer()
sess.run(init)
#声明优化器
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
loss_batch = []
#开始迭代
for i in range(200):
    rand_index = np.random.choice(200, size = batch_size)
    rand_x = np.transpose([x[rand_index]])
    rand_y = np.transpose([y[rand_index]])
    sess.run(train_step, feed_dict = {x_data: rand_x, y_target:rand_y})
    #只输出迭代次数能整除 20 的结果
    if(i+1)%20 == 0:
        print("批量训练 第" + str(i+1) + "次迭代 A 的值为" + str(sess.run(A)))
        temp_loss = sess.run(loss, feed_dict = {x_data: rand_x, y_target: rand_y})
```

```
        print("损失值为" + str(temp_loss))
        loss_batch.append(temp_loss)
#随机训练
#重置计算图
ops.reset_default_graph()
sess = tf.Session()
#声明数据
x = np.random.normal(1, 0.1, 200)
y = np.repeat(15.0, 200)
x_data = tf.placeholder(shape = [1], dtype = tf.float32)
y_target = tf.placeholder(shape = [1], dtype = tf.float32)
#声明变量 A
A = tf.Variable(tf.random_normal(shape = [1]))
#在计算图中加入矩阵乘法操作
my_output = tf.multiply(x_data, A)
#设定损失函数
loss = tf.reduce_mean(tf.square(my_output - y_target))
#初始化所有变量的值
init = tf.global_variables_initializer()
sess.run(init)
#声明优化器
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(loss)
loss_stochastic = []
#开始迭代
for i in range(200):
    rand_index = np.random.choice(200)
    rand_x = [x[rand_index]]
    rand_y = [y[rand_index]]
    sess.run(train_step, feed_dict = {x_data: rand_x, y_target: rand_y})
    #只输出迭代次数能整除 20 的结果
    if (i+1)%20 == 0:
        print("随机训练第" + str(i+1) + "次迭代 A 的值为" + str(sess.run(A)))
        temp_loss = sess.run(loss, feed_dict = {x_data: rand_x, y_target: rand_y})
        print("损失值为" + str(temp_loss))
        loss_stochastic.append(temp_loss)
#通过 matplotlib 绘制运行结果
plt.plot(range(0, 200, 20),loss_batch, "g-.", label = "Batch Loss, size = 20")
plt.plot(range(0, 200, 20),loss_stochastic, "r--", label = "Stochastic Loss")
plt.ylim(0, 100)
plt.legend(loc = "upper right", prop = {"size": 11})
plt.show()
```

输出如下所示。

```
批量训练 第 20 次迭代 A 的值为[[5.0928826]]
损失值为 100.202385
批量训练 第 40 次迭代 A 的值为[[8.36697]]
损失值为 48.93275
```

```
批量训练 第 60 次迭代 A 的值为[[10.545691]]
损失值为 23.615911
批量训练 第 80 次迭代 A 的值为[[11.984723]]
损失值为 12.791118
批量训练 第 100 次迭代 A 的值为[[12.962305]]
损失值为 5.3837404
批量训练 第 120 次迭代 A 的值为[[13.592473]]
损失值为 3.110464
批量训练 第 140 次迭代 A 的值为[[14.097708]]
损失值为 3.225243
批量训练 第 160 次迭代 A 的值为[[14.362068]]
损失值为 2.54338
批量训练 第 180 次迭代 A 的值为[[14.545248]]
损失值为 2.6303203
批量训练 第 200 次迭代 A 的值为[[14.649292]]
损失值为 2.1601777
随机训练第 20 次迭代 A 的值为[[5.016013]]
损失值为 117.44718
随机训练第 40 次迭代 A 的值为[[8.343757]]
损失值为 36.61693
随机训练第 60 次迭代 A 的值为[[10.509306]]
损失值为 24.086134
随机训练第 80 次迭代 A 的值为[[11.9774]]
损失值为 32.032726
随机训练第 100 次迭代 A 的值为[[12.934285]]
损失值为 18.064348
随机训练第 120 次迭代 A 的值为[[13.689663]]
损失值为 7.616622
随机训练第 140 次迭代 A 的值为[[14.190177]]
损失值为 3.8760216
随机训练第 160 次迭代 A 的值为[[14.412206]]
损失值为 7.237827
随机训练第 180 次迭代 A 的值为[[14.612629]]
损失值为 2.97766
随机训练第 200 次迭代 A 的值为[[14.611997]]
损失值为 5.1589394
```

通过 matplotlib 绘制的结果图如图 3.7 所示。

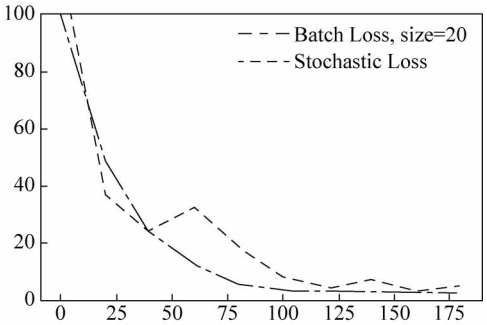

图 3.7　批量训练与随机训练的损失值变化

从图中可以直观地看出，批量训练的损失曲线更加平滑，而随机训练的损失则更容易出现较大波动。

3.7 TensorFlow 创建分类器

本节为大家分享了 TensorFlow 实现创建分类器的具体实例代码，供大家参考，具体内容如下：创建一个 iris(鸢尾花)数据集的分类器。鸢尾花数据集(iris data)是机器学习和数据分析中的经典数据集，是常用的多重变量分析数据集，由 Fisher 教授于 1936 年收集整理完成。iris 数据集包含 150 个数据集，分为 3 类，每类数据 50 个样本，每个样本包含如下 4 种属性：花萼长度、花萼宽度、花瓣长度、花瓣宽度。通过之前所述的 4 个属性来预测鸢尾花卉所属的种类：山鸢尾(setosa)、变色鸢尾(versicolour)和维吉尼亚鸢尾(virginica)。

通过 Python 加载样本数据集时，可以使用 Scikit Learn 的数据集函数，示例代码如下所示(Scikit Learn 可在 Anaconda 的 Environments 中进行下载安装，具体过程不再赘述)。

```
from sklearn import datasets
iris = datasets.load_iris()
print(len(iris.data))
150
print(len(iris.target))
150
print(iris.target[0])
Petal width
[5.1 3.5 1.4 0.2]
print(set(iris.target))
{0, 1, 2}
```

了解了加载样本数据集的方法，接下来介绍如何用 TensorFlow 根据鸢尾花数据集来简单的实现预测一朵“花”是否属于“山鸢尾”的二值分类器。导入 iris 数据集和工具库，相应地对原数据集进行转换。

(1) 导入相关工具库，初始化计算图。

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
import tensorflow as tf
```

(2) 导入鸢尾花数据集，若输入数据是山鸢尾，则令其输出值为 1，否则为 0。由于鸢尾花数据集默认山鸢尾的标记为 0，在此需要将其重置为 1，同时将其他种类置 0。本次训练只使用两种特征：花瓣长度和花瓣宽度，这两个特征在 x-value 的第三列和第四列。代码如下所示。

```
iris = datasets.load_iris()
binary_target = np.array([1. if x == 0 else 0. for x in iris.target])
iris_2d = np.array([[x[2], x[3]] for x in iris.data])
```

(3) 声明批量训练大小、数据占位符和模型变量。由于并不确定数据占位符的维度,可以将第一维度设为 None,代码如下所示。

```
batch_size = 20
x1_data = tf.placeholder(shape = [None, 1], dtype = tf.float32)
x2_data = tf.placeholder(shape = [None, 1], dtype = tf.float32)
y_target = tf.placeholder(shape = [None, 1], dtype = tf.float32)
A = tf.Variable(tf.random_normal(shape = [1, 1]))
b = tf.Variable(tf.random_normal(shape = [1, 1]))
```

(4) 定义线性模型。线性模型的表达式为:$x_2 = x_1 \times A + b$。将数据点代入 $x_2 - x_1 \times A - b$,比较计算结果与 0 的大小,若结果大于 0 则表明数据点在直线上方,结果小于 0 则表明数据点在直线下方。将表达式 $x_2 = x_1 \times A + b$ 传入 Sigmoid 损失函数,然后预测结果。代码如下所示。

```
my_mult = tf.matmul(x2_data, A)
my_add = tf.add(my_mult, b)
my_output = tf.subtract(x1_data, my_add)
```

(5) 引入 Sigmoid 交叉熵损失函数。

```
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(logits = my_output, labels = y_target)
```

(6) 声明优化器方法,最小化损失值。在此选择的学习率为 0.05。

```
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(xentropy)
```

(7) 创建并执行一个变量初始化操作。

```
init = tf.global_variables_initializer()
sess.run(init)
```

(8) 迭代 100 次训练线性模型,传入三种数据:花瓣长度、花瓣宽度和目标变量。

```
for i in range(1000):
  rand_index = np.random.choice(len(iris_2d), size = batch_size)
  rand_x = iris_2d[rand_index]
  rand_x1 = np.array([[x[0]] for x in rand_x])
  rand_x2 = np.array([[x[1]] for x in rand_x])
  rand_y = np.array([[y] for y in binary_target[rand_index]])
  sess.run(train_step, feed_dict = {x1_data: rand_x1, x2_data: rand_x2, y_target: rand_y})
  if (i + 1) % 200 == 0:
    print('Step #' + str(i + 1) + 'A = ' + str(sess.run(A)) + ', b = ' + str(sess.run(b)))
```

完整实现代码如下所示。

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
import tensorflow as tf
from tensorflow.python.framework import ops
ops.reset_default_graph()
# 导入 iris 数据集
# 根据目标数据是否为山鸢尾将其转换成 1 或者 0
# 由于 iris 数据集将山鸢尾标记为 0,将该标记默认对应数值从 0 置为 1,同时把其他物种标记为 0
# 本次训练只使用两种特征: 花瓣长度和花瓣宽度,这两个特征在 x-value 的第三列和第四列
# iris.target = {0, 1, 2}, where '0' is setosa
# iris.data ~ [sepal.width, sepal.length, pedal.width, pedal.length]
iris = datasets.load_iris()
binary_target = np.array([1. if x == 0 else 0. for x in iris.target])
iris_2d = np.array([[x[2], x[3]] for x in iris.data])
# 声明批量训练大小
batch_size = 20
# 初始化计算图
sess = tf.Session()
# 声明数据占位符
x1_data = tf.placeholder(shape = [None, 1], dtype = tf.float32)
x2_data = tf.placeholder(shape = [None, 1], dtype = tf.float32)
y_target = tf.placeholder(shape = [None, 1], dtype = tf.float32)
# 声明模型变量
# Create variables A and b (0 = x1 - A * x2 + b)
A = tf.Variable(tf.random_normal(shape = [1, 1]))
b = tf.Variable(tf.random_normal(shape = [1, 1]))
# 定义线性模型:
# 如果找到的数据点在直线以上,则将数据点代入 x2 - x1 * A - b 计算出的结果大于 0
# 同理找到的数据点在直线以下,则将数据点代入 x2 - x1 * A - b 计算出的结果小于 0
# x1 - A * x2 + b
my_mult = tf.matmul(x2_data, A)
my_add = tf.add(my_mult, b)
my_output = tf.subtract(x1_data, my_add)
# 增加 TensorFlow 的 sigmoid 交叉熵损失函数(cross entropy)
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(logits = my_output, labels = y_target)
# 声明优化器方法
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(xentropy)
# 创建一个变量初始化操作
init = tf.global_variables_initializer()
sess.run(init)
# 运行迭代 1000 次
for i in range(1000):
    rand_index = np.random.choice(len(iris_2d), size = batch_size)
    # rand_x = np.transpose([iris_2d[rand_index]])
    # 传入三种数据: 花瓣长度、花瓣宽度和目标变量
    rand_x = iris_2d[rand_index]
```

```
    rand_x1 = np.array([[x[0]] for x in rand_x])
    rand_x2 = np.array([[x[1]] for x in rand_x])
    #rand_y = np.transpose([binary_target[rand_index]])
    rand_y = np.array([[y] for y in binary_target[rand_index]])
    sess.run(train_step, feed_dict={x1_data: rand_x1, x2_data: rand_x2, y_target: rand_y})
    if (i+1)%200==0:
        print('Step #' + str(i+1) + 'A = ' + str(sess.run(A)) + ', b = ' + str(sess.
        run(b)))
# 绘图
# 获取斜率/截距
# Pull out slope/intercept
[[slope]] = sess.run(A)
[[intercept]] = sess.run(b)
# 创建拟合线
x = np.linspace(0, 3, num=50)
ablineValues = []
for i in x:
ablineValues.append(slope*i+intercept)
# 绘制拟合曲线
setosa_x = [a[1] for i,a in enumerate(iris_2d) if binary_target[i]==1]
setosa_y = [a[0] for i,a in enumerate(iris_2d) if binary_target[i]==1]
non_setosa_x = [a[1] for i,a in enumerate(iris_2d) if binary_target[i]==0]
non_setosa_y = [a[0] for i,a in enumerate(iris_2d) if binary_target[i]==0]
plt.plot(setosa_x, setosa_y, 'rx', ms=10, mew=2, label='setosa')
plt.plot(non_setosa_x, non_setosa_y, 'ro', label='Non-setosa')
plt.plot(x, ablineValues, 'b-')
plt.xlim([0.0, 2.7])
plt.ylim([0.0, 7.1])
plt.suptitle('Linear Separator For I.setosa', fontsize=20)
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.legend(loc='lower right')
plt.show()
```

输出如下所示。

```
Step #200 A = [[ 8.70572948]], b = [[-3.46638322]]
Step #400 A = [[ 10.21302414]], b = [[-4.720438]]
Step #600 A = [[ 11.11844635]], b = [[-5.53361702]]
Step #800 A = [[ 11.86427212]], b = [[-6.0110755]]
Step #1000 A = [[ 12.49524498]], b = [[-6.29990339]]
```

如图 3.8 所示。

上述代码的目的是根据花瓣的长度和宽度这两个数据来区分山鸢尾和其他种类的花，绘制所有的数据点和拟合结果，并预期找到一条直线将属于山鸢尾的数据点与其他数据点分开。

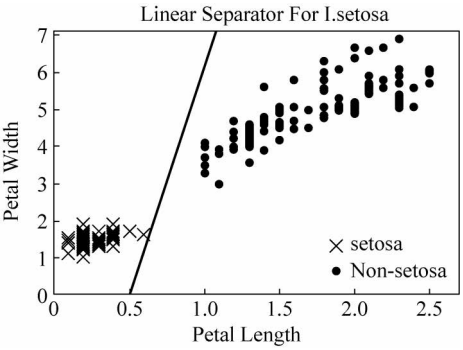

图 3.8　运行结果

3.8　TensorFlow 实现模型评估

模型训练好以后，对模型训练效果的评估至关重要，每个模型都有对应的模型评估方式。在 TensorFlow 中，将模型评估加入计算图中，然后在模型训练完后对模型进行评估。

在模型的训练过程中，模型评估可以根据模型采用的算法，给出相应的提示信息，从而方便对模型进行调试，提高模型的学习效果。本节将演示在回归算法和分类算法中进行模型评估的方法。

3.8.1　模型评估方法

通常，评估模型需要有训练数据集和测试数据集来进行对照，从而判断模型的训练效果。有时，甚至需要专门设计一个验证数据集来进行模型的评估。模型评估需要大批量数据点的支持，因此在批量训练任务中，可以直接重用模型来预测大批量数据点。然而，在随机训练任务中，由于没有现成的大批量数据点，此时需要创建单独的评估器来处理批量数据点。

在模型评估中需要注意是否保持模型的输出结果与模型的评估结果的数据类型一致。如果在损失函数中对模型的输出结果进行了转化，则模型评估时也需要进行相应的转化，从而保持评估结果的准确性。

通过之前所学的知识可以知道，分类算法模型是基于数值型输入数据进行分类的，目标值是由“0”和“1”组成的序列。可以通过度量预测值与目标值之间的距离来评估该类模型。在分类算法模型中，损失函数通常无法直接反应模型的训练效果，通常通过模型的分类预测准确率来判断模型的训练效果。需要注意的是，不管模型的预测结果是否准确，都需要测试算法模型。

在模型评估中，需要同时对训练数据和测试数据的预测准确率进行评估，以检测模型是否出现了过拟合。

3.8.2　模型评估工作原理及实现

(1) 加载相应工具库，创建计算图、数据、变量以及占位符。在创建了上述数据之后，可

以将数据随机分为训练数据集和测试数据集，这一点对于模型评估至关重要。在训练数据和测试数据上评估模型也可以判断模型是否出现过拟合。


```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
sess = tf.Session()
x_vals = np.random.normal(1, 0.1, 100)
y_vals = np.repeat(10., 100)
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
batch_size = 25
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
A = tf.Variable(tf.random_normal(shape=[1,1]))
```

（2）声明模型的损失函数和优化算法并初始化模型中的所有变量。

```
my_output = tf.matmul(x_data, A)
loss = tf.reduce_mean(tf.square(my_output - y_target))
init = tf.initialize_all_variables()
sess.run(init)
my_opt = tf.train.GradientDescentOptimizer(0.02)
train_step = my_opt.minimize(loss)
```

（3）开始训练。

```
for i in range(100):
rand_index = np.random.choice(len(x_vals_train), size=batch_ size)
rand_x = np.transpose([x_vals_train[rand_index]])
rand_y = np.transpose([y_vals_train[rand_index]])
sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
   if (i+1)%25==0:
      print('Step #' + str(i+1) + 'A = ' + str(sess.run(A)))
      print('Loss = ' + str(sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})))
```

添加了输出 matplotlib 图的完整代码如下所示。

```
1   # TensorFlow 模型评估
2   import matplotlib.pyplot as plt
3   import numpy as np
4   import tensorflow as tf
5   from tensorflow.python.framework import ops
6   ops.reset_default_graph()
```

```
# 创建计算图
sess = tf.Session()
# 回归例子
# We will create sample data as follows:
# x-data: 100 random samples from a normal ~ N(1, 0.1)
# target: 100 values of the value 10.
# We will fit the model:
# x-data * A = target
# 理论上, A = 10
# 声明批量大小
batch_size = 25
# 创建数据集
x_vals = np.random.normal(1, 0.1, 100)
y_vals = np.repeat(10., 100)
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
# 八二分训练/测试数据 train/test = 80%/20%
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
# 创建变量 (one model parameter = A)
A = tf.Variable(tf.random_normal(shape=[1,1]))
# 增加操作到计算图
my_output = tf.matmul(x_data, A)
# 增加 L2 损失函数到计算图
loss = tf.reduce_mean(tf.square(my_output - y_target))
# 创建优化器
my_opt = tf.train.GradientDescentOptimizer(0.02)
train_step = my_opt.minimize(loss)
# 初始化变量
init = tf.global_variables_initializer()
sess.run(init)
# 迭代运行
# 如果在损失函数中使用的模型输出结果经过转换操作,例如,sigmoid_cross_entropy_with_
logits()函数
# 为了精确计算预测结果,在模型评估中也要进行转换操作
for i in range(100):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
    rand_x = np.transpose([x_vals_train[rand_index]])
    rand_y = np.transpose([y_vals_train[rand_index]])
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%25==0:
        print('Step #' + str(i+1) + 'A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})))
# 评估准确率(loss)
mse_test = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_test]), y_target:
np.transpose([y_vals_test])})
```

```
mse_train = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_train]), y_target: np.transpose([y_vals_train])})
print('MSE on test:' + str(np.round(mse_test, 2)))
print('MSE on train:' + str(np.round(mse_train, 2)))
# 分类算法案例
# We will create sample data as follows:
# x-data: sample 50 random values from a normal = N(-1, 1)
#        + sample 50 random values from a normal = N(1, 1)
# target: 50 values of 0 + 50 values of 1.
#        These are essentially 100 values of the corresponding output index
# We will fit the binary classification model:
# If sigmoid(x+A) < 0.5 -> 0 else 1
# Theoretically, A should be -(mean1 + mean2)/2
# 重置计算图
ops.reset_default_graph()
# 加载计算图
sess = tf.Session()
# 声明批量大小
batch_size = 25
# 创建数据集
x_vals = np.concatenate((np.random.normal(-1, 1, 50), np.random.normal(2, 1, 50)))
y_vals = np.concatenate((np.repeat(0., 50), np.repeat(1., 50)))
x_data = tf.placeholder(shape=[1, None], dtype=tf.float32)
y_target = tf.placeholder(shape=[1, None], dtype=tf.float32)
# 分割数据集 train/test = 80%/20%
train_indices = np.random.choice(len(x_vals), round(len(x_vals)*0.8), replace=False)
test_indices = np.array(list(set(range(len(x_vals))) - set(train_indices)))
x_vals_train = x_vals[train_indices]
x_vals_test = x_vals[test_indices]
y_vals_train = y_vals[train_indices]
y_vals_test = y_vals[test_indices]
# 创建变量 (one model parameter = A)
A = tf.Variable(tf.random_normal(mean=10, shape=[1]))
# Add operation to graph
# Want to create the operation sigmoid(x + A)
# Note, the sigmoid() part is in the loss function
my_output = tf.add(x_data, A)
# 增加分类损失函数 (cross entropy)
xentropy = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=my_output, labels=y_target))
# 创建优化器
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(xentropy)
# 初始化变量
init = tf.global_variables_initializer()
sess.run(init)
# 运行迭代
for i in range(1800):
    rand_index = np.random.choice(len(x_vals_train), size=batch_size)
```

```
    rand_x = [x_vals_train[rand_index]]
    rand_y = [y_vals_train[rand_index]]
    sess.run(train_step, feed_dict = {x_data: rand_x, y_target: rand_y})
    if (i + 1) % 200 == 0:
        print('Step # ' + str(i + 1) + 'A = ' + str(sess.run(A)))
        print('Loss = ' + str(sess.run(xentropy, feed_dict = {x_data: rand_x, y_target:
 rand_y})))
# 评估预测
# 用 squeeze()函数封装预测操作,使得预测值和目标值有相同的维度
y_prediction = tf.squeeze(tf.round(tf.nn.sigmoid(tf.add(x_data, A))))
# 通过 equal()函数检测输出是否相等
# 把得到的 true 或 false 的 boolean 型张量转化成 float32 型
# 再对其取平均值,得到一个准确度值
correct_prediction = tf.equal(y_prediction, y_target)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
acc_value_test = sess.run(accuracy, feed_dict = {x_data: [x_vals_test], y_target: [y_
vals_test]})
acc_value_train = sess.run(accuracy, feed_dict = {x_data: [x_vals_train], y_target: [y_
vals_train]})
print('Accuracy on train set: ' + str(acc_value_train))
print('Accuracy on test set: ' + str(acc_value_test))
# 绘制分类结果
A_result = -sess.run(A)
bins = np.linspace(-5, 5, 50)
plt.hist(x_vals[0:50], bins, alpha = 0.5, label = 'N(-1,1)', color = 'white')
plt.hist(x_vals[50:100], bins[0:50], alpha = 0.5, label = 'N(2,1)', color = 'red')
plt.plot((A_result, A_result), (0, 8), 'k--', linewidth = 3, label = 'A = '+ str(np.round
(A_result, 2)))
plt.legend(loc = 'upper right')
plt.title('Binary Classifier, Accuracy = ' + str(np.round(acc_value_test, 2)))
plt.show()
```

输出如下所示。

```
Step #25 A = [[ 5.79096079]]
Loss = 16.8725
Step #50 A = [[ 8.36085415]]
Loss = 3.60671
Step #75 A = [[ 9.26366138]]
Loss = 1.05438
Step #100 A = [[ 9.58914948]]
Loss = 1.39841
MSE on test:1.04
MSE on train:1.13
Step #200 A = [ 5.83126402]
Loss = 1.9799
Step #400 A = [ 1.64923656]
Loss = 0.678205
Step #600 A = [ 0.12520729]
```

```
Loss = 0.218827
Step #800 A = [-0.21780498]
Loss = 0.223919
Step #1000 A = [-0.31613481]
Loss = 0.234474
Step #1200 A = [-0.33259964]
Loss = 0.237227
Step #1400 A = [-0.28847221]
Loss = 0.345202
Step #1600 A = [-0.30949864]
Loss = 0.312794
Step #1800 A = [-0.33211425]
Loss = 0.277342
Accuracy on train set: 0.9625
Accuracy on test set: 1.0
```

通过 matplotlib 绘制的图形如图 3.9 所示。

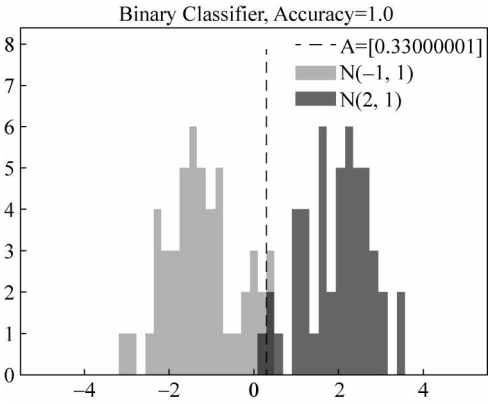

图 3.9　结果图

3.9　本 章 小 结

本章详细讲解了使用 TensorFlow 进行神经网络的进阶操作的各个环节，这些都是在使用神经网络模型时需要考虑的主要问题，从神经网络的嵌入层和嵌入多层、损失函数的选择、反向传播的实现、实现随机训练与批量训练，以及实现模型评估进行了讲解。第 4 章将对 TensorFlow 中的线性回归知识进行讲解。

3.10　习　　题

1. 填空题

(1) TensorFlow 是一个通过_____的形式来表述计算的编程系统，其中每一个计算都是图上的一个节点，而节点之间的边描述了计算之间的_____。

(2) _____用于衡量模型的输出值与预测值之间的差距。

(3) TensorFlow 可以在维持操作状态的同时，基于_____自动对模型的数据进行更新。

(4) _____可以一次性对指定批量大小的训练训练数据和目标数据进行训练。

(5) 通常，评估模型需要有_____和_____来进行对照，从而判断模型的训练效果。

2. 选择题

(1) TensorFlow 提供了自定义生成新的计算图的方法，通过(　　)函数可以生成新的计算图。

A. tf.get_variable()　　B. tf.Graph()

C. tf.placeholder()　　D. tf.Session()

(2) 在 TensorFlow 中，需要先声明数据形状，然后预估经过相应操作后返回值的形状来创建对应的(　　)。

A. 张量　　B. 计算图　　C. 占位符　　D. batch

(3) L^1 正则损失函数也被称作(　　)。

A. 欧几里得损失函数　　B. 绝对值损失函数

C. 铰链损失函数　　D. 交叉熵损失函数

(4) 用(　　)来表示迭代中所选取的数据可以降低迭代中生成的节点数量，提升学习效率。

A. 占位符　　B. 张量　　C. 计算图　　D. batch

(5) 通过(　　)优化器，可以降低梯度下降中的摆动，允许模型使用更大的学习率进行训练，从而减少完成训练所需要的迭代次数。

A. Adagrad　　B. Adadelta

C. GradientDescent　　D. SGD

3. 思考题

简要分析损失函数在反向传播算法中的意义。