1

TensorFlow 的基础知识

本章的主要内容是深度学习框架 TensorFlow 的基础知识。深度学习在模式识别方面做得很好，特别是在图像、声音、语音、语言和时间序列数据方面。在深度学习的帮助下，你可以进行分类、预测、集群和提取特征。幸运的是，在 2015 年 11 月，谷歌发布了 TensorFlow，它已被用于谷歌的大部分产品，如谷歌搜索、垃圾邮件检测、语音识别、谷歌助理、Google Now 和谷歌照片。

TensorFlow 有一个独特的能力，可以进行部分子图计算，以便在划分神经网络的帮助下进行分布式训练。换句话说，TensorFlow 允许模型并行化和数据并行化。TensorFlow 提供了多个 API。

本章结构如下：

- 张量
- 处理图
- 常量、占位符和变量
- 创建张量
- 在矩阵上工作
- 激活函数
- 创建张量
- 损失函数
- 优化器
- 衡量标准

1.1. 张量

在你进入 TensorFlow 库之前，让我们熟悉一下 TensorFlow 中的基本数据单位。张量是一个数学对象，是标量、向量和矩阵的泛化。张量可以被表示为一个多维数组。一个零等级（顺序）的张量只不过是一个标量。一个矢量/数组是一个秩为 1 的张量，而一个矩阵是一个秩为 2 的张量。简而言之，张量可以被认为是一个 n 维的数组。下面是一些张量的例子。- 5：这是一个等级为 0 的张量；这是一个标量，形状为 [ ]。- [2.,5., 3.]: 这是一个等级 1 的张量；这是一个形状为 [3] 的

向量。- [[1., 2., 7.], [3., 5., 4.]]: 这是一个等级 2 的张量；它是一个形状为 [2，3] 的矩阵。- [[[1., 2., 3.]], [[7., 8., 9.]]]: 这是一个等级 3 的张量，形状为 [2，1，3]。

1.2. 图处理

让我们了解一下 TensorFlow 是如何工作的。

- 它的程序通常被结构化为构建阶段和执行阶段。

- 构造阶段组装了一个有节点（OPS/操作）和边（张量）的图。

- 执行阶段使用会话来执行图中的 OPS（操作）。

- 最简单的操作是一个常数，它不需要输入，但将输出传递给其他进行计算的操作。

- 一个操作的例子是乘法 (或加法或减法，需要两个矩阵作为输入并传递一个矩阵作为输出)。

- TensorFlow 库有一个默认的图，OPS 构造器向其添加节点。

所以，TensorFlow 程序的结构有两个阶段，在这里显示。

Construction Phase Assembles a graph Constructs nodes(ops) and edges(tensors) of the computational graph

Execution Phase

Uses a session to execute operations Repeatedly execute a set of training operations

计算图是一系列的 TensorFlow 操作排列成的节点图。让我们来看看 TensorFlow 与 Numpy 的对比。在 Numpy 中，如果你打算将两个矩阵相乘，你可以创建矩阵并将其相乘。但是在 TensorFlow 中，你设置了一个图（一个默认的图，除非你创建另一个图）。接下来，你需要创建变量、占位符和常量值，然后创建会话并初始化变量。最后，你把这些数据输入占位符，以便调用任何动作。

为了实际评估节点，你必须在一个会话中运行计算图。一个会话封装了 TensorFlow 运行时的控制和状态。下面的代码创建了一个会话对象。

$$sess = tf.Session() \tag{1.1}$$

然后，它调用其运行方法来运行足够的计算图以评估节点 1 和节点 2。计算图定义了计算。它既不计算任何东西，也不持有任何值。它是为了定义代码中提到的操作。一个默认的图被创建。所以，你不需要创建它，除非你想为多种目的创建图。会话允许你执行图形或图形的一部分。它为执行分配资源（在一个或多个 CPU 或 GPU 上）。它持有中间结果和变量的实际值。在 TensorFlow 中创建的变量的值，只在一个会话中有效。如果你试图在之后的第二个会话中查询该值，TensorFlow 将引发一个错误，因为该变量在那里没有被初始化。为了运行任何操作，你需要为该图创建一个会话。该会话也将分配内存来存储变量的当前值

下面是演示的代码。

Listing 1.1: 演示代码 1

```
import tensorflow as tf
sess = tf. Session()
```

Listing 1.2: 演示代码 2

```
# Creating a new graph (not default)
myGraph = tf.Graph()
with myGraph. as_default():
variable = tf. Variable(30, name-"navin")
```

```
initialize = tf.global_variables_initializer()
```

Listing 1.3: 演示代码 3

```
with tf.Session(graph=myGraph) as sess:
sess run(initialize)
print(sess. run (variable))
```

Listing 1.4: 演示代码 3

```
# Tensorboard can be used. It is optionalmy_
# Output graph can be seen on tensorboard
import os
merged - tf.summary.merge_all(key='summaries')
if not os.path. exists ('tenosrboard_logs/°):
os. makedirs(*tenosrboard_logs/')
my writer - tf. summary. FileNriter(*/home/manaswi/tenosrboard_logs/*, sess.graph)
def TB(cleanup=False):
import webbrowser
webbrowser. open(*http: //127.0.1.1:6006°)
Itensorboard--logdir='/home/manaswi/tenosrboard_logs
if cleanup:
Irm -R tensorboard logs/
TB(1)
# Launch graph on tensorborad on your browser
```

1.3. 常量、占位符和变量

TensorFlow 程序使用张量数据结构来表示所有的数据–只有张量在计算图的操作之间被传递。你可以把 TensorFlow 的张量看作是一个 n 维的数组或列表。一个张量有一个静态类型，一个等级，和一个形状。这里的图产生一个恒定的结果。变量在图的整个执行过程中保持状态。

一般来说，你必须在深度学习中处理许多图像，所以你必须为每个图像放置像素值，并在所有图像上不断迭代。

为了训练模型，你需要能够修改图来调整一些对象，如权重和偏置。简而言之，变量使你能够向图中添加可训练的参数。它们由一个类型和初始值构成。

让我们在 TensorFlow 中创建一个常量并打印它。

Listing 1.5: 演示代码

```
import tensorflow as tf
tf.constant(12, dtype= 'float32 ‘)
sess - tf.Session()
print (sess.run(x))
```

下面是对前面代码的简单解释。

1. 导入 tensorflow 模块，并称其为 tf。
2. 创建一个常量值 (x)，并给它分配数值 12。
3. 创建一个用于计算数值的会话。
4. 只运行变量 x 并打印出其当前值。

前两个步骤属于构造阶段，后两个步骤属于执行阶段。我现在将讨论 TensorFlow 的构建和执行阶段。

你可以用另一种方式重写前面的代码，代码如下：

Listing 1.6: 演示代码

```
import tensorflow as tf
x = tf.constant(12, dtype='float32 ‘)
with tf. Session() as sess:
        print(sess. run(x))
```

如何创建一个变量并初始化它。代码如下：

Listing 1.7: 演示代码

```
import tensorflow as tf
x = tf.constant(12, dtype- float32°)
y = tf.Varlable(x411)
model
rarlables.Initlalizer()
wlth tf.Session() as sess:
Sess.run (model)
print(sess. run(y))
```

下面是对前面代码的解释。

1. 导入 tensorflow 模块并调用 tf。
2. 创建一个名为 x 的常量值，并赋予其数值为 12。
3. 创建一个名为 y 的变量，并将其定义为方程 12+11。
4. 用 tf.global_variables_initializer() 初始化这些变量。
5. 创建一个会话来计算数值。
6. 6. 运行步骤 4 中创建的模型。
7. 只运行变量 y 并打印出其当前值。

下面是完整代码：

Listing 1.8: 演示代码

```
import tensorflow as tf
x = tf. constant([14, 23, 40, 30])
y = tf. Variable(x*2 + 100)
model - tf.global_variables_initializer()
with tf. Session() as sess:
        sess.run( model)
        print(sess.run(y))
```

1.4. 占位符

占位符是一个变量，你可以在以后的时间里向其输入一些东西。它是为了接受外部输入。占位符可以有一个或多个维度，用于存储 n 维数组。

Listing 1.9: Placeholders 示例代码

```
import tensorflow as tf
x = tf.placeholder("float", None)
y = x*10 + 500
with tf. Session() as sess:
        placeX - sess.run(y, feed _dict-{x: [0, 5, 15, 25]})
        print(placeX)
```

下面是对前面代码的解释。

1. 导入 tensorflow 模块并调用 tf。

2. 创建一个名为 x 的占位符，提到 float 类型。3. 3. 创建一个名为 y 的张量，它是 x 乘以 10 并加上 500 的操作。注意，x 的任何初始值都没有定义。

4. 4. 创建一个会话来计算这些值。

5. 在 feed_dict 中定义 x 的值，以便运行 y。

6. 打印出它的值。

在下面的例子中，你创建了一个 2×4 的矩阵（一个二维数组）用于存储一些数字。然后，你使用与之前相同的操作，进行逐元乘以 10 和加 1 的操作。占位符的第一维是无，这意味着允许任何行数。

用一个二维数组来代替一维数组，代码如下：

Listing 1.10: 二维数组来代替一维数组的示例代码

```
import tensorflow as tf
x = tf.placeholder ("float", [None, 4])
y = x*10 + 1
with tf. Session() as sess:
        dataX = [[12, 2, 0, -2],
                        [14, 4, 1, 0]
        placeX = sess.run(y, feed _dict-{x: dataX})
        print(placeX)
```

这是一个 2×4 的矩阵。因此，如果你用 2 代替 none，你可以看到同样的输出。

Listing 1.11: 用 2 代替 none 的示例代码

```
import tensorflow as tf
x = tf.placeholder ("float", [2, 4])
y = x*10 + 1
with tf. Session() as sess:
        dataX = [[12, 2, 0, -2],
                        [14, 4, 1, 0]
        placeX = sess.run(y, feed _dict-{x: dataX})
        print(placeX)
```

但如果你创建一个 [3, 4] 形状的占位符（注意，你将在以后的时间里输入一个 2×4 的矩阵），会出现错误，如图所示。

图 1.1: 出错截图

Listing 1.12: Evaluate how the model does on the test set

```
################## What happens in a linear model ########## # Weight and Bias as Variables as they
    are to be tuned
W = tf.Variable([2], dtype=tf.float32)
b = tf.Variable([3], dtype=tf.float32)
# Training dataset that will be fed while training as Placeholders x = tf.placeholder(tf.float32)
# Linear Model
y=W*x+ b
```

常量在你调用 tf.constant 时被初始化，并且它们的值永远不会改变。相比之下，当你调用 tf.Variable 时，变量不会被初始化。为了初始化 TensorFlow 程序中的所有变量，你必须声明调用一个特殊的操作，如下所示。

Listing 1.13: 声明调用一个特殊的操作

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

重要的是要认识到 init 是 TensorFlow 子图的一个句柄，它初始化了所有的全局变量。在你调用 sess.run 之前，这些变量是未初始化的。

1.5. 创建张量

一个图像是一个三阶的张量，其尺寸属于高度、宽度和通道数（红、蓝、绿）。在这里你可以看到图像是如何被转换成张量的。

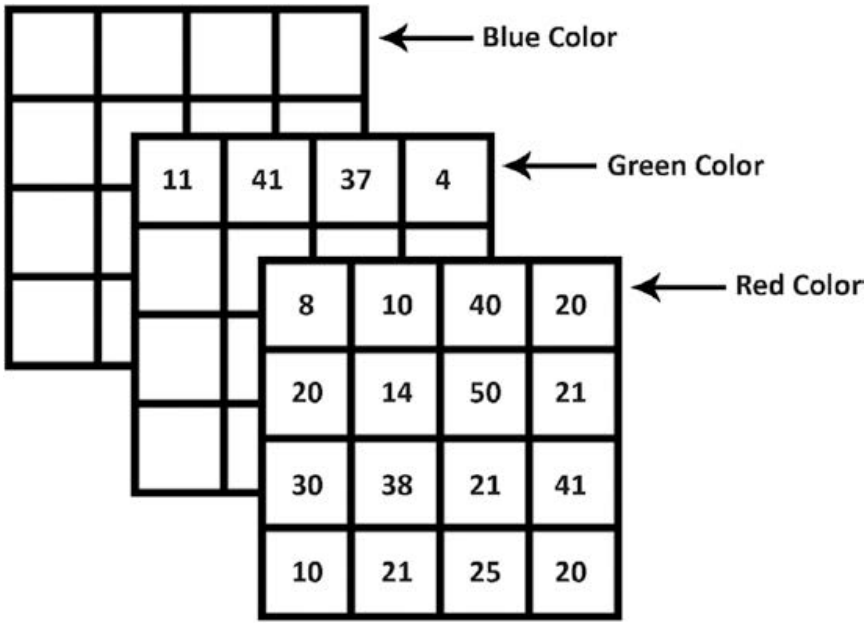

图 1.2: 图像是如何被转换成张量的

图 1.3: 图像是如何被转换成张量的

Listing 1.14: 图像是如何被转换成张量的示例代码

```
image = tf.image.decode_jpeg(tf.read file("./Desktop/image. jpg"), channels=3)
sess = tf. InteractiveSession()
print(sess. run(tf.shape(image)))
```

Listing 1.15: 图像是如何被转换成张量的示例代码

```
print(sess.run (image[10:15,0:4,1]))
```

你可以生成各种类型的张量，如固定张量、随机张量和连续张量。

1.5.1. 适配张量

适配张量的代码:

Listing 1.16: 适配张量的代码 1

```
import tensorflow as tf
sess = tf.Session()
A = tf.zeros([2,3])
print(sess.run (A))
```

Listing 1.17: 适配张量的代码 2

```
B = tf.ones([4,3])
print (sess. run (B))
```

Listing 1.18: 适配张量的代码 3

```
import tensorflow as tf
Sess =  tf.Session()
A = tf.zeros([2,3])
print(sess.run(A))
```

Listing 1.19: 适配张量的代码 4

```
B = tf.ones([4,3])
print (sess.run(B))
```

tf.fill 创建了一个具有唯一数字的形状（2×3）的张量。

Listing 1.20: tf.fill 创建了一个具有唯一数字的形状（2×3）的张量代码

```
C = tf.fill([2,3], 13)
print (sess. run (C))
```

tf.diag 创建一个具有指定对角线元素的对角线矩阵。

Listing 1.21: tf.diag 创建一个具有指定对角线元素的对角线矩阵示例代码

```
D = tf.diag([4,-3, 2])
print(sess. run (D))
```

tf.constant 创建一个常数张量

Listing 1.22: tf.constant 创建一个常数张量示例代码

```
E = tf.constant([5,2,4,2])
print (sess.run(E))
```

1.5.2. 序列张量

tf.range 创建一个从指定值开始并有指定增量的数字序列。

Listing 1.23: tf.range 创建一个从指定值开始并有指定增量的数字序列示例代码。

```
G = tf.range(start=6, limit=45, delta=3)
print(sess.run(G))
```

tf.linspace 创建一个均匀间隔的数值序列。

Listing 1.24: tf.linspace 创建一个均匀间隔的数值序列示例代码

```
H = tf.linspace(10.0, 92.0, 5)
print (sess.run(H))
```

1.5.3. 随机张量

tf.random_uniform 在一个范围内从均匀分布中产生随机值。

Listing 1.25: tf.random_uniform 在一个范围内从均匀分布中产生随机值示例代码

```
R1 = tf.random uniform([2,3], minval=e, maxval=4)
print(sess. run(R1))
```

tf.random_normal 从正态分布中生成具有指定平均值和标准偏差的随机值。

Listing 1.26: tf.random_normal 从正态分布中生成具有指定平均值和标准偏差的随机值示例代码 1。

```
R2 = tf.random_ normal(2,3], mean=5, stddev=4)
print (sess. run (R2))
```

Listing 1.27: tf.random_normal 从正态分布中生成具有指定平均值和标准偏差的随机值示例代码 2

```
print (sess.run(tf.diag([3, -2, 4])))
```

Listing 1.28: tf.random_normal 从正态分布中生成具有指定平均值和标准偏差的随机值示例代码 3

```
R3 = tf.random_ shuffle(tf.diag([3, -2, 4]))
print (sess. run(R3))
```

Listing 1.29: tf.random_normal 从正态分布中生成具有指定平均值和标准偏差的随机值示例代码 4

```
R4 = tf. random_crop(tf.diag([3, -2, 4]), [3,2])
print (sess. run (R4))
```

输出结果：

Listing 1.30: 输出结果 1

```
print(sess. run(tf.zeros([2,4])))
```

Listing 1.31: 输出结果 2

```
print(sess. run(tf.diag([3, 1,5, -2])))
```

Listing 1.32: 输出结果 3

```
print(sess. run(tf.range(start=4, limit=16, delta=2)))
```

以张量替换后的输出：

Listing 1.33: 替换后的输出 1

```
print(sess.run(tf. zeros([2, 4])))
```

Listing 1.34: 替换后的输出 2

```
print(sess.run(tf.diag([3,1,5,-2])))
```

Listing 1.35: 替换后的输出 3

```
print(sess.run(tf.range(start=4, limit=16, delta=2)))
```

1.6. 矩阵控制

一旦你能自如地创建张量，你就能享受到矩阵（二维张量）的控制。

Listing 1.36: 矩阵操作 1

```
import tensorflow as tf
import numpy as np
sess = tf.Session()
A = tf.random_uniform([3, 2])
B = tf.f111([2,4], 3.5)
C = tf.Pandom _normal([3, 4])
```

Listing 1.37: 矩阵操作 2

```
print(sess.run (A))
```

Listing 1.38: 矩阵操作 3

```
print(sess.run (B))
```

Listing 1.39: 矩阵操作 4

```
print(sess.run(tf.matmul(A,B))) # Multiplication Of Matrices
```

Listing 1.40: 矩阵操作 5

```
print(sess.run(tf.matmul(A, B) + C)) # Multiplication & addition
```

1.7. 启用功能

激活函数的概念来自对人脑中神经元如何工作的分析（见图1.4）。神经元在超过一定的阈值时就会变得活跃，更好地称为激活电位。在大多数情况下，它也试图把输出放到一个小范围内。

Sigmoid、双曲正切（tanh）、ReLU 和 ELU 是最流行的激活函数。

让我们来看看这些流行的激活函数。

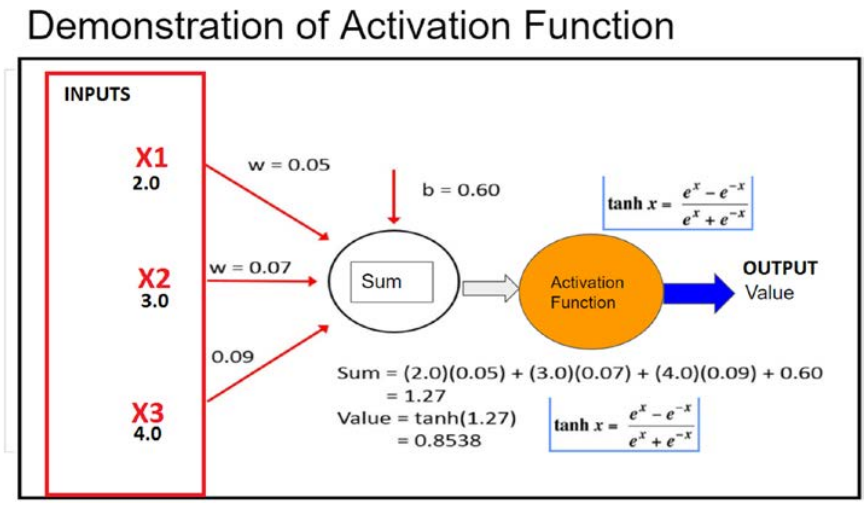

图 1.4: 激活函数的概念来自对人脑中神经元如何工作的分析示意图

正切双曲和双曲

图1.5显示了正切双曲和双曲的激活函数。

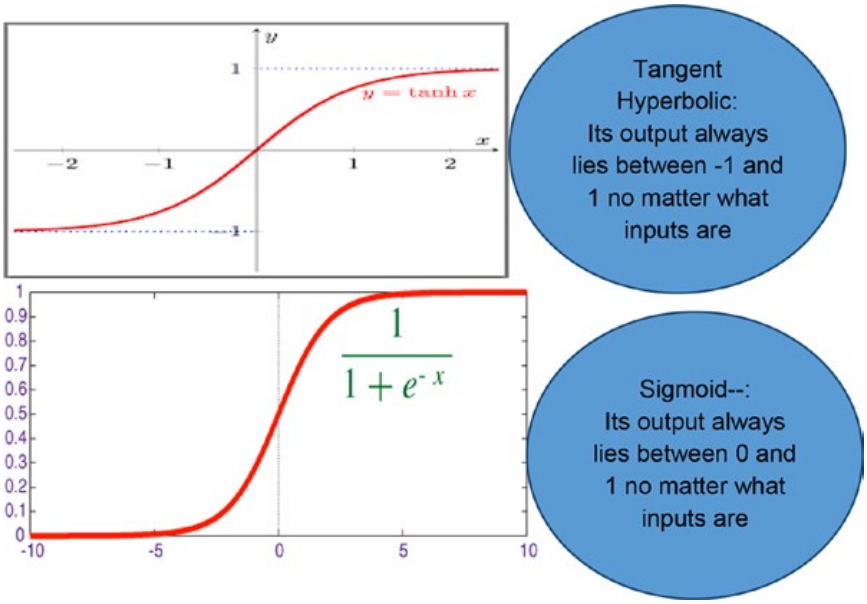

图 1.5: 正切双曲和双曲的激活函数

代码如下：

Listing 1.41: 正切双曲示例代码

```
E = tf.nn.tanh([10,2,1,0.5,0, -0.5,-1.,-2., -10.])
print(sess. run(E))
```

Listing 1.42: 双曲的激活函数示例代码

```
J = tf.nn.sigmoid([10,2,1,0.5,0,-0.5, -1., -2.,-10.])
print(sess.run(J))
```

1.7.1. ReLU 和 ELU

图1.6显示了 ReLU 和 ELU 的功能。

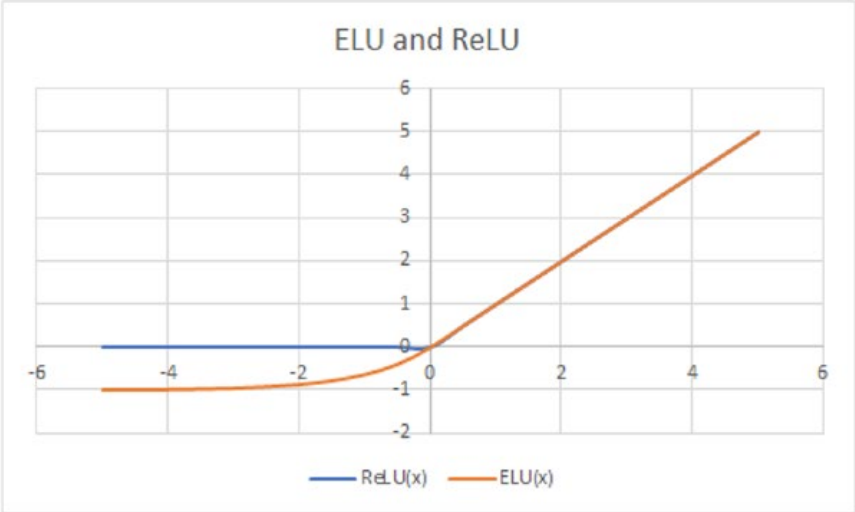

图 1.6: ReLU 和 ELU 的功能

代码如下：

Listing 1.43: ReLU 和 ELU 的功能代码

```
A = tf.nn.relu([-2,1,-3,13])
print (sess.run(A))
```

1.7.2. ReLU6

ReLU6 与 ReLU 类似，只是输出不能超过六次。

Listing 1.44: ReLU6 代码 1

```
B = tf.nn.relu6([-2, 1, -3, 13])
print(sess. run(B))
```

Listing 1.45: ReLU6 代码 2

```
C = tf.nn.relu([[-2, 1,-3], [10, -16, -5]])
print (sess.run(C))
```

请注意，tanh 是一个重新缩放的逻辑 sigmoid 函数。

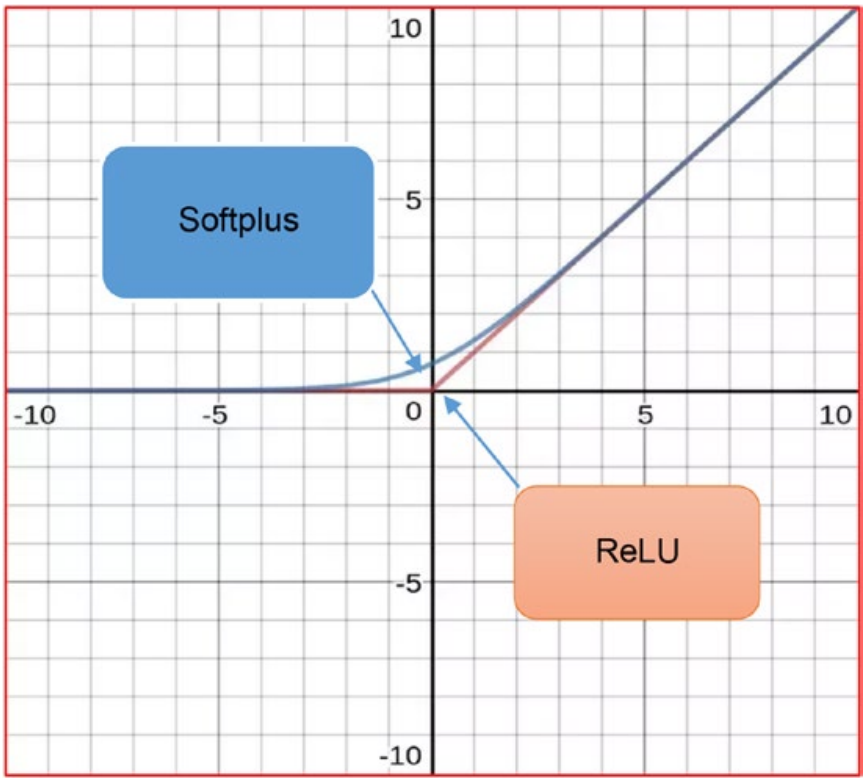

图 **1.7:** tanh 是一个重新缩放的逻辑 sigmoid 函数

Listing 1.46: 示例代码 1

```
K = tf.nn.relu([10,2,1,0.5,0,-0.5,-1.,-2.,-10.])
print(sess. run(K))
```

Listing 1.47: 示例代码 2

```
M = tf.nn. softplus([10,2, 1,0.5,0,-0.5,-1.,-2., -10.])
print(sess.run(M))
```

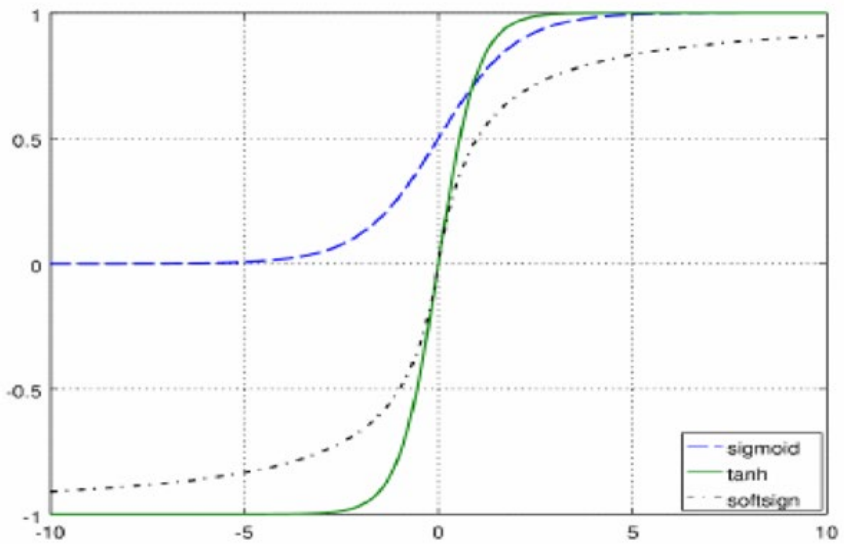

图 1.8: 三种函数的对比

Listing 1.48: 示例代码 3

```
H = tf.nn.elu([10,2,1,0.5,0,-0.5,-1.,-2.,-10.])
print(sess.run (H))
```

Listing 1.49: 示例代码 4

```
I= tf.nn.relu6( [10, 2, 1,0.5,0,-0.5,-1.,-2.,-10.])
print (sess. run(I))
```

Listing 1.50: 示例代码 5

```
G = tf.nn.softsign([10,2,1,0.5,0, -0.5, -1.,-2.,-10.])
print(sess. run(G))
```

Listing 1.51: 示例代码 6

```
F = tf.nn.softplus([10,2,1,0.5,0,-0.5, -1.,-2., -10.])
print(sess. run(F))
```

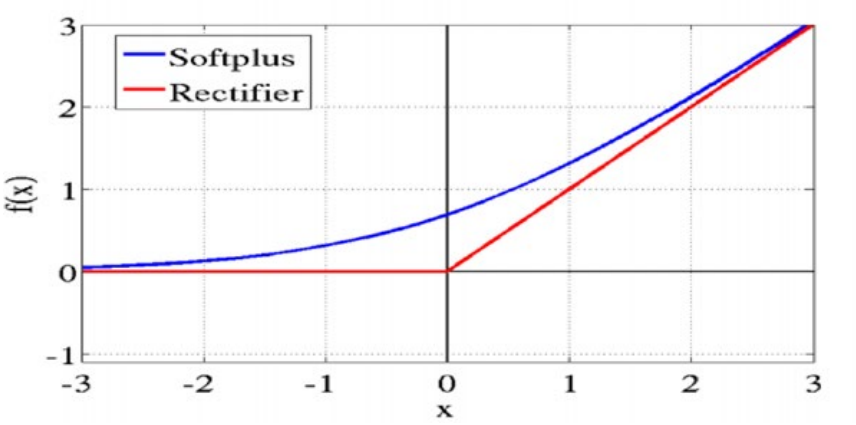

图 1.9: 函数坐标图

1.8. 损失函数

损失函数（成本函数）将被最小化，以便为模型的每个参数获得最佳值。例如，你需要得到权重（slope）和偏差（y-intercept）的最佳值，以便用预测器 (x) 来解释目标 (y)。实现斜率和 y 截距的最佳值的方法是使成本函数/损失函数/平方之和最小化。对于任何模型，都有许多参数，预测或分类中的模型结构是用参数值来表示的。

你需要评估你的模型，为此你需要定义成本函数（损失函数）。损失函数的最小化可以成为寻找每个参数的最佳值的驱动力。对于回归/数字预测，$L1$ 或 $L2$ 可以是有用的损失函数。对于分类，交叉熵可以是有用的损失函数。Softmax 或 sigmoid 交叉熵可以是相当流行的损失函数。

1.8.1. 损失函数实例

下面是演示的代码。

Listing 1.52: 损失函数代码实例 1

```
import tensorflow as tf
import nunpy as np
sess - tf.Session()
```

Listing 1.53: 损失函数代码实例 2

```
#Assuming prediction model
pred-np-asarray([0.2,0.3,0.5,10.0,12.0,13.0,3.5,7.4,3.9,2.3])
#convert ndarray into tensor
X_val=tf.convert_to_tensor (pred)
#Assuming actual values
actual=np. asarray([0.1,0.4,0.6,9.0,11.0,12.0,3.4,7.1,3.8,2.0])
```

Listing 1.54: 损失函数代码实例 3

```
#L2 Loss:L1 = (pred-actual)^2
l2 = tf. square(pred-actual)
l2_out = sess.run(tf.round(l2))
print(12_out)
```

Listing 1.55: 损失函数代码实例 4

```
#L2 Loss:L1-abs(pred-actual)
l1 = tf.abs (pred-actual)
l1_out = sess.run(l1)
print(l1_out)
```

Listing 1.56: 损失函数代码实例 5

```
#cross entropy Loss
softnax_xentropy_variable = tf.nn.sigmoid_cross_entropy_with_logits(logits=l1_out,labels=l2_out)
print (sess.run(softmax_xentropy_variable))
```

1.8.2. 常见的损失函数

以下是最常见的损失函数的列表。

tf.contrib.losses.absolute_difference

tf.contrib.losses.add_loss

tf.contrib.losses.hinge_loss

tf.contrib.losses.compute_weighted_loss

tf.contrib.losses.cosine_distance

tf.contrib.losses.get_losses

tf.contrib.losses.get_regularization_losses

tf.contrib.losses.get_total_loss

tf.contrib.losses.log_loss

tf.contrib.losses.mean_pairwise_squared_error

tf.contrib.losses.mean_squared_error

tf.contrib.losses.sigmoid_cross_entropy

tf.contrib.losses.softmax_cross_entropy

tf.contrib.losses.sparse_softmax_cross_entropy

tf.contrib.losses.log(predictions,labels,weight

1.9. 优化器

现在你应该相信，你需要使用损失函数来获得模型的每个参数的最佳值。如何才能得到最佳值呢？

最初，你假设模型（线性回归等）的权重和偏差的初始值。现在你需要找到达到参数最佳值的方法。优化器是达到参数最佳值的方法。在每次迭代中，参数值都会按照优化器建议的方向变化。假设你有 16 个权重值 $(w_1, w_2, w_3, \cdots, w_{16})$ 和 4 个偏置 (b_1, b_2, b_3, b_4)。最初你可以假设每个权重和偏置都是零（或一或任何数字）。优化器建议在下一次迭代中 w_1（和其他参数）是否应该增加或减少，同时牢记最小化的目标。经过多次迭代，w_1（和其他参数）将稳定到参数的最佳值（或数值）。

换句话说，TensorFlow 和其他每一个深度学习框架都提供了优化器，可以缓慢地改变每个参数，以最小化损失函数。优化器的目的是为下一次迭代中的权重和偏置的变化提供方向。假设你有 64 个权重和 16 个偏置；你试图在每次迭代中改变权重和偏置值（在反向传播期间），以便在多次迭代后得到正确的权重和偏置值，同时试图使损失函数最小化。

为模型选择最佳优化器以快速收敛并正确学习权重和偏置是一项棘手的任务。

自适应技术（adadelta、adagrad 等）是很好的优化器，可以让复杂的神经网络更快地收敛。据称，Adam 是大多数情况下的最佳优化器。它的性能也优于其他自适应技术（adadelta、adagrad 等），但它的计算成本很高。对于稀疏的数据集，SGD、NAG 和动量等方法不是最好的选择；自适应学习率方法才是。另外一个好处是，你不需要调整学习率，但很可能用默认值就能达到最佳效果。

1.9.1. 损失函数实例

损失函数实例代码如下：

Listing 1.57: 损失函数实例代码 1

```
# Importing Libraries
import tensorflow as tf
```

Listing 1.58: 损失函数实例代码 2

```
# Assign the value into variable
x = tf.Variable(3, name='x', dtype-tf,float32)
log_x = tf. log(x)
log_ x_squared = tf. square(log_x)
```

Listing 1.59: 损失函数实例代码 3

```
#Apply GradientDescentOptimizer
optimizer = tf.train.GradientDescentOptimizer(0.7)
train = optimizer.minimize(log_x_ squared)
```

Listing 1.60: 损失函数实例代码 4

```
# Initialize Variables
init = tf.global_variables_initializer()
```

Listing 1.61: 损失函数实例代码 4

```
# FinalLy running computation
with tf. Session() a session:
        session.run(init)
        print("starting at","x:",session.run(x), "log(x)^2:", session. run(log_x_squared))
        for step in range(10):
                session.run(train)
                print("step", step, *x:", session. run(x), "log(x)^2:", session.run(log_x_squared))
```

Listing 1.62: 损失函数实例代码 4

```
starting at x: 3.0 log(x)^2: 1.20695
step 0 x: 2.48731 log(x)^2: 0.830292
step 1 x: 1 1.97444 log(x) ^2: 0.462786
step 2 x: 1.49207 log(x)^2: 0.160134
step 3 x: 1.1166 log(x)^2: 0.0121637
step 4 x: 0.97832 log(x)^2: 0.00048043
step 5 x: 1.00969 log(x)^2: 9.29177e-05
step 6 x: 5 0.99632 log(x)^2: 1.35901e-05
```

```
step 7 x: 1.0015 log(x)^2: 2.24809e-06
step 8 x: 0.999405 log(x)^2: 3.54772e-07
step 9 x: 1.00024 log(x)^2: 5.70574e-08
```

1.9.2. 常见的优化器

下面是一个常见的优化器的列表。

tf.train.Optimizer

tf.train.GradientDescentOptimizer

tf.train.AdadeltaOptimizer

tf.train.AdagradOptimizer

tf.train.AdagradDAOptimizer

tf.train.MomentumOptimizer

tf.train.AdamOptimizer

tf.train.FtrlOptimizer

tf.train.ProximalGradientDescentOptimizer

tf.train.ProximalAdagradOptimizer

tf.train.RMSPropOptimizer

1.10. 衡量指标

在学习了一些建立模型的方法后，现在是评估模型的时候了。因此，你需要评估回归器或分类器。

有许多评估指标，其中分类准确率、对数损失和 ROC 曲线下的面积是最受欢迎的指标。

分类准确率是正确预测数与所有预测数的比率。当每个类别的观察结果没有太大的偏差时，准确率可以被认为是一个好的指标。

tf.contrib.metrics.accuracy(actual$_labels, predictions$)

还有其他的评估指标。

1.10.1. 衡量标准示例

本节展示了演示的代码。

在这里，你创建了实际值（称其为 x）和预测值（称其为 y）。然后，你检查准确性。准确度表示实际值等于预测值的次数与总实例数的比率。

1.10.2. 常见指标

以下是常见指标的清单:

tf.contrib.metrics.streaming_root_mean_squared_error

tf.contrib.metrics.streaming_covariance

tf.contrib.metrics.streaming_pearson_correlation

t.contrib.metrics.streaming_mean_cosine_distance

tf.contrib.metrics.streaming_percentage_less

tf.contrib.metrics.streaming_sensitivity_at_specificity

tf.contrib.metrics.streaming_sparse_average_precision_at_k
tf.contrib.metrics.streaming_sparse_precision_at_k
tf.contrib.metrics.streaming_sparse_precision_at_top_k
tf.contrib.metrics.streaming_specificity_at_ sensitivity
tf.contrib.metrics.streaming_concat
tf.contib.metrics.streaming_false_negatives
tf.contrib.metrics.streaming_false_negatives_at_thresholds

Listing 1.63: 常见指标示例 1

```
# Importing libraries
import numpy as np
import tensorflow as tf
```

Listing 1.64: 常见指标示例 2

```
# Placeholders declaration
x = tf. placeholder (tf. int32, [5])
y = tf. placeholder(tf. int32, [5])
```

Listing 1.65: 常见指标示例 2

```
# Metrices declaration
acc, acc_op=tf.metrics.accuracy(labels=x, predictions=y)
```

Listing 1.66: 常见指标示例 4

```
# Session initialization
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
sess.run(tf.local_variables_initializer())
```

Listing 1.67: 常见指标示例 5

```
# Value assign
val = sess.run([acc,acc_op], feed _dict={x: [1,1,0,1,0], y: [0,1,0,0,1]})
```

Listing 1.68: 常见指标示例 6

```
# Print Accuracy
val_acc = sess.run(acc)
print (val_acc)
# You can see only 2nd and 3rd positions value are same
```

Listing 1.69: 常见指标示例 7

```
0.4
```