深度学习库——TensorFlow 的基本使用

王锴

2019.04.30

目录

介绍

Tensorflow 是广泛使用的实现机器学习以及其它涉及大量数学运算的算法库之一。它由 Google 开发,2015 年 11 月,Google 公司开源了 TensorFlow ，随后不久 TensorFlow 成为 GitHub 上最受欢迎的机器学习库。Google 几乎在所有应用程序中都使用 Tensorflow 来实现机器学习。例如，如果你使用到了 Google 照片或 Google 语音搜索，那么你就间接使用了 Tensorflow 模型。它们在大型 Google 硬件集群上工作，在感知任务方面功能强大。

Google 著名人工智能程序 AlphaGo 于 2017 年年初化身 Master，在弈城和野狐等平台上连胜中日韩围棋高手,其中包括围棋世界冠军井山裕太、朴廷桓、柯洁等，还有棋圣聂卫平，总计取得 60 连胜，未尝败绩。而 AlphaGo 背后神秘的推动力就是 TensorFlow。DeepMind 宣布全面迁移到 TensorFlow 后，AlphaGo 的算法训练任务就全部放在了 TensorFlow 这套分布式框架上。

TensorFlow 采用数据流图（data flow graphs）用于数值计算，这些数据流图也称计算图是有向无环图，并且支持并行计算。节点（nodes）在图中表示数学操作，图中的线（edges）则表示在节点间相互联系的多维数据数组，即张量（tensor)。TensorFlow 创建计算图、自动求导和定制化的方式使得其能够很好地解决许多不同的机器学习问题,它灵活的架构让你可以在多种平台上展开计算，例如台式计算机中的一个或多个 CPU（或 GPU)、服务器和移动设备等。

本文将介绍 TensorFlow 的基本使用方法，第一章介绍 TensorFlow 的基础知识，第二章介绍 TensorFlow 计算图的有关操作，第三章将给出一个 TensorFlow 的具体应用实例，本文最后将总结 TensorFlow 算法的一般流程。

本文中的所有代码均在附件中。

第一章 TensorFlow 基础

本章将介绍 TensorFlow 中张量、占位符、变量、矩阵和激励函数的创建与基本操作方法，以及如何通过 TensorFlow 和 Python 访问各种数据源。

1.1 在 TensorFlow 中创建张量

张量是矢量概念的推广，标量是零阶张量，矢量是一阶阶张量，通常意义上的矩阵是二阶张量，除此之外，还有更高阶张量。例如，在 TensorFlow 中一张图片由一个四阶张量表示，它的四个维度分别表示为图片序号、高度、宽度和颜色通道，因此它可以表示一张动图。下面介绍如何在 TensorFlow 中创建张量。

首先，导入必要的模块（本节中的代码文件为 tensorflow_tensor.py）：

```
import tensorflow as tf
import numpy as np
from tensorflow.python.framework import ops
```

在创建一个新的计算图之前，初始化计算图是一个良好的习惯，当然，大部分时候，即便不初始化计算图也不会出错，但有少数情况下会导致程序崩溃，例如在交互式 Python 编译器中上一个计算图创建过同名的含有未指定纬度大小的占位符。初始化计算图如下：

```
# 初始化计算图
ops.reset_default_graph()
# 创建计算图会话
sess = tf.Session()
```

在开始工作之前，我们首先定义一个函数用来打印张量的值和大小，以减少繁琐的输入：

```
""" 查看张量的值和大小的函数 """
def print_tensor(tensor_to_print):
    # 打印张量名
    print("\n" + tensor_to_print + ":")
    # 打印张量的值
    with tf.Session() as sess: # 创建一个图会话
        print(eval("sess.run(" + tensor_to_print + ")"))
    # 打印张量的大小
```

```
    print(eval(tensor_to_print + ".shape"))
```

1.1.1 固定张量

固定张量是指根据既定值创建的张量，包括创建指定维度的零张量、指定维度的单位张量、指定维度的常数填充的张量、用已知常数张量创建的张量和广播一个值为张量：

```
# 创建指定维度的零张量
zero_tsr = tf.zeros([1, 2])
print_tensor('zero_tsr')

zero_tsr:
[[0. 0.]]
(1, 2)
# 创建指定维度的单位张量
ones_tsr = tf.ones([2, 1])
print_tensor('ones_tsr')

ones_tsr:
[[1.]
 [1.]]
(2, 1)
# 创建指定维度的常数填充的张量
filled_tsr = tf.fill([2, 2], 42)
print_tensor('filled_tsr')

filled_tsr:
[[42 42]
 [42 42]]
(2, 2)
# 用已知常数张量创建一个张量
constant_tsr = tf.constant([[1,2,3],[4,5,6],[7,8,9]])
print_tensor('constant_tsr')

constant_tsr:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
(3, 3)
# 广播一个值为张量
broadcast_tsr = tf.constant(42, shape=[1, 2])
print_tensor('broadcast_tsr')
```

```
broadcast_tsr:
[[42 42]]
(1, 2)
```

1.1.2 相似形状的张量

tf.zeros_like() 和 tf.ones_like() 函数可以用来创建相似形状的张量。注意，因为这些张量依赖于给定的张量，所以初始化时需要按序进行。如果打算一次性初始化所有张量，那么程序将会报错：

```
# 新建一个与给定的 tensor 类型大小一致的 tensor，其所有元素为 0 或者 1
zeros_similar = tf.zeros_like(constant_tsr)
print_tensor('zeros_similar')

zeros_similar:
[[0 0 0]
 [0 0 0]
 [0 0 0]]
(3, 3)

ones_similar = tf.ones_like(constant_tsr)
print_tensor('ones_similar')

ones_similar:
[[1 1 1]
 [1 1 1]
 [1 1 1]]
(3, 3)
```

1.1.3 序列张量

TensorFlow 可以创建指定间隔的张量，下面的函数的输出跟 numpy 的 linspace() 函数和 python 的 range() 函数的输出相似：

```
# 创建指定间隔的张量
linear_tsr = tf.linspace(start=0.0, stop=1.0, num=3) # 包含 stop
print_tensor('linear_tsr')

linear_tsr:
[0.  0.5 1. ]
(3,)
```

```
integer_seq_tsr = tf.range(start=6, limit=15, delta=3) # 不包含 limit
print_tensor('integer_seq_tsr')

integer_seq_tsr:
[ 6  9 12]
(3,)
```

1.1.4 随机张量

TensorFlow 的内置方法可以生成均匀分布、正态分布和指定边界的正态分布随机数，也可以对张量进行随机化和随机裁剪操作：

```
# 生成均匀分布的随机数
randunif_tsr = tf.random_uniform([3, 2], minval=0, maxval=1)
print_tensor('randunif_tsr')

randunif_tsr:
[[0.00991988 0.07117581]
 [0.04049361 0.05054855]
 [0.2913829  0.03344667]]
(3, 2)
# 生成正态分布的随机数
randnorm_tsr = tf.random_normal([2, 3], mean=0.0, stddev=1.0)
print_tensor('randnorm_tsr')

randnorm_tsr:
[[-0.9756406  -0.18820201  0.04726819]
 [-1.2959559   0.40868336  0.79056275]]
(2, 3)
# 生成带有指定边界的正态分布的随机数，其正态分布的随机数位于指定均值（期望）到两个标准差之间的区间
randtruncnorm_tsr = tf.truncated_normal([3, 3], mean=0.0, stddev=1.0)
print_tensor('randtruncnorm_tsr')

randtruncnorm_tsr:
[[-1.2924061  -1.8048544   1.8702108 ]
 [-1.7231227  -1.3029523  -1.8433377 ]
 [-1.4790725   0.07584414 -0.5379492 ]]
(3, 3)
# 张量的随机化(沿着第一纬度)
randshuffled_tsr = tf.random_shuffle(constant_tsr)
print_tensor('randshuffled_tsr')
```

```
randshuffled_tsr:
[[7 8 9]
 [4 5 6]
 [1 2 3]]
(3, 3)
# 张量的随机剪裁
randcropped_tsr = tf.random_crop(constant_tsr, [1, 2])
print_tensor('randcropped_tsr')

randcropped_tsr:
[[2 3]]
(1, 2)
```

1.1.5 将其他数据变成张量

TensorFlow 也可将列表或 numpy 数组转为张量：

```
# 将列表转为张量
list_data = list([1,2,3])
listdata_tsr = tf.convert_to_tensor(list_data)
print_tensor('listdata_tsr')

listdata_tsr:
[1 2 3]
(3,)
# 将 numpy 数组转为张量
nparray_data = np.array([4,5,6])
nparraydata_tsr = tf.convert_to_tensor(nparray_data)
print_tensor('nparraydata_tsr')

nparraydata_tsr:
[4 5 6]
(3,)
```

1.2 使用 TensorFlow 的变量和占位符

TensorFlow 中的变量是指那些在模型迭代优化过程中不断进行调整以使模型损失达到最小的可变的值，它们是一个学习模型或计算图的可变参数，变量在模型开始训练之前需要进行初始化。而占位符是指一个计算图中输入数据的节点

位置，它允许模型迭代过程中输入与之大小匹配的数据，可以是训练数据，也可以是验证和检验数据。（本节中的代码文件为 tensorflow_placeholders_and_variables.py）

1.2.1 创建变量并初始化

TensorFlow 通过封装一个张量来创建变量，声明所有变量后需要在迭代模型之前初始化变量：

```
# 初始化计算图
ops.reset_default_graph()
sess = tf.Session()
# 创建张量
my_tsr = tf.zeros([2,3])
# 封装张量来作为变量
my_var = tf.Variable(my_tsr)
# 初始化变量
initialize_op = tf.global_variables_initializer()
sess.run(initialize_op)
# 打印变量值和大小
print('\nmy_var:')
print(sess.run(my_var))
print(my_var.shape)

my_var:
[[0. 0. 0.]
 [0. 0. 0.]]
(2, 3)
```

1.2.2 按序进行变量的初始化

对于那些依赖于其他变量而创建的变量，必须按序进行初始化：

```
# 创建一个图会话
sess = tf.Session()
first_var = tf.Variable(tf.zeros([3,2]))
# 初始化 first_var
sess.run(first_var.initializer)
print('\nfirst_var:')
print(sess.run(first_var))
print(my_var.shape)

first_var:
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]]
(2, 3)
# second_var 依赖于 first_var
second_var = tf.Variable(tf.ones_like(first_var))
# 初始化 second_var
sess.run(second_var.initializer)
print('\nsecond_var:')
print(sess.run(second_var))
print(my_var.shape)

second_var:
[[1. 1.]
 [1. 1.]
 [1. 1.]]
(2, 3)
```

1.2.3 占位符的使用

占位符仅仅声明数据位置，用于传入数据到计算图：

```
# 创建一个图会话
sess = tf.Session()
# 创建占位符 x
x = tf.placeholder(tf.float32, shape=[2,2])
# y 作为一个虚拟节点来控制计算图的操作
y = tf.identity(x)
# x_vals 是要传入到计算图的数据
x_vals = np.random.rand(2,2)
# 传入数据到计算图
sess.run(y, feed_dict={x: x_vals})
```

1.3 TensorFlow 操作矩阵

矩阵事实上是二阶张量，许多机器学习算法依赖矩阵操作，因此 TensorFlow 中有一套专门用于矩阵操作的方法。在开始之前，我们同样先定义一个用于打印矩阵值和大小的函数，这事实上和打印张量的函数是一样的：（本节中的代码文件为 tensorflow_matrices.py）

```
""" 查看矩阵的值和大小的函数 """
```

```
def print_matrix(matrix_to_print):
    # 打印矩阵名
    print("\n" + matrix_to_print + ":")
    # 打印矩阵的值
    with tf.Session() as sess: # 创建一个图会话
        print(eval("sess.run(" + matrix_to_print + ")"))
    # 打印矩阵的大小
    print(eval(matrix_to_print + ".shape"))
```

1.3.1 创建矩阵

TensorFlow 可以使用 numpy 数组（或者嵌套列表）来创建矩阵，也可以从二阶张量创建矩阵，也可以用 tf.diag() 函数创建对角矩阵：

```
# 创建对角矩阵
diag_matrix = tf.diag([1.0, 1.0, 1.0])
print_matrix('diag_matrix')

diag_matrix:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
(3, 3)
# 从张量创建矩阵
truncated_matrix = tf.truncated_normal([2, 3])
print_matrix('truncated_matrix')

truncated_matrix:
[[ 0.14744104    1.0317733    -0.06878334]
 [ 0.90400624    0.06932766 -1.3771396 ]]
(2, 3)

filled_matrix = tf.fill([2,3], 5.0)
print_matrix('filled_matrix')

filled_matrix:
[[5. 5. 5.]
 [5. 5. 5.]]
(2, 3)

randunif_matrix = tf.random_uniform([3,2])
print_matrix('randunif_matrix')
```

```
randunif_matrix:
[[0.32320595 0.60990095]
 [0.41111875 0.6045526 ]
 [0.4693904  0.5809059 ]]
(3, 2)

npconv_matrix = tf.convert_to_tensor(np.array([[1., 2., 3.],[-3., -7., -1.],[0., 5., -2.]]))
print_matrix('npconv_matrix')

npconv_matrix:
[[ 1.  2.  3.]
 [-3. -7. -1.]
 [ 0.  5. -2.]]
(3, 3)
```

1.3.2 矩阵转置

tf.transpose() 函数用于求矩阵的转置：

```
A = tf.constant([[1.,2.,3.],[4.,5.,6.]])
print_matrix('A')

A:
[[1. 2. 3.]
 [4. 5. 6.]]
(2, 3)

A_trans = tf.transpose(A)
print_matrix('A_trans')

A_trans:
[[1. 4.]
 [2. 5.]
 [3. 6.]]
(3, 2)
```

1.3.3 矩阵乘法

进行矩阵乘法运算时必须符合矩阵乘法的运算规则，即前者的列数应与后者的行数一致：

```
B = tf.constant([[4.,5.,6.],[7.,8.,9.]])
```

```
print_matrix('B')

B:
[[4. 5. 6.]
 [7. 8. 9.]]
(2, 3)

AT_matmul_B = tf.matmul(A_trans,B)
print_matrix('AT_matmul_B')

AT_matmul_B:
[[32. 37. 42.]
 [43. 50. 57.]
 [54. 63. 72.]]
(3, 3)
```

1.3.4 矩阵内积

矩阵内积有两种方式，它们与矩阵乘法有着本质上的区别：

```
A_mul_B = A*B
print_matrix('A_mul_B')

A_mul_B:
[[ 4. 10. 18.]
 [28. 40. 54.]]
(2, 3)

A_mul_B_tf = tf.multiply(A, B)
print_matrix('A_mul_B_tf')

A_mul_B_tf:
[[ 4. 10. 18.]
 [28. 40. 54.]]
(2, 3)
```

1.3.5 矩阵的加法和减法

矩阵加法和减法均有两种方式：

```
# 加法有两种方式
A_add_B = A + B
print_matrix('A_add_B')
```

```
A_add_B:
[[ 5.  7.  9.]
 [11. 13. 15.]]
(2, 3)

A_add_B_tf = tf.add(A, B)
print_matrix('A_add_B_tf')

A_add_B_tf:
[[ 5.  7.  9.]
 [11. 13. 15.]]
(2, 3)
# 减法有两种方式
A_subtract_B = A - B
print_matrix('A_subtract_B')

A_subtract_B:
[[-3. -3. -3.]
 [-3. -3. -3.]]
(2, 3)

A_subtract_B_tf = tf.subtract(A, B)
print_matrix('A_subtract_B_tf')

A_subtract_B_tf:
[[-3. -3. -3.]
 [-3. -3. -3.]]
(2, 3)
```

1.3.6 矩阵的行列式

tf.matrix_determinant() 函数用于求取矩阵的行列式，它的结果是一个零阶张量：

```
C = tf.constant([[1.,2.,3.],[5.,6.,4.],[9.,7.,8.]])
print_matrix('C')

C:
[[1. 2. 3.]
 [5. 6. 4.]
 [9. 7. 8.]]
(3, 3)
```

```
C_det = tf.matrix_determinant(C)
print_matrix('C_det')

C_det:
-45.000004
()
```

1.3.7 矩阵的逆矩阵

TensorFlow 中的矩阵求逆方法是 Cholesky 矩阵分解法（又称为平方根法），因此矩阵需要为对称正定矩阵或者可进行 LU 分解，否则报错：

```
C_inv = tf.matrix_inverse(C)
print_matrix('C_inv')

C_inv:
[[-0.44444442 -0.11111114  0.2222222 ]
 [ 0.08888893  0.42222223 -0.24444446]
 [ 0.42222217 -0.24444443  0.08888891]]
(3, 3)
```

1.3.8 矩阵的分解

Cholesky 矩阵分解法如下：

```
D = tf.diag([1.0, 2.0, 3.0])
print_matrix('D')

D:
[[1. 0. 0.]
 [0. 2. 0.]
 [0. 0. 3.]]
(3, 3)

D_chol = tf.cholesky(D)
print_matrix('D_chol')

D_chol:
[[1.        0.        0.       ]
 [0.        1.4142135 0.       ]
 [0.        0.        1.7320508]]
(3, 3)
# 检验是否正确
```

```
D_test = tf.matmul(D_chol, tf.transpose(D_chol))
print_matrix('D_test')

D_test:
[[1.          0.          0.        ]
 [0.          1.9999999 0.        ]
 [0.          0.          3.        ]]
(3, 3)
```

1.3.9 矩阵的切片与连接

矩阵切片可以使用类似 Matlab 中的方法，也可以使用 TensorFlow 内置方法：

```
# 矩阵切片有两种方式
D_slice = D[0,:]
print_matrix('D_slice')

D_slice:
[1. 0. 0.]
(3,)

D_slice_tf = tf.slice(D, begin=[0,0], size=[1,3])
print_matrix('D_slice_tf')

D_slice_tf:
[[1. 0. 0.]]
(1, 3)
# 矩阵连接
D_concat = tf.concat([C,D], axis=0)
print_matrix('D_concat')

D_concat:
[[1. 2. 3.]
 [5. 6. 4.]
 [9. 7. 8.]
 [1. 0. 0.]
 [0. 2. 0.]
 [0. 0. 3.]]
(6, 3)
```

1.3.10 对称矩阵的特征值和特征向量

TensorFlow 只能计算对称矩阵的特征值与特征向量，与 np.linalg.eigh()函数相同，非对称矩阵求取的特征值与特征向量并非真正意义上的特征值与特征向量：

```
E = tf.constant([[1.,2.,3.],[2.,4.,5.],[3.,5.,6.]]) # 对称矩阵
print_matrix('E')

E:
[[1. 2. 3.]
 [2. 4. 5.]
 [3. 5. 6.]]
(3, 3)

E_eigenvalues, E_eigenvectors = tf.self_adjoint_eig(E)
print_matrix('E_eigenvalues')
print_matrix('E_eigenvectors')

E_eigenvalues:
[-0.5157295     0.17091452 11.344817   ]
(3,)

E_eigenvectors:
[[ 0.7369764     0.5910092     0.32798532]
 [ 0.3279854    -0.7369763     0.59100896]
 [-0.5910091     0.3279852     0.73697627]]
(3, 3)
# 检验是否正确
E_test =
tf.matmul(tf.matmul(E_eigenvectors,tf.diag(E_eigenvalues)),tf.matrix_inverse(E_ei
genvectors))
print_matrix('E_test')

E_test:
[[1.          2.0000007 3.0000014]
 [2.0000002 3.9999995 5.0000014]
 [3.          5.000001  6.0000024]]
(3, 3)
```

1.4 TensorFlow 张量的基本操作

介绍完了二阶张量即矩阵的操作之后，我们介绍一下普通张量的基本操作，

普通张量的操作通常是对张量间单个对应元素的操作。（本节中的代码文件为tensorflow_tensor_operations.py）

1.4.1 张量的加减乘除

这里需要注意的是张量间的除法运算，使用除法运算符事实上得到的是商，除此之外，TensorFlow 还内置了计算三维向量的外积的方法 tf.cross()，即所谓的向量叉乘运算：

```
A = tf.constant([1,2])
print_tensor('A')

A:
[1 2]
(2,)

B = tf.constant([3,4])
print_tensor('B')

B:
[3 4]
(2,)
# 张量加法
A_add_B = A + B
print_tensor('A_add_B')

A_add_B:
[4 6]
(2,)

A_add_B_tf = tf.add(A, B)
print_tensor('A_add_B_tf')

A_add_B_tf:
[4 6]
(2,)
# 张量减法
A_subtract_B = A - B
print_tensor('A_subtract_B')

A_subtract_B:
[-2 -2]
(2,)
```

```
A_subtract_B_tf = tf.subtract(A, B)
print_tensor('A_subtract_B_tf')

A_subtract_B_tf:
[-2 -2]
(2,)
# 张量元素间乘法
A_multiply_B = A*B
print_tensor('A_multiply_B')

A_multiply_B:
[3 8]
(2,)

A_multiply_B_tf = tf.multiply(A, B)
print_tensor('A_multiply_B_tf')

A_multiply_B_tf:
[3 8]
(2,)
# 张量元素间除法
A_divide_B = tf.divide(A, B)
print_tensor('A_divide_B')

A_divide_B:
[0.33333333 0.5        ]
(2,)
# 求商
rem_A_divide_B = A/B
print_tensor('rem_A_divide_B')

rem_A_divide_B:
[0 0]
(2,)

rem_A_divide_B_tf = tf.floordiv(A, B)
print_tensor('rem_A_divide_B_tf')

rem_A_divide_B_tf:
[0 0]
(2,)
# 求余数
mod_A_divide_B = A%B
```

```
print_tensor('mod_A_divide_B')

mod_A_divide_B:
[1 2]
(2,)

mod_A_divide_B_tf = tf.mod(A, B)
print_tensor('mod_A_divide_B_tf')

mod_A_divide_B_tf:
[1 2]
(2,)
# 三维向量的外积（只能是三维向量）
C = tf.constant([1.,0.,0.])
print_tensor('C')

C:
[1. 0. 0.]
(3,)

D = tf.constant([0.,1.,0.])
print_tensor('D')

D:
[0. 1. 0.]
(3,)

C_cross_D = tf.cross(C, D)
print_tensor('C_cross_D')

C_cross_D:
[0. 0. 1.]
(3,)
```

1.4.2 数学函数

TensorFlow 内置了一些常用的基本数学函数，如绝对值、倒数、相反数、三角函数、指数函数、对数函数和符号函数等。需要注意的是最大值、最小值和均值函数是以 reduce_ 开头的函数，意味着经过运算后张量的维度将会减少：

```
E = tf.constant([-1.2,0.3,4.5])
print_tensor('E')
```

```
E:
[-1.2    0.3    4.5]
(3,)
# 绝对值
abs_E = tf.abs(E)
print_tensor('abs_E')

abs_E:
[1.2 0.3 4.5]
(3,)
# 倒数
rec_E = tf.reciprocal(E)
print_tensor('rec_E')

rec_E:
[-0.8333333      3.3333333      0.22222222]
(3,)
# 相反数
neg_E = tf.negative(E)
print_tensor('neg_E')

neg_E:
[ 1.2 -0.3 -4.5]
(3,)
# 最大值
max_E = tf.reduce_max(E)
print_tensor('max_E')

max_E:
4.5
()
# 最小值
min_E = tf.reduce_min(E)
print_tensor('min_E')

min_E:
-1.2
()
# 平均值
mean_E = tf.reduce_mean(E)
print_tensor('mean_E')

mean_E:
1.1999999
```

```
()
# 求和
sum_E = tf.reduce_sum(E)
print_tensor('sum_E')

sum_E:
3.6
()
# 符号函数
sign_E = tf.sign(E)
print_tensor('sign_E')

sign_E:
[-1.  1.  1.]
(3,)
# 向上取整
ceil_E = tf.ceil(E)
print_tensor('ceil_E')

ceil_E:
[-1.  1.  5.]
(3,)
# 向下取整
floor_E = tf.floor(E)
print_tensor('floor_E')

floor_E:
[-2.  0.  4.]
(3,)
# 四舍五入
round_E = tf.round(E)
print_tensor('round_E')

round_E:
[-1.  0.  4.]
(3,)
# 两个张量中对应元素的最大值
max_D_E = tf.maximum(D, E)
print_tensor('max_D_E')

max_D_E:
[0.  1.  4.5]
(3,)
# 两个张量中对应元素的最小值
```

```
min_D_E = tf.minimum(D, E)
print_tensor('min_D_E')

min_D_E:
[-1.2    0.3    0. ]
(3,)
# 正弦函数
sin_E = tf.sin(E)
print_tensor('sin_E')

sin_E:
[-0.9320391      0.29552022 -0.9775301 ]
(3,)
# 余弦函数
cos_E = tf.cos(E)
print_tensor('cos_E')

cos_E:
[ 0.3623577    0.9553365 -0.2107958]
(3,)
# 自然底数的指数函数
exp_E = tf.exp(E)
print_tensor('exp_E')

exp_E:
[ 0.3011942    1.3498589 90.01713    ]
(3,)
# 自然对数
log_E = tf.log(E)
print_tensor('log_E')

log_E:
[            nan -1.2039728    1.5040774]
(3,)
# 对应元素的幂
E_pow_D = tf.pow(E, D)
print_tensor('E_pow_D')

E_pow_D:
[1.    0.3 1. ]
(3,)
# 平方
square_E = tf.square(E)
print_tensor('square_E')
```

```
square_E:
[ 1.44    0.09 20.25]
(3,)
# 平方根
sqrt_E = tf.sqrt(E)
print_tensor('sqrt_E')

sqrt_E:
[          nan 0.5477226 2.1213202]
(3,)
# 平方根的倒数
rsqrt_E = tf.rsqrt(E)
print_tensor('rsqrt_E')

rsqrt_E:
[         nan 1.8257418  0.47140455]
(3,)
```

1.4.3 特殊数学函数

有些用在机器学习中的特殊数学函数值得一提，TensorFlow 也有对应的内建函数，例如计算差的平方的函数和一些基本的统计函数：

```
# 两个张量差的平方
sqdiff_D_E = tf.squared_difference(D, E)
print_tensor('sqdiff_D_E')

sqdiff_D_E:
[ 1.44              0.48999998 20.25          ]
(3,)
# 高斯误差函数
erf_E = tf.erf(E)
print_tensor('erf_E')

erf_E:
[-0.91031396   0.32862678   1.              ]
(3,)
# 互补误差函数
erfc_E = tf.erfc(E)
print_tensor('erfc_E')

erfc_E:
```

```
[1.9103140e+00 6.7137325e-01 1.9661604e-10]
(3,)
# 下不完全伽马函数
igamma_D_E = tf.igamma(D, E)
print_tensor('igamma_D_E')

igamma_D_E:
[           nan 0.25918174           nan]
(3,)
# 上不完全伽马函数
igammac_D_E = tf.igammac(D, E)
print_tensor('igammac_D_E')

igammac_D_E:
[           nan 0.74081826           nan]
(3,)
# 贝塔函数绝对值的自然对数，纬度降 1
lbeta_E = tf.lbeta(E)
print_tensor('lbeta_E')

lbeta_E:
3.815787
()
# 伽马函数绝对值的自然对数
lgamma_E = tf.lgamma(E)
print_tensor('lgamma_E')

lgamma_E:
[1.5791758 1.0957979 2.4537365]
(3,)
# Psi 函数，lgamma 函数的导数
Psi_E = tf.digamma(E)
print_tensor('Psi_E')

Psi_E:
[ 4.8683233 -3.502524    1.3888711]
(3,)
```

1.4.4 组合基本函数生成自定义函数

我们通过组合 TensorFlow 基本函数生成许多自定义函数，例如正切函数：

```
# 正切函数
F = tf.constant([0.,3.1416/4.])
```

```
print_tensor('F')

F:
[0.         0.7854]
(2,)

tan_F = tf.divide(tf.sin(F), tf.cos(F))
print_tensor('tan_F')

tan_F:
[0.          1.0000036]
(2,)
# 二次多项式函数
def my_polynomial(value): # 3x^2-x+10
    return(tf.subtract(3 * tf.square(value), value) + 10)
poly_F = my_polynomial(F)
print_tensor('poly_F')

poly_F:
[10.         11.065159]
(2,)
```

1.5 TensorFlow 的激励函数

激励函数是使用所有神经网络算法的必备“神器”，TensorFlow 的激励函数位于神经网络（neural network，nn）库。激励函数的目的是为了调节权重和误差。在 TensorFlow 中，激励函数是作用在张量上的非线性操作。激励函数的使用方法和前面的数学操作相似。激励函数的功能有很多，但其主要是为计算图归一化返回结果而引进的非线性部分。（本节中的代码文件为 tensorflow_activation_functions.py）

1.5.1 整流线性单元（Rectifier linear unit，ReLU）

整流线性单元（Rectifier linear unit，ReLU）是神经网络最常用的非线性函数。其函数为 max（0，x），连续但不平滑：

```
A = tf.constant([-3.,3.,10.])
print_tensor('A')

A:
```

```
[-3.  3. 10.]
(3,)

relu_A = tf.nn.relu(A)
print_tensor('relu_A')

relu_A:
[ 0.  3. 10.]
(3,)
```

1.5.2 ReLU6 函数

有时为了抵消 ReLU 激励函数的线性增长部分，会在 min（） 函数中嵌入 max（o，x），其在 TensorFlow 中的实现称作 ReLU6，表示为 min（max（0，x），6）。这是 hard-sigmoid 函数的变种，计算运行速度快，解决梯度消失（无限趋近于 0）效应：

```
relu6_A = tf.nn.relu6(A)
print_tensor('relu6_A')

relu6_A:
[0. 3. 6.]
(3,)
```

1.5.3 sigmoid 函数

sigmoid 函数是最常用的连续、平滑的激励函数。它也被称作逻辑函数（Logistic 函数），表示为 1/（1+exp（-x））。sigmoid 函数由于在机器学习训练过程中反向传播项趋近于 0，因此不怎么使用：

```
sigmoid_A = tf.nn.sigmoid(A)
print_tensor('sigmoid_A')

sigmoid_A:
[0.04742587 0.95257413 0.9999546 ]
(3,)
```

1.5.4 双曲正切函数（hyper tangent，tanh）

另外一种激励函数是双曲正切函数（hyper tangent，tanh）。双曲正切函数

与 sigmoid 函数相似，但有一点不同：双曲正切函数取值范围为 0 到 1；sigmoid 函数取值范围为 -1 到 1。双曲正切函数是双曲正弦与双曲余弦的比值，另外一种写法是（(exp（x）-exp（-x)）/（exp（x）+exp（-x））：

```
tanh_A = tf.nn.tanh(A)
print_tensor('tanh_A')

tanh_A:
[-0.9950547  0.9950547  1.        ]
(3,)
```

1.5.5 softsign 函数

softsign 函数也是一种激励函数，表达式为：x/（abs（x）+1）。 softsign 函数是符号函数的连续估计：

```
softsign_A = tf.nn.softsign(A)
print_tensor('softsign_A')

softsign_A:
[-0.75        0.75        0.90909094]
(3,)
```

1.5.6 softplus 激励函数

softplus 激励函数是 ReLU 激励函数的平滑版，表达式为：log（exp（x）+1)。需要注意的是，当输入增加时，softplus 激励函数趋近于无限大，softsign 函数趋近于 1；当输入减小时，softplus 激励函数趋近于 0，softsign 函数趋近于 -1：

```
softplus_A = tf.nn.softplus(A)
print_tensor('softplus_A')

softplus_A:
[ 0.04858733  3.0485873  10.000046  ]
(3,)
```

1.5.7 ELU 激励函数（Exponential Linear Unit，ELU）

ELU 激励函数（Exponential Linear Unit，ELU）与 softplus 激励函数相

似，不同点在于：当输入无限小时，ELU 激励函数趋近于 -1，而 softplus 激励函数趋近于 0。表达式为（exp（x）+1）if x<0 else x：

```
elu_A = tf.nn.elu(A)
print_tensor('elu_A')

elu_A:
[-0.95021296  3.          10.        ]
(3,)
```

1.6 通过 TensorFlow 和 Python 访问各种数据源

本节仅以马萨诸塞大学（University of Massachusetts，UMASS）的出生体重数据集（Birth weight data）的获取为例，更多数据集获取例子请参见代码文件 tensorflow_get_datasources.py，包括诸如鸢尾花卉数据集（Iris data）、波士顿房价数据（Boston Housing data）、MNIST 手写体字库、垃圾短信文本数据集（Spam-ham text data）、影评样本数据集（Movie review data）、莎士比亚著作文本数据集（Shakespeare text data）和英德句子翻译样本集（English-German sentence translation data）等数据集的获取方法。

出生体重数据集（Birth weight data）是婴儿出生体重以及母亲和家庭历史人口统计学、医学指标，有 189 个样本集，包含 11 个特征变量。使用 Python 访问的数据的方式如下：

```
import requests
birthdata_url = 'https://raw.githubusercontent.com/GeoKylin/GA---Data-
Science/master/Week%206/lowbwt.dat'
birth_file = requests.get(birthdata_url)
birth_data = birth_file.text.split('\n')
birth_header = [x for x in birth_data[0].split() if len(x)>=1]
birth_data = [[float(x) for x in y.split() if len(x)>=1] for y in birth_data[1:] if
len(y)>=1]
print('\nBirth weight data:')
print(len(birth_data))
# 189
print(len(birth_data[0]))
# 11
```

第二章 TensorFlow 计算图

Tensorflow 是一个通过计算图的形式来表述计算过程的编程系统，计算图也叫数据流图，它是一个有向无环图。Tensorflow 计算的过程就是利用 Tensor 来建立一个计算图，然后使用 Session 会话来启动计算，最后得到结果的过程。

2.1 TensorFlow 计算图中对象和层的操作

TensorFlow 计算图通常由多个层组成，这也体现了机器学习的“深度”。（本节中的代码文件为 tensorflow_graph_and_layer.py）

2.1.1 计算图中对象的操作

创建一个乘法操作的计算图（见图 2.1）。首先，创建数据集和计算图操作，然后传入数据、打印返回值：

```
# 初始化计算图
ops.reset_default_graph()
sess = tf.Session()
# 创建要传入的数据
x_vals = np.array([1., 3., 5., 7., 9.])
# 创建占位符
x_data = tf.placeholder(tf.float32)
# 创建常量矩阵
m_const = tf.constant(3.)
# 声明操作，表示成计算图
my_product = tf.multiply(x_data, m_const)
# 通过计算图赋值
print('Operations in a Computational Graph:')
for x_val in x_vals:
    print(sess.run(my_product, feed_dict={x_data: x_val}))
# 3.0
# 9.0
# 15.0
# 21.0
# 27.0
```

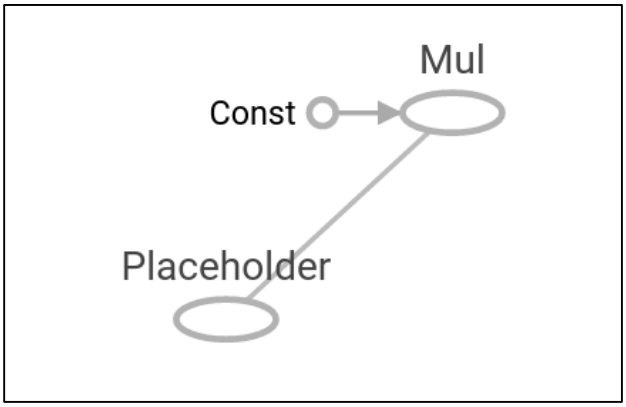

图 2.1 一个乘法操作的计算图

2.1.2 TensorFlow 嵌入 Layer 的操作

在同一个计算图中进行多个操作，计算图见图 2.2。传入两个形状为 3×5 的 numpy 数组，然后每个矩阵乘以常量矩阵（形状为：5×1），将返回一个形状为 3×1 的矩阵。紧接着再乘以 1×1 的矩阵，返回的结果矩阵仍然为 3×1。最后，加上一个 3×1 的矩阵：

```
# 创建要传入的数据
my_array = np.array([[1., 3., 5., 7., 9.],
                     [-2., 0., 2., 4., 6.],
                     [-6., -3., 0., 3., 6.]])
x_vals = np.array([my_array, my_array + 1])
# 创建占位符
x_data = tf.placeholder(tf.float32, shape=(3, 5))
# 创建常量矩阵
m1 = tf.constant([[1.],[0.],[-1.],[2.],[4.]])
m2 = tf.constant([[2.]])
a1 = tf.constant([[10.]])
# 声明操作，表示成计算图
prod1 = tf.matmul(x_data, m1)
prod2 = tf.matmul(prod1, m2)
add1 = tf.add(prod2, a1)
# 通过计算图赋值
print('\nLayering Nested Operations:')
for x_val in x_vals:
    print(sess.run(add1, feed_dict={x_data: x_val}))
# [[ 102.]
# [ 66.]
# [ 58.]]
# [[ 114.]
# [ 78.]
```

```
# [ 70.]]
```

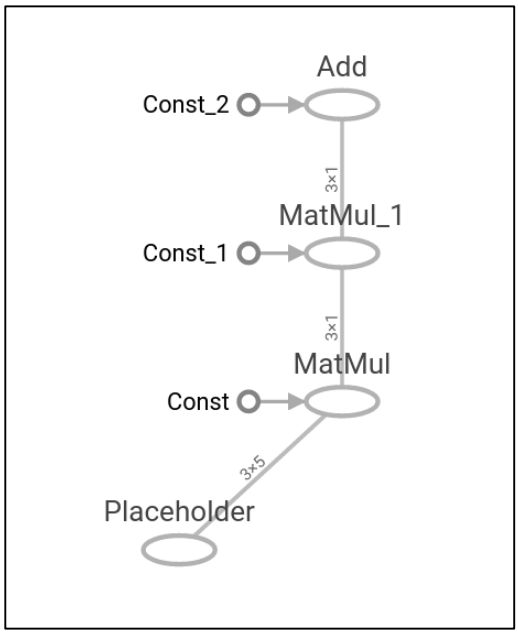

图 2.2 一个嵌入层操作的计算图

2.1.3 TensorFlow 多层 Layer 的操作

连接传播数据的多层 Layer，计算图见图 2.3。生成随机图片数据，对 2D 图像进行滑动窗口平均，然后通过自定义操作层 Layer 返回结果：

```
# 通过 numpy 创建 2D 图像，4×4 像素图片
# 注意：TensorFlow 的图像函数是处理四维图片的，这四维是：图片数量、高度、宽度和颜色通道
x_shape = [1, 4, 4, 1]
x_val = np.random.uniform(size=x_shape)
# 创建占位符，用来传入图片
x_data = tf.placeholder(tf.float32, shape=x_shape)
# 创建过滤 4×4 像素图片的滑动平均窗口（第一层 Layer）
my_filter = tf.constant(0.25, shape=[2, 2, 1, 1]) # 滑动窗口
my_strides = [1, 2, 2, 1] # 滑动窗口的步幅
mov_avg_layer= tf.nn.conv2d(x_data, my_filter, my_strides, padding='SAME', name='Moving_Avg_Window')
# 定义一个自定义 Layer，操作滑动平均窗口的 2×2 的返回值（第二层 Layer）
def custom_layer(input_matrix):
    # 压缩尺寸为 1 的纬度
    input_matrix_sqeezed = tf.squeeze(input_matrix)
    # 创建常量矩阵
    A = tf.constant([[1., 2.], [-1., 3.]])
```

```
        b = tf.constant(1., shape=[2, 2])
        # 声明操作，表示成计算图 Ax + b
        temp1 = tf.matmul(A, input_matrix_sqeezed)
        temp = tf.add(temp1, b)
        # 返回激励函数
        return(tf.nn.sigmoid(temp))
# 把新定义的 Layer 加入到计算图中，表示为 mov_avg_layer 层的下一层
with tf.name_scope('Custom_Layer') as scope:
        custom_layer1 = custom_layer(mov_avg_layer)
# 为占位符传入 4×4 像素图片，执行计算图
print('\nWorking with Multiple Layers:')
print(sess.run(custom_layer1, feed_dict={x_data: x_val}))
# [[0.95360523 0.92379194]
#  [0.92614335 0.89943546]]
```

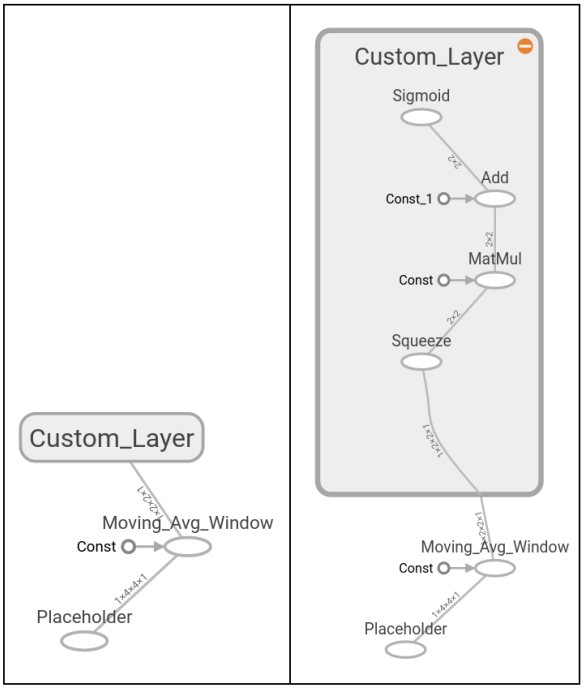

图 2.3 一个多层计算图的操作

2.2 TensorFlow 实现损失函数

为了优化机器学习算法，我们需要评估机器学习模型训练输出结果。在 TensorFlow 中评估输出结果依赖损失函数。损失函数告诉 TensorFlow，预测结果相比期望的结果是好是坏。在大部分场景下，我们会有算法模型训练的样本数据集和目标值。损失函数比较预测值与目标值，并给出两者之间的数值化的差值。为了比较不同损失函数的区别，我们将使用 matplotlib 将它们绘制出来。（本

节中的代码文件为 tensorflow_loss_functions.py）

2.2.1 回归算法的损失函数

回归算法是预测连续因变量的，常用的损失函数有 L2 正则损失函数（即欧拉损失函数）、L1 正则损失函数（即绝对值损失函数）和 Pseudo-Huber 损失函数。其中，Huber 损失函数的连续、平滑估计，试图利用 L1 和 L2 正则削减极值处的陡峭，使得目标值附近连续，它的表达式依赖参数 delta：

```
""" L2 正则损失函数（即欧拉损失函数） """
# 初始化计算图
ops.reset_default_graph()
sess = tf.Session()
# 创建预测序列和目标序列作为张量
x_vals = tf.linspace(-1., 1., 500)
target = tf.constant(0.)
# L2 正则损失
l2_y_vals = tf.square(target - x_vals)
l2_y_out = sess.run(l2_y_vals)
print('L2 norm loss:')
print('l2_y_out = %f' % sess.run(tf.reduce_sum(l2_y_vals)))
# TensorFlow 内建的 L2 正则损失是实际 L2 正则的一半
l2_y_vals_nn = tf.nn.l2_loss(target - x_vals)
l2_y_out_nn = sess.run(l2_y_vals_nn)
print('l2_y_out_nn = %f' % sess.run(tf.reduce_sum(l2_y_vals_nn)))
print('l2_y_out / l2_y_out_nn = %f' % sess.run(tf.divide(tf.reduce_sum(l2_y_vals),
l2_y_vals_nn)))

L2 norm loss:
l2_y_out = 167.334671
l2_y_out_nn = 83.667336
l2_y_out / l2_y_out_nn = 2.000000

""" L1 正则损失函数（即绝对值损失函数） """
l1_y_vals = tf.abs(target - x_vals)
l1_y_out = sess.run(l1_y_vals)

""" Pseudo-Huber 损失函数 """
""" Huber 损失函数的连续、平滑估计，试图利用 L1 和 L2 正则削减极值处的陡峭，
使得目标值附近连续。它的表达式依赖参数 delta。 """
# delta = 0.25 时
delta1 = tf.constant(0.25)
```

```
phuber1_y_vals = tf.multiply(tf.square(delta1), tf.sqrt(1. + tf.square((target - x_vals)/delta1)) - 1.)
phuber1_y_out = sess.run(phuber1_y_vals)
# delta = 5 时
delta2 = tf.constant(5.)
phuber2_y_vals = tf.multiply(tf.square(delta2), tf.sqrt(1. + tf.square((target - x_vals)/delta2)) - 1.)
phuber2_y_out = sess.run(phuber2_y_vals)
```

2.2.2 分类算法的损失函数

分类损失函数是用来评估预测分类结果的，常用的损失函数有 Hinge 损失函数（主要用来评估支持向量机算法，但有时也用来评估神经网络算法）、交叉熵损失函数（Cross-entropy loss，有时也作为逻辑损失函数）、Sigmoid 交叉熵损失函数（Sigmoid cross entropy loss，先把 x_vals 值通过 sigmoid 函数转换，再计算交叉熵损失）、加权交叉熵损失函数（Weighted cross entropy loss，Sigmoid 交叉熵损失函数的加权，对正目标加权）、Softmax 交叉熵损失函数（Softmax cross-entropy loss，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）和稀疏 Softmax 交叉熵损失函数（Sparse softmax cross-entropy loss，把目标分类为 true 的转化成 index，而 Softmax 交叉熵损失函数将目标转成概率分布）：

```
""" Hinge 损失函数，主要用来评估支持向量机算法，但有时也用来评估神经网络算法 """
# 创建预测序列和目标序列作为张量
x_vals = tf.linspace(-3., 5., 500)
target = tf.constant(1.)
targets = tf.fill([500,], 1.)
# Hinge 损失，使用目标值 1，预测值离 1 越近，损失函数值越小
hinge_y_vals = tf.maximum(0., 1. - tf.multiply(target, x_vals))
hinge_y_out = sess.run(hinge_y_vals)

""" 交叉熵损失函数（Cross-entropy loss），有时也作为逻辑损失函数 """
xentropy_y_vals = - tf.multiply(target, tf.log(x_vals)) - tf.multiply((1. -target), tf.log(1. - x_vals))
xentropy_y_out = sess.run(xentropy_y_vals)

""" Sigmoid 交叉熵损失函数（Sigmoid cross entropy loss） """
""" 先把 x_vals 值通过 sigmoid 函数转换，再计算交叉熵损失 """
```

```
xentropy_sigmoid_y_vals =
tf.nn.sigmoid_cross_entropy_with_logits(logits=x_vals, labels=targets)
xentropy_sigmoid_y_out = sess.run(xentropy_sigmoid_y_vals)

""" 加权交叉熵损失函数（Weighted cross entropy loss） """
""" 是 Sigmoid 交叉熵损失函数的加权，对正目标加权 """
# 将正目标加权权重 0.5
weight = tf.constant(0.5)
xentropy_weighted_y_vals =
tf.nn.weighted_cross_entropy_with_logits(logits=x_vals, targets=targets,
pos_weight=weight)
xentropy_weighted_y_out = sess.run(xentropy_weighted_y_vals)

""" Softmax 交叉熵损失函数（Softmax cross-entropy loss） """
""" 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 """
unscaled_logits = tf.constant([[1., -3., 10.]])
target_dist = tf.constant([[0.1, 0.02, 0.88]])
softmax_xentropy =
tf.nn.softmax_cross_entropy_with_logits_v2(logits=unscaled_logits,
labels=target_dist)
print('\nSoftmax cross-entropy loss:')
print(sess.run(softmax_xentropy))
# [ 1.16012561]

""" 稀疏 Softmax 交叉熵损失函数（Sparse softmax cross-entropy loss） """
""" 把目标分类为 true 的转化成 index，而 Softmax 交叉熵损失函数将目标
转成概率分布 """
unscaled_logits = tf.constant([[1., -3., 10.]])
sparse_target_dist = tf.constant([2])
sparse_xentropy =
tf.nn.sparse_softmax_cross_entropy_with_logits(logits=unscaled_logits,
labels=sparse_target_dist)
print('\nSparse softmax cross-entropy loss:')
print(sess.run(sparse_xentropy))
# [ 0.00012564]
```

2.2.3 用 matplotlib 绘制损失函数

因为篇幅原因，这里只给出结果图（图 2.4），绘图代码请参见代码文件 tensorflow_loss_functions.py，损失函数的选择通常与要解决的问题有关。

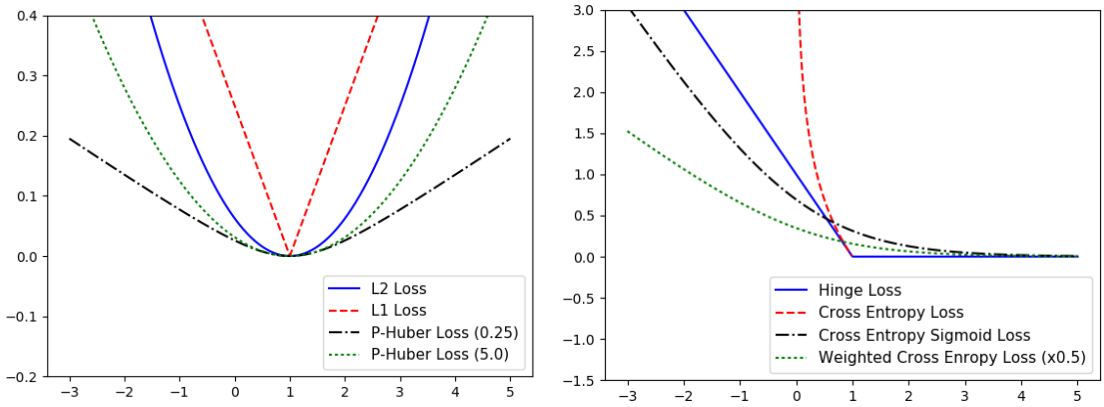

图 2.4 左图：回归算法的损失函数；右图：分类算法的损失函数

2.3 TensorFlow 实现反向传播

TensorFlow 基于反向传播自动地更新模型变量，调节模型变量来最小化损失函数。要实现反向传播，首先要声明优化函数（optimization function），声明好优化函数后，TensorFlow 将根据计算图计算反向传播的项。当我们传入数据和最小化损失函数时，TensorFlow 会在计算图中根据状态相应的调节变量。（本节中的代码文件为 tensorflow_back_propagation.py）

2.3.1 回归算法的例子

拟合一个平面 Z=0.1x+0.2y+0.3，损失函数采用 L2 正则损失，优化函数采用标准梯度下降法并最小化方差：

```
# 初始化计算图
ops.reset_default_graph()
sess = tf.Session()
# 使用 NumPy 生成假数据(phony data), 总共 100 个点
xy_vals = np.float32(np.random.rand(2, 100)) # 随机输入二维因变量
z_vals = np.dot([0.100, 0.200], xy_vals) + 0.300 # 平面 z=0.1x+0.2y+0.3
# 创建占位符
xy_data = tf.placeholder(shape=[2, 100], dtype=tf.float32)
z_target = tf.placeholder(shape=[100], dtype=tf.float32)
# 创建变量 W,b 作为拟合参数
W = tf.Variable(tf.random_uniform([1, 2], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
# 创建计算图操作，构造线性模型
z = tf.matmul(W, xy_vals) + b
# 创建 L2 正则损失函数
loss = tf.reduce_mean(tf.square(z - z_vals))
```

```
# 声明变量的优化器，标准梯度下降法，最小化方差
# 小学习率收敛慢、精度高；大学习率收敛快，精度低
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5)
train = optimizer.minimize(loss)
# 初始化变量
init = tf.global_variables_initializer()
# 启动图
sess.run(init)
# 训练算法，拟合平面。迭代 201 次，每 20 次迭代打印返回结果。
# 每次迭代将所有点坐标传入计算图中。TensorFlow 将自动地计算损失，调整 W,b 偏差来最小化损失
print('Regression example:')
for step in range(0, 200):
    sess.run(train, feed_dict={xy_data: xy_vals, z_target: z_vals})
    if (step+1)%20 == 0:
        print('Step #' + str(step+1) + ': W = ' + str(sess.run(W)) + '; b = ' +
                str(sess.run(b)) + '; Loss = ' +
                str(sess.run(loss, feed_dict={xy_data: xy_vals, z_target:
z_vals})))
# 得到最佳拟合结果应接近于 W = [[0.100   0.200]]; b = [0.300]

Regression example:
Step #20: W = [[-0.115726   0.170699]]; b = [0.41953927]; Loss = 0.0033730662
Step #40: W = [[0.0328682   0.17926575]]; b = [0.3434922]; Loss = 0.000358589
Step #60: W = [[0.078413 0.1900973]]; b = [0.31573766]; Loss = 4.1624484e-05
Step #80: W = [[0.0928651 0.1958547]]; b = [0.30567417]; Loss = 5.0792564e-06
Step #100: W = [[0.097589 0.1983708]]; b = [0.3020409]; Loss = 6.358276e-07
Step #120: W = [[0.099171 0.1993816]]; b = [0.30073288]; Loss = 8.057695e-08
Step #140: W = [[0.099712 0.1997701]]; b = [0.3002629]; Loss = 1.02705e-08
Step #160: W = [[0.099898 0.1999156]]; b = [0.30009422]; Loss = 1.3125765e-09
Step #180: W = [[0.099964 0.1999693 ]]; b = [0.30003378]; Loss = 1.6816148e-10
Step #200: W = [[0.099987 0.1999889 ]]; b = [0.30001208]; Loss = 2.1527589e-11
```

2.3.2 分类算法的例子

在建立另一个计算图之前，应该首先重置计算图。分类算法的例子是一个简单的二值分类算法，从两个正态分布（N（-1，1）和N（3，1））随机生成 100 个数。所有从正态分布 N（-1，1）生成的数据标为目标类 0；从正态分布 N（3，1）生成的数据标为目标类 1，模型算法通过 sigmoid 函数将这些生成的数据转换成目标类数据。我们试图找到一个最佳分类边界将正态分布分割成不同的两类，显然这个最佳分类边界应当趋近于 1，算法中变量 A 表示最佳聚类边界的负值，

即变量 A 随着迭代进行应当趋近于 -1：

```
# 重置计算图
ops.reset_default_graph()
sess = tf.Session()
# 从正态分布 (N(-1,1), N(3,1)) 生成数据，总共 100 个点
x_vals = np.concatenate((np.random.normal(-1, 1, 50), np.random.normal(3, 1,
50)))[:,np.newaxis]
# 创建目标标签，N(-1,1) 为 '0' 类，N(3,1) 为 '1' 类，各 50 个点
y_vals = np.concatenate((np.repeat(0., 50), np.repeat(1., 50)))[:,np.newaxis]
# 创建占位符，一次使用一个随机数据
x_data = tf.placeholder(shape=[100, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[100, 1], dtype=tf.float32)
# 创建变量 A 作为最佳聚类边界的负值，初始值为 10 附近的随机数，远离
理论值 -(-1+3)/2 = -1
A = tf.Variable(tf.random_normal(mean=10, shape=[1]))
# 创建计算图操作，模型算法是 sigmoid(x+A)
my_output = tf.add(x_data, A)
# 创建损失函数，使用 Sigmoid 交叉熵损失函数（Sigmoid cross entropy
loss）
xentropy =
tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(logits=my_output,
labels=y_target))
# 声明变量的优化器，标准梯度下降法，最小化交叉熵
my_opt = tf.train.GradientDescentOptimizer(0.01)
train_step = my_opt.minimize(xentropy)
# 初始化变量
init = tf.global_variables_initializer()
sess.run(init)
# 通过随机选择的数据迭代 101 次，相应地更新变量 A。每迭代 10 次打印
出变量 A 和损失的返回值
print('\nClassification example:')
for i in range(100):
    sess.run(train_step, feed_dict={x_data: x_vals, y_target: y_vals})
    if (i+1)%10 == 0:
        print('Step #' + str(i+1) + ': A = ' + str(sess.run(A)) + '; Loss = ' +
                str(sess.run(xentropy, feed_dict={x_data: x_vals, y_target:
y_vals})))
# 得到最佳拟合结果应趋近于 A = [-1.]
```

```
Classification example:
Step #10: A = [3.7139716]; Loss = 151.36496
Step #20: A = [0.33025175]; Loss = 33.819824
Step #30: A = [-0.6884335]; Loss = 22.662157
```

```
Step #40: A = [-0.95006174]; Loss = 21.926872
Step #50: A = [-1.0175546]; Loss = 21.877947
Step #60: A = [-1.034981]; Loss = 21.874685
Step #70: A = [-1.0394813]; Loss = 21.874468
Step #80: A = [-1.0406435]; Loss = 21.874454
Step #90: A = [-1.0409436]; Loss = 21.874453
Step #100: A = [-1.0410212]; Loss = 21.874453
```

2.4 TensorFlow 实现批量训练和随机训练

TensorFlow 通过反向传播算法更新模型变量，它可以一次训练一个数据点，也可以一次训练大量数据。一次训练一个数据点可能会导致比较“古怪”的学习过程，但使用大批量的训练会造成计算成本昂贵。前者称为随机训练，后者称为批量训练，具体选用哪种训练类型对机器学习算法的收敛非常关键。

本节以回归算法为例比较批量训练和随机训练，随机数据分布在（1，10）点附近，拟合变量 A 表示过数据中心点与原点所在直线的斜率，其理论值为 10。（本节中的代码文件为 tensorflow_batch_and_stochastic_training.py）

2.4.1 TensorFlow 批量训练

批量训练大小可以一次上扩到整个数据集，也可以是一个数据点，这时批量训练就退化为了随机训练：

```
# 初始化计算图
ops.reset_default_graph()
sess = tf.Session()
# 声明批量大小，批量大小是指通过计算图一次传入训练数据的多少
batch_size = 20
# 声明模型的数据、占位符和变量
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)
A = tf.Variable(tf.random_normal(shape=[1,1]))
# 在计算图中增加矩阵乘法操作
my_output = tf.matmul(x_data, A)
# 声明损失函数
loss = tf.reduce_mean(tf.square(my_output - y_target))
# 声明优化器
```

```
my_opt = tf.train.GradientDescentOptimizer(0.02)
train_step = my_opt.minimize(loss)
# 初始化变量
init = tf.global_variables_initializer()
# 启动图
sess.run(init)
# 通过循环迭代优化模型算法。每间隔 5 次迭代保存损失函数，用来绘制损失值图
print('Batch Training:')
loss_batch = []
for i in range(100):
    # 产生批量训练数据集
    rand_index = np.random.choice(100, size=batch_size)
    rand_x = np.transpose([x_vals[rand_index]])
    rand_y = np.transpose([y_vals[rand_index]])
    # 训练模型
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%5==0:
        temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
        print('Step #' + str(i+1) + ': A = ' + str(sess.run(A)) + '; Loss = ' +
str(temp_loss))
        loss_batch.append(temp_loss)
```

```
Batch Training:
Step #5: A = [[0.8000524]]; Loss = 85.24268
Step #10: A = [[2.4625654]]; Loss = 58.610535
Step #15: A = [[3.8429325]]; Loss = 38.12146
Step #20: A = [[4.968774]]; Loss = 26.04155
Step #25: A = [[5.8888993]]; Loss = 18.784458
Step #30: A = [[6.6416354]]; Loss = 12.07374
Step #35: A = [[7.2623706]]; Loss = 7.2879133
Step #40: A = [[7.7697453]]; Loss = 6.7543473
Step #45: A = [[8.18522]]; Loss = 4.2405763
Step #50: A = [[8.531599]]; Loss = 2.0551047
Step #55: A = [[8.795953]]; Loss = 2.1727984
Step #60: A = [[9.030075]]; Loss = 2.9733653
Step #65: A = [[9.210251]]; Loss = 1.801594
Step #70: A = [[9.356403]]; Loss = 1.5553963
Step #75: A = [[9.491015]]; Loss = 1.6395788
Step #80: A = [[9.583339]]; Loss = 0.9720012
Step #85: A = [[9.683713]]; Loss = 0.9329313
Step #90: A = [[9.782524]]; Loss = 1.1064217
Step #95: A = [[9.836063]]; Loss = 1.0621531
```

```
Step #100: A = [[9.851362]]; Loss = 0.9794037
```

2.4.2 TensorFlow 随机训练

随机训练可以看做是批量训练在训练集大小为 1 的情况下的特例：

```
print('\nStochastic Training:')
loss_stochastic = []
for i in range(100):
    # 产生随机训练数据
    rand_index = np.random.choice(100)
    rand_x = np.array([x_vals[rand_index]])[:,np.newaxis]
    rand_y = np.array([y_vals[rand_index]])[:,np.newaxis]
    # 训练模型
    sess.run(train_step, feed_dict={x_data: rand_x, y_target: rand_y})
    if (i+1)%5==0:
        temp_loss = sess.run(loss, feed_dict={x_data: rand_x, y_target: rand_y})
        print('Step #' + str(i+1) + ': A = ' + str(sess.run(A)) + '; Loss = ' +
str(temp_loss))
        loss_stochastic.append(temp_loss)

Stochastic Training:
Step #5: A = [[9.999702]]; Loss = 1.4709787
Step #10: A = [[9.94747]]; Loss = 0.6618977
Step #15: A = [[9.971459]]; Loss = 0.6653751
Step #20: A = [[10.0871935]]; Loss = 0.49029216
Step #25: A = [[9.912683]]; Loss = 0.17007475
Step #30: A = [[9.906863]]; Loss = 0.00035235687
Step #35: A = [[10.046809]]; Loss = 0.31774017
Step #40: A = [[10.074819]]; Loss = 0.062483788
Step #45: A = [[9.931749]]; Loss = 0.38980645
Step #50: A = [[9.892796]]; Loss = 4.083847
Step #55: A = [[10.008774]]; Loss = 0.49948332
Step #60: A = [[9.972428]]; Loss = 1.3113406
Step #65: A = [[9.940978]]; Loss = 5.449776
Step #70: A = [[9.983938]]; Loss = 0.29599583
Step #75: A = [[10.050091]]; Loss = 0.05119669
Step #80: A = [[10.128044]]; Loss = 0.31691822
Step #85: A = [[10.113219]]; Loss = 0.052486476
Step #90: A = [[10.17403]]; Loss = 1.1689844
Step #95: A = [[10.201076]]; Loss = 0.078442045
Step #100: A = [[10.366297]]; Loss = 0.3529299
```

2.4.3 绘制回归算法的批量训练损失和随机训练损失图

因为篇幅关系，这里仅给出绘图结果（见图 2.5），具体绘图代码请参见代码文件 tensorflow_batch_and_stochastic_training.py。可以看到，批量训练的损失函数整体上为逐渐下降趋势，表现为训练结果逐渐趋向于真实结果，而随机训练损失函数则表现得更为“随机”，这使得训练结果的精度很差，但是随机训练的成本更低。

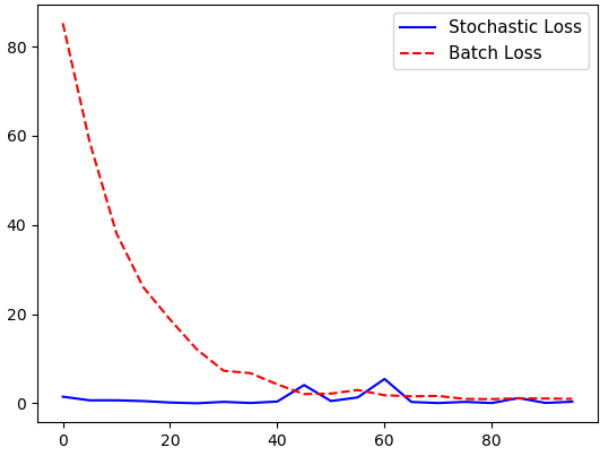

图 2.5 批量训练与随机训练的损失函数随得带次数的变化曲线

2.5 TensorFlow 实现模型评估

模型评估是非常重要的，在训练模型过程中，模型评估能洞察模型算法，给出提示信息来调试、提高或者改变整个模型。不管算法模型预测的如何，我们都需要测试算法模型，在训练数据和测试数据上都要进行模型评估，以搞清楚模型是否过拟合。（本节中的代码文件为 tensorflow_evaluating_models.py）

2.5.1 评估回归算法模型

回归算法模型用来预测连续数值型，其目标不是分类值而是数字。为了评估这些回归预测值是否与实际目标相符，我们需要度量两者间的距离，通常用均方误差（MSE）来表示：

```
# 初始化计算图
ops.reset_default_graph()
sess = tf.Session()
# 创建数据，要拟合的事实上是数据中心点与原点所在直线的斜率
x_vals = np.random.normal(1, 0.1, 100)
y_vals = np.repeat(10., 100)
```

```
# 分割训练数据集和测试数据集
train_indices = np.random.choice(len(x_vals), int(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]
# 创建占位符
x_data = tf.placeholder(shape=[None, 1], dtype=tf.float32)
y_target = tf.placeholder(shape=[None, 1], dtype=tf.float32)
# 声明变量，作为数据中心点与原点所在直线的斜率，理论值为 10.
A = tf.Variable(tf.random_normal(shape=[1,1]))
# 声明批量训练集大小
batch_size = 25
# 在计算图中声明算法模型
my_output = tf.matmul(x_data, A)
# 声明损失函数
loss = tf.reduce_mean(tf.square(my_output - y_target))
# 声明优化器算法
my_opt = tf.train.GradientDescentOptimizer(0.02)
train_step = my_opt.minimize(loss)
# 初始化模型变量 A
init = tf.global_variables_initializer()
# 启动图
sess.run(init)
# 迭代训练模型
print('Evaluate the regression model:')
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)) + '; Loss = ' +
                  str(sess.run(loss, feed_dict={x_data: rand_x, y_target:
rand_y})))
# 得到最佳拟合结果应接近于 A = [10.]
# 打印训练数据集和测试数据集训练的 MSE 损失函数值，以评估训练模型
mse_train = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_train]),
y_target: np.transpose([y_vals_train])})
mse_test = sess.run(loss, feed_dict={x_data: np.transpose([x_vals_test]), y_target:
np.transpose([y_vals_test])})
```

```
print('MSE on train set: ' + str(np.round(mse_train, 2)))
print('MSE on test set: ' + str(np.round(mse_test, 2)))

Evaluate the regression model:
Step #25: A = [[6.3496976]]; Loss = 12.945509
Step #50: A = [[8.57165]]; Loss = 2.742483
Step #75: A = [[9.311588]]; Loss = 1.4431452
Step #100: A = [[9.551025]]; Loss = 1.114175
MSE on train set: 1.06
MSE on test set: 1.56
```

2.5.2 评估分类算法模型

分类算法模型基于数值型输入预测分类值，实际目标是 1 和 0 的序列。我们需要度量预测值与真实值之间的距离。分类算法模型的损失函数一般不容易解释模型好坏，所以通常情况是看下准确预测分类的结果的百分比，从结果可以看到，分类学习结果在训练数据集上的分类准确率为 0.9875，而在测试数据集上的准确率为 0.95，准确率较高，分类模型结果见图 2.6：

```
# 重置计算图
ops.reset_default_graph()
sess = tf.Session()
# 从正态分布 (N(-1,1), N(2,1)) 生成数据，总共 100 个点
x_vals = np.concatenate((np.random.normal(-1, 1, 50), np.random.normal(2, 1,
50)))
# 创建目标标签，N(-1,1) 为 '0' 类，N(2,1) 为 '1' 类，各 50 个点
y_vals = np.concatenate((np.repeat(0., 50), np.repeat(1., 50)))
# 分割训练数据集和测试数据集
train_indices = np.random.choice(len(x_vals), int(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]
# 创建占位符
x_data = tf.placeholder(shape=[1, None], dtype=tf.float32)
y_target = tf.placeholder(shape=[1, None], dtype=tf.float32)
# 创建变量 A 作为最佳聚类边界的负值，初始值为 10 附近的随机数，远离
理论值 -(-1+2)/2 = -0.5
A = tf.Variable(tf.random_normal(mean=10, shape=[1]))
# 声明批量训练集大小
```

```
batch_size = 25
# 在计算图中声明算法模型
my_output = tf.add(x_data, A)
# 声明损失函数
xentropy =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=my_output,
labels=y_target))
# 声明优化器算法
my_opt = tf.train.GradientDescentOptimizer(0.05)
train_step = my_opt.minimize(xentropy)
# 初始化变量 A
init = tf.global_variables_initializer()
sess.run(init)
# 迭代训练
print('\nEvaluate the classification model:')
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)) + '; Loss = ' +
str(sess.run(xentropy, feed_dict={x_data: rand_x, y_target: rand_y})))
# 得到最佳拟合结果应趋近于 A = [-0.5]
# 为了评估训练模型，我们创建预测操作。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。然后用 equal() 函数检测是否相等，
# 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。我们将用这个函数评估训练模型和测试模型。
y_prediction = tf.squeeze(tf.round(tf.nn.sigmoid(tf.add(x_data, A))))
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))

Evaluate the classification model:
Step #200: A = [5.237268]; Loss = 1.8633449
Step #400: A = [1.144051]; Loss = 0.449895
Step #600: A = [-0.13747545]; Loss = 0.25440043
```

```
Step #800: A = [-0.48275685]; Loss = 0.23504467
Step #1000: A = [-0.60453427]; Loss = 0.2488542
Step #1200: A = [-0.60862726]; Loss = 0.23602723
Step #1400: A = [-0.56232816]; Loss = 0.18370804
Step #1600: A = [-0.57881457]; Loss = 0.23135981
Step #1800: A = [-0.5501353]; Loss = 0.20658727
Accuracy on train set: 0.9875
Accuracy on test set: 0.95
```

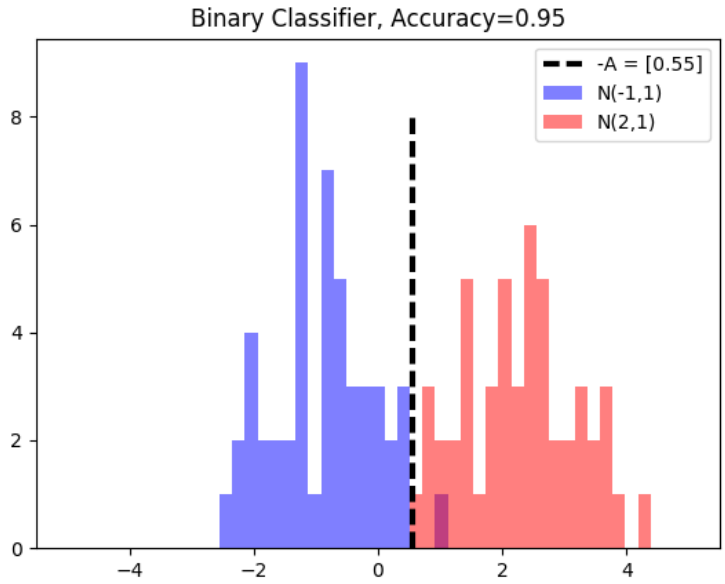

图 2.6 分类模型结果，最佳聚类边界应接近于 0.5

2.6 TensorFlow 的可视化: Tensorboard

Tensorboard 是 TensorFlow 的可视化工具，它可以通过 TensorFlow 程序运行过程中输出的日志文件可视化 TensorFlow 程序的运行状态。Tensorboard 和 TensorFlow 在不同的进程中运行，Tensorboard 会自动读取最新的 TensorFlow 日志文件，并呈现当前 TensorFlow 程序运行的最新状态。

在启动 Tensorboard 可视化工具之前，首先要将 TensorFlow 汇总文件写入到日志文件夹。随后只需在新的命令行终端输入以下命令即可打开 Tensorboard 可视化工具（其中 LOGDIR 表示 TensorFlow 日志文件夹路径），然后在浏览器打开网页 http://localhost:6006 即可查看 TensorFlow 运行状态：

```
tensorboard --logdir LOGDIR --host localhost --port 6006
```

本节以拟合一个 y=2x 的线性回归模型为例，介绍 Tensorboard 可视化工

具的使用。因为篇幅关系，仅给出结果展示，具体代码请参见代码文件为 tensorflow_use_tensorboard.py。

图 2.8 展示了 TensorFlow 计算图的可视化，可以清晰地看到张量在图中的计算过程，对于各个节点也可以放大看到内部计算图。图 2.8 展示了模型斜率估计的可视化，可以看到斜率估计值逐渐趋于 2 的过程。图 2.9 展示了模型误差和残差直方图的可视化，可以清晰地看到模型的误差和残差的均值在逐渐减小，并且误差和残差的分布逐渐变得集中。图 2.10 展示了模型拟合结果的可视化，可以看到，训练的模型较好地拟合了给出的数据点。

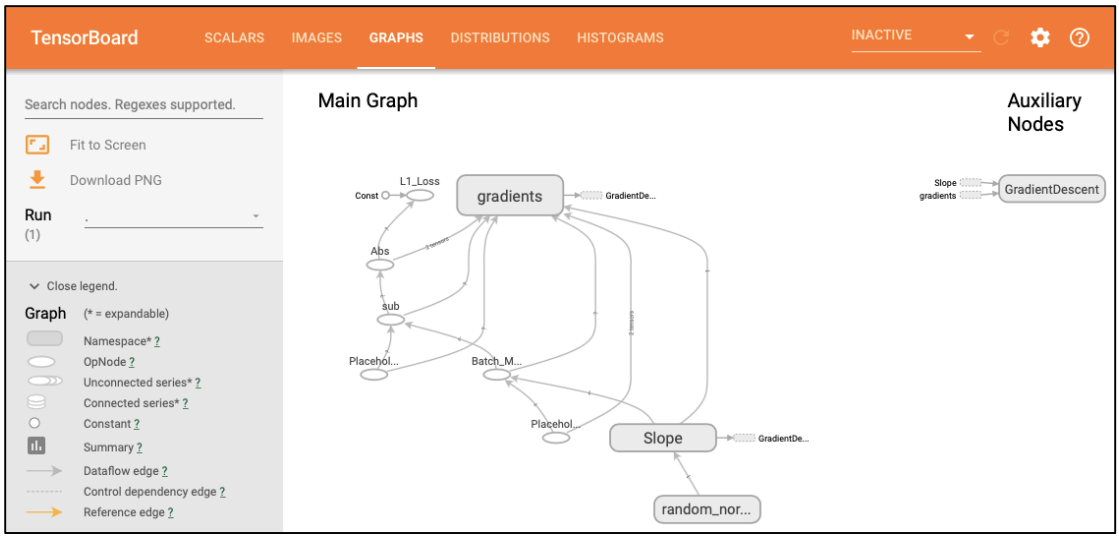

图 2.8 TensorFlow 计算图的可视化

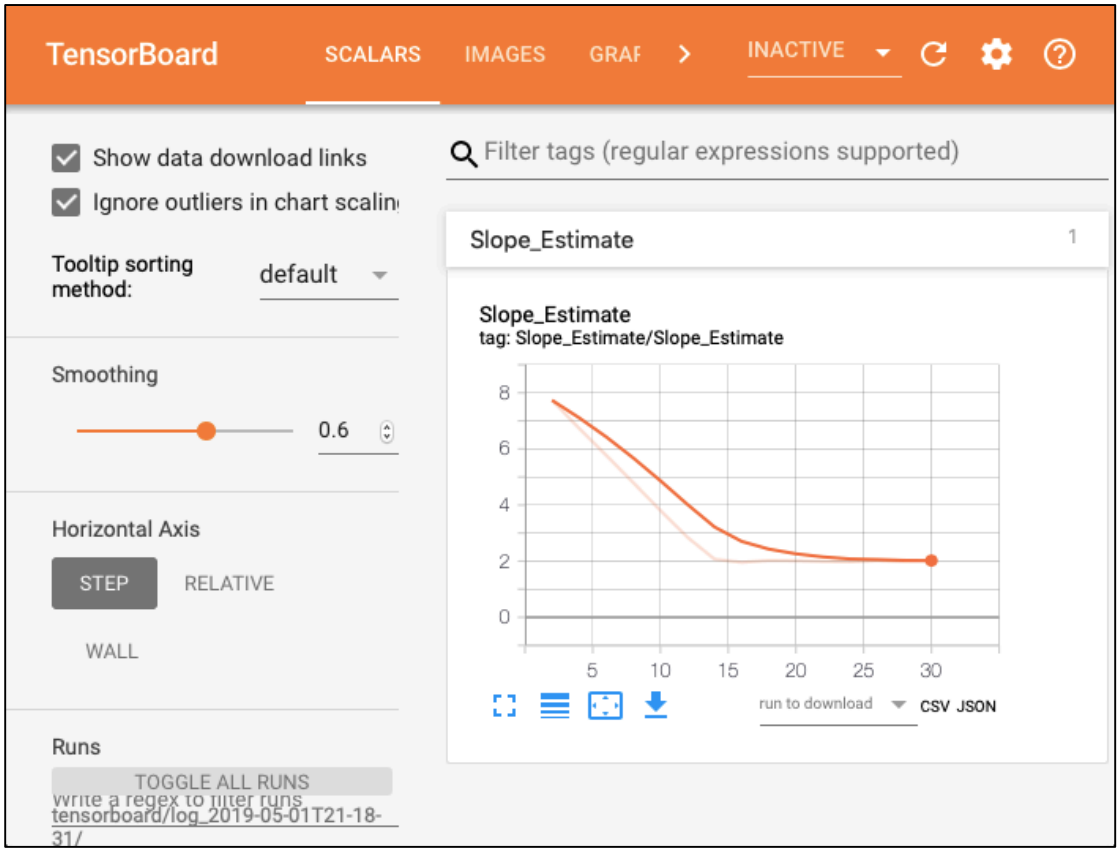

图 2.8 模型斜率估计的可视化

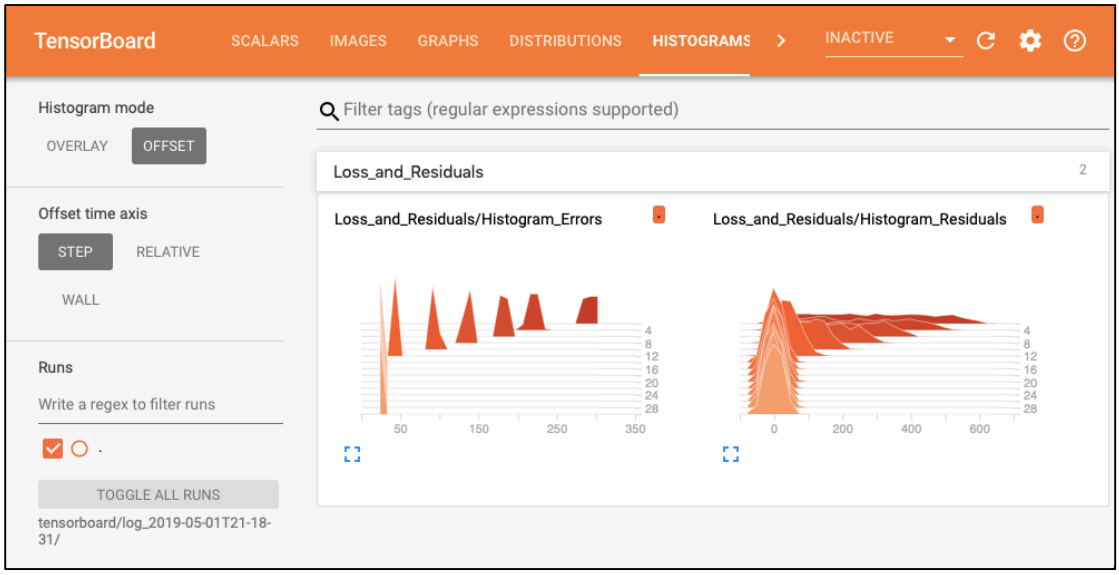

图 2.9 可视化模型的误差直方图（左图）和残差直方图（右图）

图 2.10 可视化模型拟合结果

第三章 TensorFlow 综合应用实例

本章我们给出一个 TensorFlow 应用实例来解决一个在神经网络领域最为经典的问题——MNIST 手写数字分类问题，它将综合前面各章节所述的所有内容，包括获取数据集、创建占位符和变量、批量训练、计算图操作、创建损失函数和优化器、模型评估和使用 Tensorboard 汇总等。

MNIST 是机器学习领域的一个经典问题，指的是让机器查看一系列大小为 28x28 像素的手写数字灰度图像（如图 3.1），并判断这些图像代表 0-9 中的哪一个数字。数据集获取自美国国家标准与技术研究院（National Institute of Standards and Technology，NIST）。

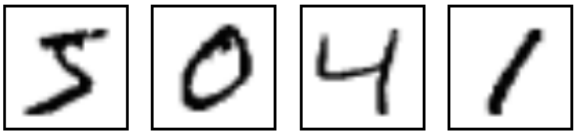

图 3.1 MNIST 手写数字灰度图像

因为篇幅关系，这里仅给出模型训练结果（省略了中间结果）和 Tensorboard 汇总图（见图），具体代码请参见代码文件 tensorflow_mnist.py。可以看到模型的训练误差、学习率和验证数据集的误差均在逐渐减小，最终的测试数据集的误差仅为 1.1%，当然，如果我们调整训练模型的参数，并且迭代更多的次数，这个误差很可能继续减小，但是要付出更多的计算时间。

图 3.2 展示了 TensorFlow 计算图的可视化。图 3.3 展示了模型学习率和损失函数的可视化，可以清晰地看出学习率和损失函数的下降过程。图 3.4 展示了模型部分变量随着迭代逐渐优化的过程的可视化，可以清晰地看到神经网络各层的权重逐渐调整的过程。

```
Step #100 of 2578 (epoch 0.12 of 3): Average time = 149.0 ms
Minibatch loss: 3.282; learning rate: 0.010000
Minibatch error: 4.7%
Validation error: 7.0%

Step #200 of 2578 (epoch 0.23 of 3): Average time = 204.8 ms
Minibatch loss: 3.388; learning rate: 0.010000
Minibatch error: 12.5%
Validation error: 3.8%

……
```

```
Step #2400 of 2578 (epoch 2.79 of 3): Average time = 168.4 ms
Minibatch loss: 2.516; learning rate: 0.009025
Minibatch error: 1.6%
Validation error: 1.1%

Step #2500 of 2578 (epoch 2.91 of 3): Average time = 171.7 ms
Minibatch loss: 2.482; learning rate: 0.009025
Minibatch error: 0.0%
Validation error: 1.3%

Step #2578 of 2578 (epoch 3.00 of 3): Average time = 139.0 ms
Minibatch loss: 2.515; learning rate: 0.009025
Minibatch error: 3.1%
Validation error: 1.1%

End of the training.
Test error: 1.1%
```

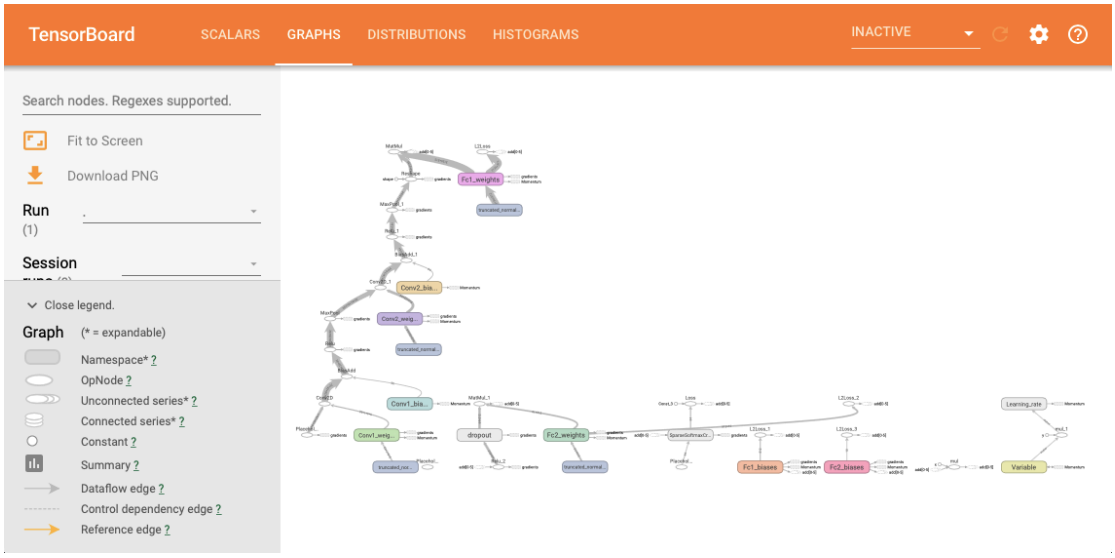

图 3.2 TensorFlow 计算图的可视化

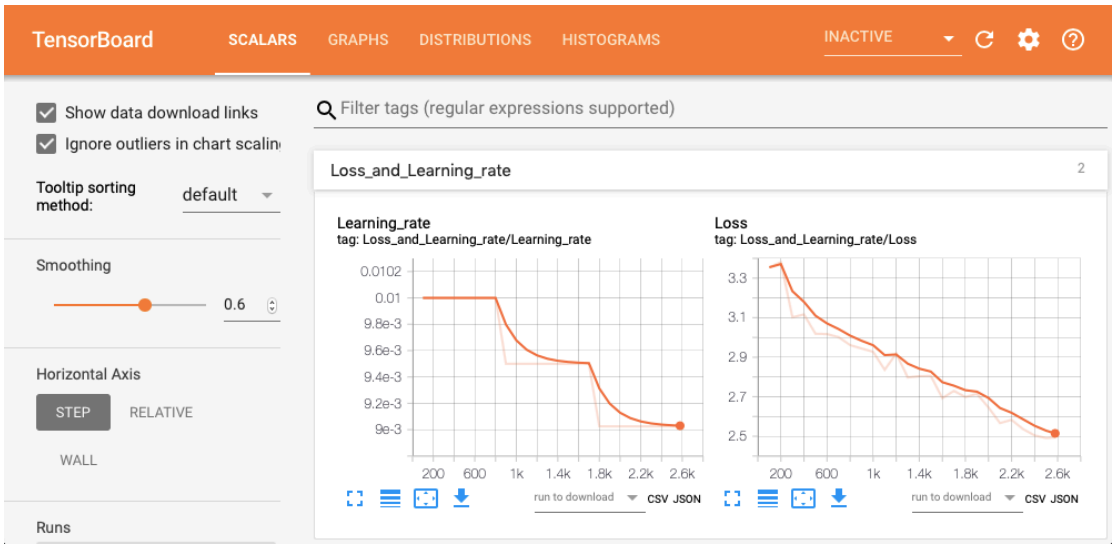

图 3.3 模型学习率（左图）和损失函数（右图）的可视化

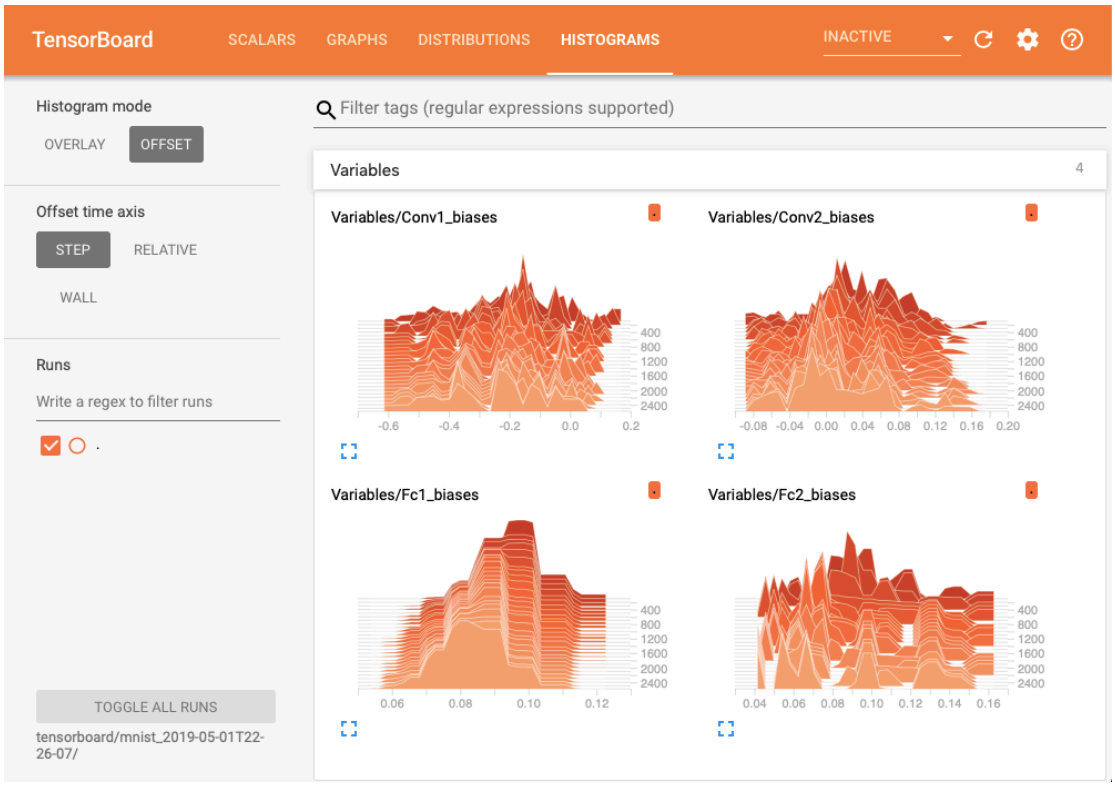

图 3.4 模型部分变量随着迭代逐渐优化的过程的可视化

结语

本文介绍了 TensorFlow 的基础知识，包括 TensorFlow 中张量、占位符、变量、矩阵和激励函数的创建与基本操作方法，以及如何通过 TensorFlow 和 Python 访问各种数据源等。

同时也简单介绍了 TensorFlow 计算图的有关操作，包括 TensorFlow 计算图中对象和层的操作、损失函数的实现、反向传播的实现、批量训练和随机训练的实现、模型评估的实现以及 Tensorboard 可视化操作等。

最后给出了一个 TensorFlow 的具体应用实例，实现了对 MNIST 手写数字分类问题模型的训练。

通过这些基本操作的学习，我们可以发现 TensorFlow 算法的一般流程为：

（1）获取训练数据集和测试数据集；

（2）将原始训练数据集分割为验证数据集和新的训练数据集；

（3）创建占位符和变量；

（4）定义模型结构；

（5）定义损失函数和优化器；

（6）创建 Tensorboard 汇总；

（7）创建模型评估函数；

（8）初始化模型；

（9）通过迭代训练模型并将 Tensorboard 汇总写入日志文件。

参考文献

McClure, Nick. *TensorFlow Machine Learning Cookbook.* Packt Publishing Ltd. 2017.

黄文坚，唐源. *TensorFlow 实战*. 电子工业出版社. 2017.