

目录

1	deeplearning	1
1.1	降维	1
1.1.1	自编码	1
1.1.2	自动降噪编码	1
1.1.3	手写体数据自编码	2
1.2	稀疏编码	7
1.2.1	稀疏编码的概率表示	8
1.3	PCA	10
1.3.1	数学定义	10
1.4	KL 散度	11
1.4.1	交叉熵	13
1.4.2	相对熵	13
2	Tensorflow 基础	15
2.1	Tensorflow 基础概念	15
2.1.1	Rank	15
2.1.2	获取 Tensor 对象的 rank	16
2.1.3	Tensor 的切片	16
2.1.4	形状	17
2.1.5	获取 tf.Tensor 对象的形状	17
2.1.6	改变 Tensor 的形状	17
2.1.7	数据类型	18
2.1.8	计算 Tensor	18
2.1.9	打印 Tensor	19
2.2	Variable	19
2.2.1	创建变量	19
2.2.2	变量集合	20

2.2.3 配置设备	20
2.2.4 初始化变量	21
2.2.5 用变量	21
2.2.6 保存和恢复	22
2.2.7 checkpoint 文件	22
2.2.8 保存变量	22
2.2.9 恢复变量	23
2.2.10 选择变量恢复	23
2.2.11 共享变量	24
2.3 图 (Graphs) 和会话 (Session)	26
2.3.1 为什么用数据流图?	26
2.3.2 建立一个 tf.Graph	26
2.3.3 命名你的操作	27
2.3.4 放置操作在不同的设备上	27
2.3.5 Tensor-like 对象	29
2.3.6 在 tf.Session 执行图	29
2.3.7 创建 tf.Session	29
2.3.8 用 tf.Session.run 执行操作	30
2.3.9 GraphDef 和 MetaGraphDef	31
2.3.10 可视化你的图	32
2.3.11 用多图编程	35
2.4 数据集	36
2.4.1 基本的机制	36
2.4.2 数据结构	36
2.4.3 创建一个迭代器	37
2.4.4 消耗迭代器的值	40
2.4.5 读输入数据	41
2.4.6 消耗 TFRecord 数据	42
2.4.7 用 Dataset.map() 处理数据	42
2.4.8 解析 tf.Example protocol buffer 消息	43
2.4.9 解码图像数据变换大小	43
2.4.10 用专门的 Python logic	43
2.4.11 简单的批处理	44
2.4.12 批量的 tensorpadding	45
2.4.13 处理多 epoch	45

目录	iii
2.4.14 随机打乱输入数据	46
2.4.15 用高级 APIs	46
2.5 线程和队列	48
2.5.1 队列用法	48
2.5.2 手动线程管理	50
2.5.3 Coordinator	50
2.5.4 QueueRunner	51
2.5.5 处理异常	52
2.6 tf.estimator 快速导航	53
2.6.1 完成神经网络源代码	53
2.6.2 载入 CSV 数据进入 TensorFlow	55
2.6.3 构造神经网络分类器	57
2.6.4 描述训练的输入 pipline	58
2.6.5 为 iris 训练集拟合 DNNClassifier	58
2.6.6 评估模型的精度	58
2.6.7 分类新的样本	59
2.7 用 tf.estimator 创建一个输入函数	60
2.7.1 用 input_fn 自定义 Pipeline	60
2.7.2 input_fn 的分解	60
2.7.3 转换特征数据为 Tensor	61
2.7.4 传递 input_fn 数据到你的模型	62
2.7.5 波士顿房价的神经网络模型	63
2.7.6 建立	63
2.7.7 导入的房子数据	64
2.7.8 定义特征列创建回归器	64
2.7.9 构建 input_fn	65
2.7.10 训练回归器	65
2.7.11 评估模型	65
2.7.12 做出预测	66
2.7.13 batch normalization	66
2.8 常见的激活函数	67
2.8.1 relu	67
2.8.2 relu6	68
2.8.3 sigmoid	70
2.8.4 relu 和 softplus	71

2.8.5 dropout	73
2.9 CNN 常用函数	73
2.9.1 卷积函数	73
2.9.2 常见的分类函数	74
2.10 优化方法	75
2.10.1 BGD	75
2.10.2 SGD	75
2.10.3 momentum	76
2.10.4 Nesterov Momentum	76
2.10.5 Adagrad	76
2.10.6 RMSprop	76
2.10.7 Adam	76
2.10.8 构造简单的神经网络拟合数据	78
2.11 TensorBoard	80
2.11.1 TensorBoard Histogram Dashboard	82
2.11.2 一个简单的例子	82
2.11.3 Overlay Mode	85
2.11.4 多个分布	86
2.11.5 更多分布	88
2.11.6 poisson 分布	91
2.11.7 结合所有的数据到一张图向上	92
2.12 CNN 手写体数据识别	93
2.12.1 mnist 数据集	93
2.13 RNN	97
2.13.1 The Problem Long-Term Dependencies	98
2.13.2 LSTM 网络	98
2.13.3 LSTMs 想法的核心	99
2.13.4 一步步的设置	99
2.13.5 LSTM 的多种变体	100
2.13.6 向量字表示	102
2.13.7 RNN	110
3 Tensorflow 进阶	115
3.1 模型存储和加载	115
3.2 用 GPU	116
3.2.1 手工配置设备	116

3.2.2 允许 GPU 的内存增长	117
3.3 如何利用 Inception 的最后一层重新训练新的分类	119
3.3.1 训练花	119
3.3.2 瓶颈	120
3.3.3 训练	120
3.3.4 用 TensorBoard 可视化	121
3.3.5 用重新训练的模型	121
3.3.6 在你自己的分类上训练	121
3.3.7 创建一个训练图像集合	122
3.3.8 训练步骤	122
3.3.9 扭曲	122
3.3.10 超参数	123
3.3.11 训练, 验证, 测试集	123
3.3.12 更对模型架构	124
3.4 TF layer 向导: 建立一个卷积神经网络	124
3.4.1 开始	124
3.4.2 介绍卷积神经网络	125
3.4.3 建立 CNN MNIST 分类器	125
3.4.4 输入层	126
3.4.5 第一层卷积层	126
3.4.6 池化层 1	127
3.4.7 二层卷积和池化	127
3.4.8 Dense layer	128
3.4.9 Logits Layers	128
3.4.10 常见的预测	129
3.4.11 计算 Loss	129
3.4.12 配置训练操作	130
3.4.13 增加评估度量	130
3.5 训练评估 CNN MNIST 分类器	130
3.5.1 载入训练和测试数据	131
3.5.2 创建 Estimator	131
3.5.3 建立 Logging Hook	131
3.5.4 选练模型	132
3.5.5 评估模型	132
3.5.6 运行模型	132

4 扩展	135
4.1 TensorFlow 架构	135
4.2 概述	135
4.2.1 Client	137
4.2.2 Distributed master	137
4.2.3 Worker Service	138
4.3 内核实现	139
5 Performance	141
5.1 最好的实践	141
5.2 从源代码创建安装	141
5.2.1 利用队列读取数据	142
5.2.2 在 CPU 上的预处理	142
5.2.3 用大文件	143
5.2.4 用 NCHW 图像数据格式	143
5.2.5 用融批规范	143
5.3 性能向导	144
5.4 好性能模型	144
5.5 Benchmark	144
5.6 如何用 TensorFlow 量化神经网络	144
5.6.1 为什么做量化工作	144
5.6.2 为什么量化	144
5.6.3 为什么不直接训练低精度	145
5.6.4 你能如何量化你的模型	145
5.6.5 如何量化处理工作	146
5.6.6 量化 Tensor 将呈现什么	148
5.6.7 下一步	149
6 常用的 python 模块	151
6.1 Argparse	151
6.1.1 ArgumentParser 对象	152
6.1.2 prog	152
6.1.3 add_argument() 方法	157
6.2 path	185
6.2.1 函数说明	185
6.2.2 例子	187

6.2.3 常见问题	188
6.3 正则表达式介绍	198
6.4 RE 库的主要功能函数	201
6.4.1 re 表达式中的 flags	203
6.5 常用的 sys 函数	208
6.6 collections	215
6.6.1 namedtuple	215
6.6.2 deque	215
6.6.3 defaultdict	216
6.6.4 OrderedDict	216
6.6.5 Counter	217
6.7 base64	218
6.8 struct	220
6.9 hashlib	221
6.10 itertools	223
6.10.1 cycle	223
6.10.2 chain()	224
6.11 contextlib	225
6.12 XML	228
6.13 HTMLParser	229
6.14 ZipFile	230
6.15 url	231
6.15.1 urllib.request	231
6.16 requests	232
6.16.1 发送请求	232
6.16.2 requests 库的 7 个主要方法	232
6.16.3 request 对象的属性	232
6.16.4 理解 encoding 和 apparent_encoding	233
6.16.5 理解 Requests 库的异常	233
6.16.6 HTTP 协议	233
7 Bazel	237
7.1 Bazel start	237
7.1.1 用工作空间	237

8 Tensorflow 技巧	239
8.1 文件读取	239
9 Tensorflow API	241
9.1 tf.app.flags	241
9.1.1 DEFINE_boolean	241
9.1.2 DEFINE_boolean	241
9.1.3 DEFINE_float	241
9.1.4 DEFINE_integer	242
9.1.5 DEFINE_string	242
9.1.6 tf.gather	242
9.1.7 tf.placeholder	243
9.1.8 tf.squeeze	244
9.1.9 tf.metrics	245
9.1.10 tf.stack	245
9.1.11 tf.reshape	246
9.1.12 tf.random_crop	247
9.1.13 tf.random_gamma	247
9.1.14 tf.random_normal	248
9.1.15 tf.random_normal_initializer	249
9.1.16 tf.random_possion	250
9.1.17 random_shuffle	251
9.1.18 tf.random_uniform	251
9.1.19 tf.random_uniform_initializer	252
9.1.20 tf.one_hot	253
9.1.21 tf.unstack	255
9.2 tf.Vairable	257
9.2.1 Variable 类	257
9.2.2 方法	258
9.2.3 参数	264
9.3 tf.image	271
9.3.1 tf.image.decode_gif	271
9.3.2 tf.image.decode_jpeg	271
9.3.3 tf.image.encode_jpeg	272
9.3.4 tf.image.decode_png	272
9.3.5 tf.image.encode_png	273

9.3.6	tf.image.decode_image	273
9.3.7	tf.image.resize_images	273
9.4	layer	275
9.4.1	tf.layers.average_pooling1d	275
9.4.2	tf.layers.average_pooling2d	275
9.4.3	tf.layers.average_pooling3d	276
9.4.4	tf.layers.batch_normalization	277
9.4.5	conv1d	279
9.4.6	conv2d	280
9.4.7	conv2d_transpose	282
9.4.8	conv3d	283
9.4.9	conv3d_transpose	285
9.4.10	dense	286
9.4.11	dropout	287
9.4.12	max_pool1d	288
9.4.13	max_pool2d	288
9.4.14	max_pool3d	289
9.4.15	separable_conv2d	290
9.5	tf.train	292
9.5.1	优化器	292

Chapter 1

deeplearning

1.1 降维

1.1.1 自编码

人工神经网络（ANN）本身就是具有层次结构的系统，如果给定一个神经网络，我们假设其输出与输入是相同的，然后训练调整其参数，得到每一层中的权重。自然地，我们就得到了输入 I 的几种不同表示（每一层代表一种表示），这些表示就是特征。在研究中可以发现，如果在原有的特征中加入这些自动学习得到的特征可以大大提高精确度，甚至在分类问题中比目前最好的分类算法效果还要好！这种方法称为 AutoEncoder（自动编码器）。自动编码器就是一种尽可能复现输入信号的神经网络。为了实现这种复现，自动编码器就必须捕捉可以代表输入数据的最重要的因素，就像 PCA 那样，找到可以代表原信息的主要成分。我们将 input 输入一个 encoder 编码器，就会得到一个 code，这个 code 也就是输入的一个表示，那么我们怎么知道这个 code 表示的就是 input 呢？我们加一个 decoder 解码器，这时候 decoder 就会输出一个信息，那么如果输出的这个信息和一开始的输入信号 input 是很像的（理想情况下就是一样的），那很明显，我们就有理由相信这个 code 是靠谱的。所以，我们就通过调整 encoder 和 decoder 的参数，使得重构误差最小，这时候我们就得到了输入 input 信号的第一个表示了，也就是编码 code 了。因为是无标签数据，所以误差的来源就是直接重构后与原输入相比得到。

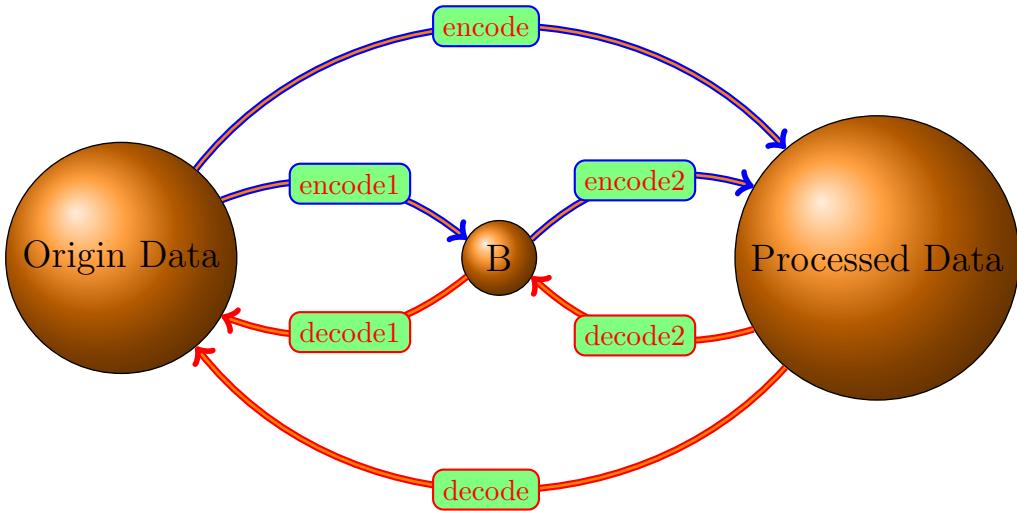
1.1.2 自动降噪编码

以一定的概率分布擦出原始数据（将数据置为 0），这样操作后的数据称为破损失数据，这样的数据有两个作用：

1. 通过破损失数据和非破损失数据相比，破损失数据训练出来的权重噪声小（可能不小心删除了噪声）。

2. 破损数据一定程度上减轻了训练数据和测试数据之间的代沟。由于数据部分被擦除，因而训练出来的权重的健壮性就提高了。

1.1.3 手写体数据自编码



```

1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3 data_path = '/home/hpc/文档/mnist_tutorial/mnist'
4
5 from tensorflow.examples.tutorials.mnist import input_data
6 mnist = input_data.read_data_sets(data_path, one_hot=False)
7
8
9 # Visualize decoder setting
10 learning_rate = 0.01
11 training_epochs = 5
12 batch_size = 256
13 display_step = 1
14 examples_to_show = 10
15
16 n_input = 784 # MNIST data input (img shape: 28*28)
17
18 x = tf.placeholder(tf.float32, [None, n_input])
19
20 n_hidden_1 = 256
21 n_hidden_2 = 128
22 weights = {
23     'encode_h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
  
```

```

24     'encode_h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),  

25     'decode_h2': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_1])),  

26     'decode_h1': tf.Variable(tf.random_normal([n_hidden_1, n_input]))  

27 }  

28 bias = { 'encode_h1': tf.Variable(tf.random_normal([n_hidden_1])),  

29         'encode_h2': tf.Variable(tf.random_normal([n_hidden_2])),  

30         'decode_h2': tf.Variable(tf.random_normal([n_hidden_1])),  

31         'decode_h1': tf.Variable(tf.random_normal([n_input]))  

32 }  

33 }  

34 def encode(x):  

35     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encode_h1']), bias[  

36                                     'encode_h1']))  

37     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['encode_h2']), bias[  

38                                     'encode_h2']))  

39     return layer_2  

40  

41 def decode(x):  

42     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['decode_h2']), bias[  

43                                     'decode_h2']))  

44     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['decode_h1']), bias[  

45                                     'decode_h1']))  

46     return layer_2  

47  

48 encode_op = encode(x)  

49 decode_op = decode(encode_op)  

50 y_pred = decode_op  

51 y_true = x  

52 cost = tf.reduce_mean(tf.square(y_pred - y_true))  

53 optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)  

54  

55 with tf.Session() as sess:  

56     init = tf.global_variables_initializer()  

57     sess.run(init)  

58     total_batch = int(mnist.train.num_examples/batch_size)  

59     for epoch in range(training_epochs):  

60         for i in range(total_batch):  

61             batch_xs, batch_ys = mnist.train.next_batch(batch_size)  

62             _, c = sess.run([optimizer, cost], feed_dict={x: batch_xs})  

63             if epoch%display_step==0:  

64                 print("Epoch:", '%04d' % (epoch+1), 'cost=', '{:.9f}'.format(c))

```

```

63     print('Optimize finish')
64     encode_decode = sess.run(y_pred, feed_dict={x: mnist.test.images[:, examples_to_show]})
65     f, a = plt.subplots(2, 10, figsize=(10, 2))
66     for i in range(examples_to_show):
67         a[0][i].imshow(sess.run(tf.reshape(mnist.test.images[i], [28, 28])))
68         a[1][i].imshow(sess.run(tf.reshape(encode_decode[i], [28, 28])))
69     plt.savefig('auto_encode.png', dpi=800)

```

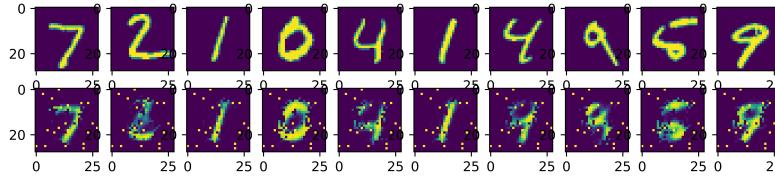


图 1.1: 原图和自编码解码后的图像

编码器输出可视化:

```

1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3
4 from tensorflow.examples.tutorials.mnist import input_data
5 path = '/home/hpc/文档/mnist_tutorial/mnist'
6 mnist = input_data.read_data_sets(path, one_hot=False)
7
8 learning_rate = 0.01
9 training_epochs = 5
10 batch_size = 256
11 display_step = 1
12 examples_to_show = 10
13
14 n_input = 784 # MNIST data input (img shape: 28*28)
15
16 X = tf.placeholder("float", [None, n_input])
17
18 n_hidden_1 = 256 # 1st layer num features
19 n_hidden_2 = 128 # 2nd layer num features
20
21 learning_rate = 0.01 # 0.01 this learning rate will be better! Tested
22 training_epochs = 10

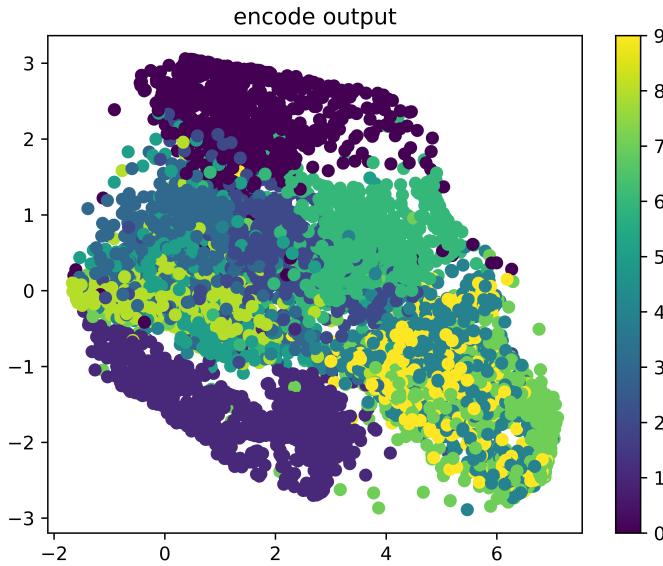
```



```

66 layer_3 = tf.nn.sigmoid(tf.add(tf.matmul(layer_2, weights[ 'decoder_h3' ]),
67                         biases[ 'decoder_b3' ]))
68 layer_4 = tf.nn.sigmoid(tf.add(tf.matmul(layer_3, weights[ 'decoder_h4' ]),
69                         biases[ 'decoder_b4' ]))
70 return layer_4
71
72 encoder_op = encoder(X)
73 decoder_op = decoder(encoder_op)
74
75 y_pred = decoder_op
76 y_true = X
77
78 cost = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
79 optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
80
81
82 with tf.Session() as sess:
83     init = tf.global_variables_initializer()
84     sess.run(init)
85     total_batch = int(mnist.train.num_examples/batch_size)
86     for epoch in range(training_epochs):
87         for i in range(total_batch):
88             batch_xs, batch_ys = mnist.train.next_batch(batch_size) # max(x) =
89                                     # min(x) = 0
90             _, c = sess.run([optimizer, cost], feed_dict={X: batch_xs})
91             if epoch % display_step == 0:
92                 print("Epoch:", "%04d" % (epoch+1),
93                       "cost=", "{:.9f}".format(c))
94     print("Optimization Finished!")
95 encode_decode = sess.run(
96     y_pred, feed_dict={X: mnist.test.images[:examples_to_show] })
97 encoder_result = sess.run(encoder_op, feed_dict={X: mnist.test.images})
98 plt.scatter(encoder_result[:, 0], encoder_result[:, 1], c=mnist.test.labels)
99 plt.title('encode output')
100 plt.colorbar()
101 plt.savefig('auto_encode_v.png', dpi=800)

```



1.2 稀疏编码

稀疏编码算法是一种无监督学习方法，它用来寻找一组“超完备”基向量来更高效地表示样本数据。稀疏编码算法的目的就是找到一组基向量 ϕ_i ，使得我们能将输入向量 \mathbf{x} 表示为这些基向量的线性组合：

$$\mathbf{x} = \sum_{i=1}^k a_i \phi_i \quad (1.1)$$

虽然形如主成分分析技术（PCA）能使我们方便地找到一组“完备”基向量，但是这里我们想要做的是找到一组“超完备”基向量来表示输入向量 $\mathbf{x} \in \mathbb{R}^n$ （也就是说， $k > n$ ）。超完备基的好处是它们能更有效地找出隐含在输入数据内部的结构与模式。然而，对于超完备基来说，系数 a_i 不再由输入向量 \mathbf{x} 唯一确定。因此，在稀疏编码算法中，我们另加了一个评判标准“稀疏性”来解决因超完备而导致的退化（degeneracy）问题。

这里，我们把“稀疏性”定义为：只有很少的几个非零元素或只有很少的几个远大于零的元素。要求系数 a_i 是稀疏的意思就是说：对于一组输入向量，我们只想有尽可能少的几个系数远大于零。选择使用具有稀疏性的分量来表示我们的输入数据是有原因的，因为绝大多数的感官数据，比如自然图像，可以被表示成少量基本元素的叠加，在图像中这些基本元素可以是面或者线。同时，比如与初级视觉皮层的类比过程也因此得到了提升。

我们把有 m 个输入向量的稀疏编码代价函数定义为：

$$\underset{a_i^{(j)}, \phi_i}{\text{minimize}} \sum_{j=1}^m \left\| \mathbf{x}^{(j)} - \sum_{i=1}^k a_i^{(j)} \phi_i \right\|^2 + \lambda \sum_{i=1}^k S(a_i^{(j)}) \quad (1.2)$$

此处 $S(\cdot)$ 是一个稀疏代价函数，由它来对远大于零的 a_i 进行“惩罚”。我们可以把稀疏编码目标函式的第一项解释为一个重构项，这一项迫使稀疏编码算法能为输入向量 \mathbf{x} 提供一个高拟合度的线性表达式，而公式第二项即“稀疏惩罚”项，它使 \mathbf{x} 的表达式变得“稀疏”。常量 λ 是一个变换量，由它来控制这两项式子的相对重要性。

虽然“稀疏性”的最直接测度标准是”L0”范式 ($S(a_i) = \mathbf{1}(|a_i| > 0)$)，但这是不可微分的，而且通常很难进行优化。在实际中，稀疏代价函数 $S(\cdot)$ 的普遍选择是 L1 范式代价函数 $S(a_i) = |a_i|_1$ 及对数代价函数 $S(a_i) = \log(1 + a_i^2)$ 。

此外，很有可能因为减小 a_i 或增加 ϕ_i 至很大的常量，使得稀疏惩罚变得非常小。为防止此类事件发生，我们将限制 $\|\phi\|^2$ 要小于某常量 C 。包含了限制条件的稀疏编码代价函数的完整形式如下：

$$\underset{a_i^{(j)}, \phi_i}{\text{minimize}} \sum_{j=1}^m \|x^{(j)} - \sum_{i=1}^k a_i^{(j)} \phi_i\|^2 + \lambda \sum_{i=1}^k S(a_i^{(j)}) \quad \|\phi_i\|^2 \leq C, \forall i = 1, \dots, k \quad (1.3)$$

1.2.1 稀疏编码的概率表示

到目前为止，我们所考虑的稀疏编码，是为了寻找到一个稀疏的、超完备基向量集，来覆盖我们的输入数据空间。现在换一种方式，我们可以从概率的角度出发，将稀疏编码算法当作一种“生成模型”。

我们将自然图像建模问题看成是一种线性叠加，叠加元素包括 k 个独立的源特征 ϕ_i 以及加性噪声：

$$\mathbf{x} = \sum_{i=1}^k a_i \phi_i + v(\mathbf{x}) \quad (1.4)$$

我们的目标是找到一组特征基向量 ϕ ，它使得图像的分布函数 $P(\mathbf{x} | \phi)$ 尽可能地近似于输入数据的经验分布函数 $P^*(\mathbf{x})$ 。一种实现方式是，最小化 $P^*(\mathbf{x})$ 与 $P(\mathbf{x} | \phi)$ 之间的 KL 散度，此 KL 散度表示如下：

$$D(P^*(\mathbf{x}) || P(\mathbf{x} | \phi)) = \int P^*(\mathbf{x}) \log \left(\frac{P^*(\mathbf{x})}{P(\mathbf{x} | \phi)} \right) d\mathbf{x} \quad (1.5)$$

因为无论我们如何选择 ϕ ，经验分布函数 $P^*(\mathbf{x})$ 都是常量，也就是说我们只需要最大化对数似然函数 $P(\mathbf{x} | \phi)$ 。假设 v 是具有方差 σ^2 的高斯白噪声，则有下式：

$$P(\mathbf{x} | \mathbf{a}, \phi) = \frac{1}{Z} \exp \left(-\frac{(\mathbf{x} - \sum_{i=1}^k a_i \phi_i)^2}{2\sigma^2} \right) \quad (1.6)$$

为了确定分布 $P(\mathbf{x} | \phi)$ ，我们需要指定先验分布 $P(\mathbf{a})$ 。假定我们的特征变量是独立的，我们就可以将先验概率分解为：

$$P(\mathbf{a}) = \prod_{i=1}^k P(a_i) \quad (1.7)$$

此时，我们将“稀疏”假设加入进来——假设任何一幅图像都是由相对较少的一些源特征组合起来的。因此，我们希望 a_i 的概率分布在零值附近是凸起的，而且峰值很高。一个方便的参数化先验分布就是：

$$P(a_i) = \frac{1}{Z} \exp(-\beta S(a_i)) \quad (1.8)$$

这里 $S(a_i)$ 是决定先验分布的形状的函数。

当定义了 $P(\mathbf{x} | \mathbf{a}, \phi)$ 和 $P(\mathbf{a})$ 后，我们就可以写出在由 ϕ 定义的模型之下的数据 \mathbf{x} 的概率分布：

$$P(\mathbf{x} | \phi) = \int P(\mathbf{x} | \mathbf{a}, \phi) P(\mathbf{a}) d\mathbf{a} \quad (1.9)$$

那么，我们的问题就简化为寻找：

$$\phi^* = \operatorname{argmax}_\phi \langle \log(P(\mathbf{x} | \phi)) \rangle \quad (1.10)$$

这里 $\langle \cdot \rangle$ 表示的是输入数据的期望值。

不幸的是，通过对 \mathbf{a} 的积分计算 $P(\mathbf{x} | \phi)$ 通常是难以实现的。虽然如此，我们注意到如果 $P(\mathbf{x} | \phi)$ 的分布（对于相应的 \mathbf{a} ）足够陡峭的话，我们就可以用 $P(\mathbf{x} | \phi)$ 的最大值来估算以上积分。估算方法如下：

$$\phi^{*'} = \operatorname{argmax}_\phi \langle \max_{\mathbf{a}} \log(P(\mathbf{x} | \phi)) \rangle \quad (1.11)$$

跟之前一样，我们可以通过减小 a_i 或增大 ϕ 来增加概率的估算值（因为 $P(a_i)$ 在零值附近陡升）。因此我们要对特征向量 ϕ 加一个限制以防止这种情况发生。最后，我们可以定义一种线性生成模型的能量函数，从而将原先的代价函数重新表述为：

$$E(x, a | \phi) := -\log(P(x | \phi, \mathbf{a}) P(\mathbf{a})) \quad (1.12)$$

$$= \sum_{j=1}^m \|x^{(j)} - \sum_{i=1}^k a_i^{(j)} \phi_i\|^2 + \lambda \sum_{i=1}^k S(a_i^{(j)}) \quad (1.13)$$

其中 $\lambda = 2\sigma 2\beta$ ，并且关系不大的常量已被隐藏起来。因为最大化对数似然函数等同于最小化能量函数，我们就可以将原先的优化问题重新表述为：

$$\phi^*, \mathbf{a}^* = \operatorname{argmin}_{\phi, \mathbf{a}} \sum_{j=1}^m \left\| \mathbf{x}^{(j)} - \sum_{i=1}^k a_i^{(j)} \phi_i \right\|^2 + \lambda \sum_{i=1}^k S(a_i^{(j)}) \quad (1.14)$$

使用概率理论来分析，我们可以发现，选择 L1 惩罚和 $\log(1 + a_i^2)$ 惩罚作为函数 $S(\cdot)$ ，分别对应于使用了拉普拉斯概率 $P(a_i) \propto \exp(-\beta|a_i|)$ 和柯西先验概率 $P(a_i) \propto \frac{\beta}{1+a_i^2}$ 。

1.3 PCA

在多元统计分析中，主成分分析（英语：Principal components analysis, PCA）是一种分析、简化数据集的技术。主成分分析经常用于减少数据集的维数，同时保持数据集中的对方差贡献最大的特征。这是通过保留低阶主成分，忽略高阶主成分做到的。这样低阶成分往往能够保留住数据的最重要方面。但是，这也不是一定的，要视具体应用而定。由于主成分分析依赖所给数据，所以数据的准确性对分析结果影响很大。主成分分析由卡尔·皮尔逊于 1901 年发明，用于分析数据及建立数理模型。其方法主要是通过对协方差矩阵进行特征分解，以得出数据的主成分（即特征向量）与它们的权值（即特征值）。PCA 是最简单的以特征量分析多元统计分布的方法。其结果可以理解为对原数据中的方差做出解释：哪一个方向上的数据值对方差的影响最大？换而言之，PCA 提供了一种降低数据维度的有效办法；如果分析者在原数据中除掉最小的特征值所对应的成分，那么所得的低维度数据必定是最优化的（也即，这样降低维度必定是失去讯息最少的方法）。主成分分析在分析复杂数据时尤为有用，比如人脸识别。PCA 是最简单的以特征量分析多元统计分布的方法。通常情况下，这种运算可以被看作是揭露数据的内部结构，从而更好的解释数据的变量的方法。如果一个多元数据集能够在一个高维数据空间坐标系中被显现出来，那么 PCA 就能够提供一幅比较低维度的图像，这幅图像即为在讯息最多的点上原对象的一个‘投影’。这样就可以利用少量的主成分使得数据的维度降低了。PCA 跟因子分析密切相关，并且已经有很多混合这两种分析的统计包。而真实要素分析则是假定底层结构，求得微小差异矩阵的特征向量。

1.3.1 数学定义

PCA 的数学定义是：一个正交化线性变换，把数据变换到一个新的坐标系统中，使得这一数据的任何投影的第一大方差在第一个坐标（称为第一主成分）上，第二大方差在第二个坐标（第二主成分）上，依次类推。定义一个 $n \times m$ 的矩阵， X^T 为去平均值（以平均值为中心移动至原点）的数据，其行为数据样本，列为数据类别（注意，这里定义的是 X^T 而不是 X ）。则 X 的奇异值分解为 $X = W\Sigma V^T$ ，其中 $m \times m$ 矩阵 W 是 XX^T 的本征矢量矩阵， Σ 是 $m \times n$ 的非负矩形对角矩阵， V 是 $m \times n$ 的 $X^T X$ 的本征矢量矩阵。据此，

$$\begin{aligned} Y^T &= X^T W \\ &= V \Sigma^T W^T W \\ &= V \Sigma^T \end{aligned} \tag{1.15}$$

当 $m < n$ 时， V 在通常情况下不是唯一定义的，而 Y 则是唯一定义的。 W 是一个正交矩阵， Y^T 是 X^T 的转置，且 Y^T 的第一列由第一主成分组成，第二列由第二主成分组成，依次类推。为了得到一种降低数据维度的有效办法，我们可以利用 W_L 把 X 映射到一个只应

用前面 L 个向量的低维空间中去：

$$\mathbf{Y} = \mathbf{W}_L^T \mathbf{X} = \Sigma_L \mathbf{V}^T \quad (1.16)$$

其中 $\Sigma_L = \mathbf{I}_{L \times m} \Sigma$ 且 $\mathbf{I}_{L \times m}$ 为 $L \times mL \times m$ 的单位矩阵。X 的单向量矩阵 W 相当于协方差矩阵的本征矢量 $C = XX^T$,

$$XX^T = W\Sigma\Sigma^TW^T \quad (1.17)$$

在欧几里得空间给定一组点数，第一主成分对应于通过多维空间平均点的一条线，同时保证各个点到这条直线距离的平方和最小。去除掉第一主成分后，用同样的方法得到第二主成分。依此类推。在 Σ 中的奇异值均为矩阵 XX^T 的本征值的平方根。每一个本征值都与跟它们相关的方差是成正比的，而且所有本征值的总和等于所有点到它们的多维空间平均点距离的平方和。PCA 提供了一种降低维度的有效办法，本质上，它利用正交变换将围绕平均点的点集中尽可能多的变量投影到第一维中去，因此，降低维度必定是失去讯息最少的方法。PCA 具有保持子空间拥有最大方差的最优正交变换的特性。然而，当与离散余弦变换相比时，它需要更大的计算需求代价。非线性降维技术相对于 PCA 来说则需要更高的计算要求。PCA 对变量的缩放很敏感。如果我们只有两个变量，而且它们具有相同的样本方差，并且成正相关，那么 PCA 将涉及两个变量的主成分的旋转。但是，如果把第一个变量的所有值都乘以 100，那么第一主成分就几乎和这个变量一样，另一个变量只提供了很小的贡献，第二主成分也将和第二个原始变量几乎一致。这就意味着当不同的变量代表不同的单位（如温度和质量）时，PCA 是一种比较武断的分析方法。但是在 Pearson 的题为”On Lines and Planes of Closest Fit to Systems of Points in Space”的原始文件里，是假设在欧几里得空间里不考虑这些。一种使 PCA 不那么武断的方法是使用变量缩放以得到单位方差。

1.4 KL 散度

$$H_p(q) = \sum_x q(x) \log_2 \left(\frac{1}{p(x)} \right) \quad (1.18)$$

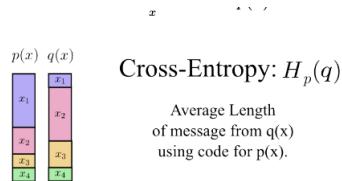


图 1.2: Cross_Entropy_exp

如果按照分别按照 $p(x)$ 和 $q(x)$ 出现的概率计算:

- $H(p) = \frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \frac{1}{8} \times 3 = 1.75\text{bit}$
- $H_p(q) = \frac{1}{8} \times 1 + \frac{1}{2} \times 2 + \frac{1}{4} \times 3 + \frac{1}{8} \times 3 = 2.25\text{bit}$
- $H(q) = \frac{1}{8} \times 3 + \frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 = 1.75\text{bit}$
- $H_q(p) = \frac{1}{2} \times 3 + \frac{1}{4} \times 1 + \frac{1}{8} \times 2 + \frac{1}{8} \times 3 = 2.375\text{bit}$

将上面的四种情况用图画出来, 如果两组概率服从统一分布, 他们将相邻:

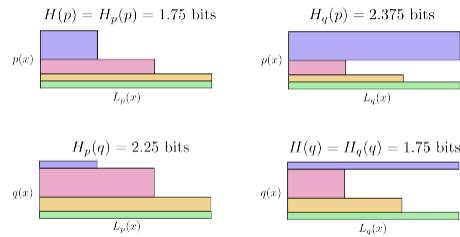


图 1.3: 比较四种情况算得的信息 bit

上图可以看出 $H_p(q) \neq H_q(p)$, 为什么? $H_q(p)$ 更大, 因为蓝色被分配了更多的 bit, 交叉熵给我们了一种方法来衡量两个概率分布的不同。p 和 q 越多不同, p 和 q 对应的交叉熵比 p 的熵就越大。

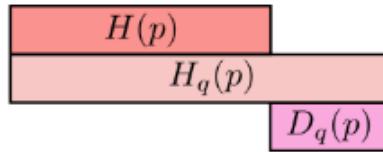


图 1.4: p 对 q 的交叉熵和 p 的熵

类似的, p 和 q 的差别越大, 相应的 q 对 p 的交叉熵比 q 的熵越大。

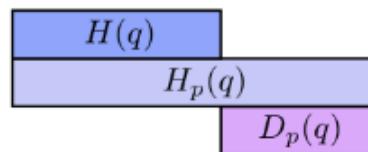


图 1.5: q 对 p 的交叉熵和 q 的熵

1.4.1 交叉熵

$$D_p(Q) = H_q(p) - H(p) = \sum_x p(x) \log_2 \left(\frac{p(x)}{q(x)} \right) \quad (1.19)$$

$\log_2 \left(\frac{p(x)}{q(x)} \right)$ 表示 q 表示的代码和 q 表示的代码有多少个 bit 不同，整个表达式表示两个代码有多少 bit 不同。KL divergence 实际上相当于两个分布之间的距离。

相对熵 (relative entropy) 又称为 KL 散度 (Kullback-Leibler divergence, 简称 KLD), 信息散度 (information divergence), 信息增益 (information gain) KL 散度是两个概率分布 P 和 Q 差别的非对称性度量。KL 散度是用来度量基于 Q 的编码来编码来自 P 的样本平均所需的额外的位元数。典型情况下, P 表示数据的真实分布, Q 表示数据的理论分布, 模型分布或 P 的近似分布。

对于离散随机变量, 其概率分布 P 和 Q 的 KL 散度可以按下面定义为

$$D_{KL}(P||Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)} \quad (1.20)$$

即按概率 P 求得的 P 和 Q 的对数差的平均值。KL 散度仅当 P 和 Q 各自总和均为 1, 且对任何 t 皆满足对于 $Q(i) > 0$ 及 $P(i) > 0$ 时才有定义。式子出现 $\ln 0$ 其值按 0 处理, 对于连续随机变量, 其概率分布 P 和 Q 可按计分方式定义为:

$$D_{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \ln \frac{p(x)}{q(x)} dx \quad (1.21)$$

其中 p 和 q 分别表示分布 P 和 Q 的概率密度。

1.4.2 相对熵

由 Gibbs 不等式可知, 当且仅当 $P = Q$ 时 $D_{KL}(P||Q)$ 为 0。尽管从直觉上 KL 散度是个度量或距离函数, 但是它实际上不是一个真正的度量或距离。因为 KL 散度不具有对称性: 从分布 P 到 Q 的距离 (或度量) 通常并不等于从 Q 到 P 的距离 (或度量)。

$$D_{KL}(P||Q) \neq D_{KL}(Q||P)$$

自信息和散度的关系: $I(m) = D_{KL}(\delta_{im}||p_i)$ 。互信息和散度:

$$\begin{aligned} I(X;Y) &= D_{KL}(P(X,Y)||P(X)P(Y)) \\ &= E_x D_{KL}(P(Y|X)||P(Y)) \\ &= E_y D_{KL}(P(X|Y)||P(X)) \end{aligned}$$

信息熵和散度:

$$\begin{aligned} H(X) &= (i) E_x I(x) \\ &= (ii) \log N - D_{KL}(P(X) || P_U(X)) \end{aligned}$$

条件熵和散度:

$$\begin{aligned} H(X|Y) &= \log N - D_{KL}(P(X,Y) || P_U(X)P(Y)) \\ &= (i) \log N - D_{KL}(P(X,Y) || P(X)P(Y)) - D_{KL}(P(X) || P_U(X)) \\ &= H(x) - I(X;Y) \\ &= (ii) \log N - E_Y D_{KL}(P(X|Y) || P_U(X)) \end{aligned}$$

交叉熵与散度: $H(p, q) = E_p[-\log q] = H(p) + D_{KL}(p || q)$

Chapter 2

Tensorflow 基础

2.1 Tensorflow 基础概念

在 Tensorflow 中正如它的名字显示的定义 tensor 计算。一个 tensor 是一个概括的矩阵和向量，并且有能力表示更高的维度，我们写 Tensorflow 程序，主要的对象就是 tf.Tensor，一个 tensor 定义计算的一部分最后生成值。TensorFlow 程序首先用 tensor 建立一个图，然后运行图获得想要的数据。一个 tensor 需要指定两个参数：数据类型和形状。Tensor 中的数据类型相同，而且总是可知的，形状可能仅仅部分知道。下面是一些特殊的 Tensor 类型：

- tf.Variable
- tf.Constant
- tf.Placeholder
- tf.SparseTensor

2.1.1 Rank

tf.Tensor 的 rank 是对象的维度。TensorFlow 的 rank 和数学中矩阵的 rank 不一样，下面显示 TensorFlow rank 和相对应的数学实体

rank	数学实体
0	Scalar(只有值)
1	Vecor(值和方向)
2	矩阵 (数值表)
3	3-Tensor
n	n-Tensor

rank0

下面片段展示创建一些 0 维的变量。

```

1 mammal = tf.Variable("Elephant", tf.string)
2 ignition = tf.Variable(451, tf.int16)
3 floating = tf.Variable(3.14159265359, tf.float64)
4 its_complicated = tf.Variable((12.3, -4.85), tf.complex64)
```

Rank1 传递列表作为初始值创建 1 维 tf.Tensor 对象

```

1 mystr = tf.Variable(["Hello"], tf.string)
2 cool_numbers = tf.Variable([3.14159, 2.71828], tf.float32)
3 first_primes = tf.Variable([2, 3, 5, 7, 11], tf.int32)
4 its_very_complicated = tf.Variable([(12.3, -4.85), (7.5, -6.23)], tf.complex64)
```

更高的 rank 二维的 Tensor 至少有一行一列

```

1 mymat = tf.Variable([[7],[11]], tf.int16)
2 myxor = tf.Variable([[False, True],[True, False]], tf.bool)
3 linear_squares = tf.Variable([ [4], [9], [16], [25] ], tf.int32)
4 squarish_squares = tf.Variable([ [4, 9], [16, 25] ], tf.int32)
5 rank_of_squares = tf.rank(squarish_squares)
6 mymatC = tf.Variable([[7],[11]], tf.int32)
```

更高 rank 的 Tensor，有 n 维数组。例如在图像处理，一些 tensor 的 rank 为 4, 维度通常 是 example-in-batch,image width,image height,color chennel。

```
1 my_image = tf.zeros([10, 299, 299, 3]) # batch x height x width x color
```

2.1.2 获取 Tensor 对象的 rank

你可以使用 tf.rank 方法获取 tensor 对象的 rank。例如下面获取 my3d 的 rank。

```
1 r = tf.rank(my3d)#在图运行后，r将保持值3。
```

2.1.3 Tensor 的切片

因为 tf.Tensor 是 n 维 cell 阵列，为了访问 tf.Tensor 的单个 cell，你需要指定索引。对于 rank 为 0 的 tensor，不需要索引，因为它已经是单个值了。

对于 rank1(向量)，传递一个索引允许你访问：

```
1 my_scale = my_vector[2]
```

如果你想动态的选择向量中的元素，你可以指定 [] 一个 tf.Tensor。传递一个数值访问矩阵的子向量：

```

1 my_row_vetor = my_matrix[2]
2 my_column_vector = my_matrix[:, 3]

```

2.1.4 形状

shape 是 tensor 每一维元素的个数。TensorFlow 在构造图的时候自动计算形状。有时候自动计算可能不知道 rank，如果 rank 已经知道，每一维的形状可能直到可能不知道。

rank	shape	维数	example
0	[]	0-D	0 维 Tensor, 标量
1	[D0]	1-D	一维 tensor 的形状
2	[D0,D1]	2-D	二维 Tensor 的形状
3	[D0,D1,D2]	3-D	三维 Tensor 的形状
n	[D0,D1,...,D _{n-1}]	N 维 tensor 的形状 [D ₀ , D ₁ , ..., D _{n-1}]	

2.1.5 获取 tf.Tensor 对象的形状

当建立图的时候 tensor 的形状已知是很有用的，你可以通过 tensor 的 shape 属性得到。得到完全定义的 tf.Tensor 的形状可以使用 Tf.shape 操作。这个方法你可以建立一个图操作 tensor 的形状。

例如，这里是如何如何创建一个和给定矩阵列数相同的全零向量。

```

1 zeros = tf.zeros(tf.shape(my_matrix)[1])

```

2.1.6 改变 Tensor 的形状

tensor 的元素是所有形状值的乘积。标量的元素总是 1. 因此，因为有相同元素不同形状的 tensor，转变他们的形状是很方便的。可以使用 tf.reshape.

下面例子展示了如何 reshape tensor。

```

1 rank_three_tensor = tf.ones([3, 4, 5])
2 matrix = tf.reshape(rank_three_tensor, [6, 10]) # Reshape existing content into
3 # a 6x10 matrix
4 matrixB = tf.reshape(matrix, [3, -1]) # Reshape existing content into a 3x20
5 # matrix. -1 tells reshape to calculate
6 # the size of this dimension.
7 matrixAlt = tf.reshape(matrixB, [4, 3, -1]) # Reshape existing content into a
8 # 4x3x5 tensor
9

```

```

10 # Note that the number of elements of the reshaped Tensors has to match the
11 # original number of elements. Therefore, the following example generates an
12 # error because no possible value for the last dimension will match the number
13 # of elements.
14 yet_another = tf.reshape(matrixAlt, [13, 2, -1]) # ERROR!

```

2.1.7 数据类型

tf.Tensor 不可能有一个以上的数据类型。然而序列化数据结构作为字符串尺寸处在 tf.Tensor 里是可能的。

可以使用 tf.cast 转换一种数据类型到另一种。

```

1 # Cast a constant integer tensor into floating point.
2 float_tensor = tf.cast(tf.constant([1, 2, 3]), dtype=tf.float32)

```

通过 Tensor 的 dtype 查看 tensor 的数据类型。你通过 python 对象创建 tf.Tensor 的时候需要指定数据类型。如果你不指定 TensorFlow 选择一个代表你数据的数据类型。TensorFlow 转换 Python 整数为 tf.int32, 浮点数为 tf.float32。转换数组时 TensorFlow 用和 numpy 相同的规则。

2.1.8 计算 Tensor

当计算图被创建后你可以通过运行计算 tf.Tensor 获取指定的值。用 Tensor.eval 方法简单的计算:

```

1 constant = tf.constant([1, 2, 3])
2 tensor = constant * constant
3 print(tensor.eval())

```

eval 方法仅仅当 tf.Session() 被激活时可用。Tensor.eval 然后会得到一个一个和 tensor 相同内容的 numpy 数组。有时候没有上下文计算 tf.Tensor 是不可能的。例如，tensor 依赖于 Placeholder 在提供给 Placeholder 值之前不能计算。

```

1 p = tf.placeholder(tf.float32)
2 t = p + 1.0
3 t.eval() # This will fail, since the placeholder did not get a value.
4 t.eval(feed_dict={p: 2.0}) # This will succeed because we're feeding a value
                           # to the placeholder.

```

其他的模型结构在计算 tf.Tensor 时可能很复杂。TensorFlow 不能直接计算定义在函数内部的或者控制流结构的 tf.Tensor。如果 tf.Tensor 伊奈于队列中的值，计算 tf.Tensor 仅仅入队的时候工作，负责计算被挂起。当和 queue 工作的时候，记得在计算任何 tf.Tensor 之前用 tf.train.start_queue_runners。

2.1.9 打印 Tensor

出于调试目的，你想要打印 `tf.Tensor` 的值。`tfdbg` 提供了高级的调制支持。TensorFlow 用下面的模板打印 `tf.Tensor`:

```
1 t = <<some tensorflow operation>>
2 print t # This will print the symbolic tensor when the graph is being built.
3          # This tensor does not have a value in this context.
```

这段代码打印 `tf.Tensor` 对象不是它的值，TensorFlow 提供了 `tf.Print` 操作，然后第一个没有改变的 `Tensor` 参数然后打印 `tf.Tensor` 的第二个参数。

为了正确的使用 `tf.Print()`, 必须要用它的返回值, 查看下面的例子:

```
1     #we are using the value returned by tf.Print  
2 result = t + 1 # Now when result is evaluated the value of t will be printed.
```

当你计算 `result` 你将计算 `result` 依赖的每个结果，因为 `result` 依赖于 `t`，然后计算 `t`，打印它的输入，`t` 被打印。

2.2 Variable

Tensorflow 变量是最好的在你的程序用表现共享，永久状态的方法，Variables 通过 tf.Variable 类操作。一个 Tf.Variable 代表随着在它上面的操作的进行他的值可能被改变和 tf.Tensor 不同在于 tf.Variable 存在于 session.run 之外。一个 tf.variable 存储永久 tensor，指定操作允许你读和修改他的值修改能通过多个 tf.Session 可视化，因此多个 worker 对于同一个 tf.Variable 可以查看到同样的值。

2.2.1 创建变量

创建变量最好的方法是调用 `tf.get_variable` 函数。这个函数要求你指定变量的名字，名字将作为副本访问相同的变量，和 `checkpoint` 和导入模型时变量的名字一样。`tf.get_variable` 也允许你重用一个先前创建的有同样名字的变量，使得定义重用层很方便。创建变量提供名字和形状。

```
1     my_variable = tf.get_variable("my_variable", [1, 2, 3])
```

上面代码创建了一个 3 维 tensor 变量 my_variable，它的形状为 [1,2,3]，默认数据类型为 tf.float32，通过随机 tf.glorot_uniform_initializer 初始化值。你也可以指定 dtype 和初始化方式。

TensorFlow 提供很一些方便的初始化器,你也可以通过有值的 tf.Tensor 初始化一个 tf.Variable。

```
1 other_variable = tf.get_variable("other_variable", dtype=tf.int32, initializer=
                                    tf.constant([23, 42]))
```

鼠疫当你用 tf.Tensor 作为初始化器你不要指定变量的形状, 因为初始化器用你的 Ttensor 的形状。

2.2.2 变量集合

因为断开一部分 TensorFlow 程序也许是想创建变量, 这有时候是一个简单的访问他们的方法。因此 TensorFlow 提供了 collections(集合) 代表有名字的 tensor 列表或者其他对象, 向 tf.Variable 实体。

默认每个 tf.Variable 被放在下面的两个 collections: tf.GraphKeys.Global_VARIABLE (可以被多个设备共享的变量),tf.GraphKeys.TRAINABLE_VARIABLE (TensorFlow 将计算梯度的变量)。如果你不想一个变量被训练,将它增加到 tf.GraphKeys.LOCAL_VARIABLE 集合。例如下面的代码段展示了如何增加一个 my_local 变量到这个集合。

```
1 my_local = tf.get_variable("my_local", shape=(), collections=[tf.GraphKey.
                                LOCAL_VARIABLE])
```

你也可以指定 trainable=False。

```
1 my_non_trainable = tf.get_variable("my_non_variable", shape=(), trainable=False)
```

你也可以用你自己的 collections. 任何字符串都是一个可用的集合的名字, 不需要明确的创建集合。增加一个变量 (或者任何对象) 到集合后创建变量, 调用 tf.add_to_collection。例如, 你可以用下面的代码增加一个已经存在的变量 my_local 到一个 my_collection_name 集合:

```
1 tf.add_to_collection("my_collection_name", my_local)
```

你可以用下面的代码获取你放置在 collection 里面的变量的和对象列表。

```
1 tf.get_collection("my_collection_name")
```

2.2.3 配置设备

像任何其他 TensorFlow 操作一样, 你可以放置变量到特别的设备上。例如, 下面的代码片在第二个 GPU 上创建一个变量 v。

```
1 with tf.device("gpu:1"):
2     v = tf.get_variable("v", 1)
```

对于变量在正确的设备上部署是很重要的。有时候放变量在 worker 上而不是参数服务器上，例如可能极大的减缓训练，在对快的情况下让每个 worker 独立的复制每个变量。为此我们提供了 `tf.train.replica_device_setter`。自动防止变量到参数 servers 上。例如：

```

1 cluster_spec={
2     "ps": [ "ps0:2222", "ps1:2222" ],
3     "worker": [ "worker0:2222", "woker1:2222", "worker2:2222" ] }
4 with tf.device(tf.train.replica_device_setter(cluster=cluster_spec)):
5     v = tf.get_variable("v", shape=[20, 20])#这个变量被replica_device_setter放置在
                                         参数server上

```

2.2.4 初始化变量

在使用变量之前，你必须对变量进行初始化。如果你在低级的 TensorFlow API(明确的创建自己的图和会话)上编程,你必须明确的初始化变量。最高级的框架像 `tf.contrib.slim`,`tf.estimator.Estimator` 和 `Keras` 在你训练模型前自动初始化变量。

明确的初始化是很有用的，因为他让你从 `checkpointer` 载入模型不用重复运行代价高昂的初始化其同时允许决定什么时候随机初始化的变量在分布式设置上被共享。

为了在开始训练之前初始化可训练的变量，调用 `tf_global_variables_initializer()`。这个函数是一个初始化 `tf.GraphKeys_VARIABLES` 集合所有变量的操作。运行下面的操作初始化所有的变量：

```
1 session.run(tf.global_variables_initializer())
```

如果你需要手动初始化变量，你可以运行变量初始化操作：

```
1 session.run(my_variable.initializer)
```

你可以查询那些变量没有被初始化：

```
1 print(session.run(tf.report_uninitialized_variables()))
```

注意，默认情况下 `tf.global_variables_initializer` 不指定变量的初始化顺序。因此一个初始化值依赖于另一个初始化值，你可能得到错误。任何时候你在一个不是所有的变量被初始化（用一个变量的值时候另一个变量正在初始化）的环境下最好用 `variable.initialized_value()` 代替 `variable`。

```

1 v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
2 w = tf.get_variable("w", initializer=tf.initialized_value() + 1)

```

2.2.5 用变量

为了在 TensorFlow 图总使用 `tf.Variable`, 简单的把变量当作 `tf.Tensor`。

```

1 v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
2 w = v+1#w是一个基于v的值计算的Tensor，任何死后的用在表达式中的变量自动转化一个
      tf.Tensor到他的值。

```

赋值给一个变量用方法 assign,assign_add 和 tf.Variable。例如你可以这样调用这些方法:

```

1 v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
2   assignment = v.assign_add(1)
3 assignment = v.assign_add(1)
4 tf.global_variable_initializer().run()
5 assignment.run()

```

多数 TensorFlow 优化器根据一些类似梯度下降的算法已经指定了高效的更新变量值的操作。因为变量是可以更改的，有时候知道变量任何时间点的被使用的值是很有用的。你可以用 tf.Variable.read_value 在有时候变量使用后读取变量的值。

```

1 v = tf.get_variable("v", shape=(), initializer=tf.zeros\_initializer())
2 assignment = v.assign_add(1)
3 with tf.control_dependencies([assignment]):
4   w = v.read_value()

```

2.2.6 保存和恢复

用 tf.train.Saver 对象保存恢复模型是一种最简单的方法。这个构造提为图上所有的或者指定的变量添加 save 和 restore 操作。Saver 提供了方法运行这些操作，指定 checkpoint 文件读写的路径。为了恢复模型的 checkpoint 而不是图，你必须首先从 MetaGraph(.meta 扩展的) 文件。通过调用 tf.train.import_meta_graph, 从执行一个 restore 返回一个 Saver。

2.2.7 checkpoint 文件

TensorFlow 保存变量在一个二进制文件中，大体上是映射变量的名字到 tensor 的值。当你创建一个 Saver 对象，你可以从 checkpoint 文件选择变量，默认对每个变量用 tf.Variable.name 的值。

2.2.8 保存变量

用 tf.train.Saver() 创建一个 Saver 管理模型的所有变量。例如，下面的代码段展示了如何调用 tf.train.Saver.save() 方法保存变量为一个 checkpoint 文件。

```

1 #创建变量
2 v1 = tf.get_variable("v1", shape=[3], initializer = tf.zeros_initializer())
3 v2 = tf.get_variable("v2", shape=[5], initializer = tf.zeros_initializer())

```

```

4 inc_v1 = v1.addign(v1+1)
5 dec_v2 = v2.assign(v2-1)
6 #增加一个操作初始化变量
7 init_op = tf.global_variables_initializer()
8 #增加操作保存所有的变量
9 saver = tf.train.Saver()
10 #载入模型初始化变量，保存变量到磁盘
11 with tf.Session() as sess:
12     sess.run(init_op)
13     inc_v1.op.run()
14     dec_v2.op.run()
15     save_path = saver.save(sess, './model.ckpt')
16     print("Model saved in file:%s"%save_path)

```

2.2.9 恢复变量

tf.train.Saver 对象不仅可以保存变量到 checkpoint 文件，也可以恢复变量。注意当你从一个文件恢复变量的时候你没有必要提前初始化他们。例如，下面的代码段展示了如何调用 tf.train.Saver.restore 方法从 checkpoint 文件恢复变量。

```

1 tf.reset_default_graph()
2 v1 = tf.get_variable("v1", shape=[3])
3 v2 = tf.get_variable("v2", shape=[5])
4 saver = tf.train.Saver()
5 with tf.Session() as sess:
6     saver.restore(sess, './model.ckpt')
7     print("模型恢复")
8     print("v1:%s"%v1.eval())
9     print("v2:%s"%v2.eval())

```

2.2.10 选择变量恢复

如果你不传递任何参数为 tf.train.Saver(),saver 处理图上所有的变量。每个按照变量创建的时候给定的名字保存。有时候明确的指定 checkpoint 文件中的变量的名字是有用的。例如你也许训练一个五层神经网络，你现在想重用之前的五层网络训练一个新的 6 层网络，你可以用 saver 回复前面 5 层的权重。你可以通过传递给 tf.train.Saver() 构造体变量列表的名字或者一个 key 是名字 value 是值的 Python 字典保存和载入变量。

```

1 tf.reset_default_graph()
2 v1 = tf.get_variable("v1", [3], initializer = tf.zeros_initializer())
3 v2 = tf.get_variable("v2", [5], initializer = tf.zeros_initializer())
4 saver = tf.train.Saver({"v2":v2})

```

```

5   with tf.Session() as sess:
6       v1.initializer.run()
7       saver.restore(sess, "./model.ckpt")
8       print("v1 : %s" % v1.eval())
9       print("v2 : %s" % v2.eval())

```

注意

- 如果你想保存和恢复模型的变量的子集，你可以创建多个 Saver 对象，它的值仅仅在 Saver.restore() 方法运行的时候才被载入。
- 如果你仅仅在会话开始是恢复变量的一个子集，你必须对其它变量执行初始化操作。

2.2.11 共享变量

TensorFlow 支持两种方法的共享变量：

- 明确传递 tf.Variable() 对象
- 在 tf.variable_scope 对象中隐式打包 tf.Variable 对象。

用 Variable scope 允许你控制变量重用调用函数隐式的创建使用变量。他也允许你在你的文件结构上命名你的变量方便理解。

```

1 def conv_relu(input, kernel_shape, bias_shape):
2     weights = tf.get_variable("weight", kernel_shape, initializer=tf.
3                               random_normal_initializer())
4     biases = tf.get_variable("biase", bias_shape, initializer=tf.
5                               constant_initializer(0.0))
6     conv = tf.nn.conv2d(input, weights, strides=[1, 1, 1, 1], padding="SAME")
7     return tf.nn.relu(conv+biases)

```

这个函数用 weights 和 biases 好处是清晰。在真实的模型中，我们想有一些卷基层，重读的调用这这函数将 not work:

```

1 input1 = tf.random_normal([1, 10, 10, 32])
2 input2 = tf.random_normal([1, 20, 20, 32])
3 x = conv_relu(input1, kernel_shape=[5, 5, 1, 32], bias_shape=[32])
4 x = conv_relu(x, kernel_shape=[5, 5, 32, 32], bias_shape=[32])

```

因为希望的行为不确定 (创建新的变量还是重用之前的变量?)TensorFlow 将不能做到。在不同的 scope 调用 conv_relu，然而，名且我们想创建新的变量：

```

1 def my_image_filter(input_images):
2     with tf.variable_scope("conv1"):

```

```
3 #这里被创建的变量名字为"conv1/weights","conv1/biases"
4     relu1 = conv_relu(input_images,[5,5,1,32],[32])
5 with tf.variable_scope("conv2"):
6     return conv_relu(relu1,[5,5,32,32],[32])
```

如果你想你的变量被共享，你有两个字选择。第一，你可以在创建一个 scope 的时候用 reuse=True:

```
1 with tf.variable_scope("model"):
2     output1 = my_image_filter(input1)
3 with tf.variable_scope("model",reuse=True):
4     output2 = my_image_filter(input2)
```

你可以调用 scope.reuse_variable() 触发一个 reuse:

```
1 with tf.variable_scope("model") as scope:
2     output1 = my_image_filter(input)
3     scope.reuse_variables()
4     output2 = my_image_filter(input2)
```

因为以来与提取 sopce 名字提取字符串可能很危险，也可以用下面的方法初始化:

```
1 with tf.variable_scope("model") as scope:
2     output1 = my_image_filter(input1)
3 with tf.variable_scope(scope,reuse=True):
4     output2 = my_image_filter(input2)
```

2.3 图 (Graphs) 和会话 (Session)

TensorFlow 用数据流图 (dataflow graph) 代表操作间的相应的计算。这导致首先你需要先定义图，创建 TensorFlow Session 通过本地设备或远程设备运行图的一部分。这个向导对于你想用低级变量模型是很有用的。要记得 API 像 `tf.estimator.Estimator` 和 Keras 对于用户端隐藏了图和会话的细节。

2.3.1 为什么用数据流图？

数据流图对于并行变成来说是一个常见的模型。在数据流图中，节点 (node) 代表了计算单元，边 (edge) 代表了数据消耗和产生。例如在 TensorFlow 图中，`tf.matmul` 操作将对应两个边 (两个相乘的矩阵) 单个节点一个输出 (相乘的结果)。TensorFlow 利用数据流图计算有如下好处：

- 并行性：通过指定边代表不同操作间的依赖，系统能很容易的识别能并行执行的操作。
- 分布执行：通过用边代表值在不同操作间的流动，这对于 tensorflow 分割你的程序打不同的机器上的设备 (CPUs, GPUs, TPUs) 上。TensorFlow 插入必须的计算和不同设备间的协调。
- 编译：TensorFlow 的 XLA compiler 可以用你的数据流图的信息生成更快的打 `uma`，例如通过融合连接操作。
- 数据流图是一个代表你模型的代码，你可以用 Python 建立图，存储在 `SavedModel`，为了更低的推理延迟在 C++ 程序中恢复。

2.3.2 建立一个 `tf.Graph`

大多数的 TensorFlow 以构造一个数据流图作为开始时期，在这个时期，你利用 TensorFlow 的 API 函数构造 `tf.Operation`(节点) 和 `tf.Tensor`(边) 对象，添加他们到图实例上。TensorFlow 提供默认的图到相同上下文环境下的 API 函数，例如：

- 调用 `tf.constant(42.0)` 创建一个 `tf.Operation` 生成值 42.0，添加值到默认的图上，返回一个掉表这个常量值的 `tf.Tensor`。
- 调用 `tf.matmul(x,y)` 创建一个 `tf.Tensor` 对象 `x,y` 用 `tf.Operation` 相乘，增加它到默认的图上，返回一个代表相乘结果的 `tf.Tensor`。
- 执行 `v=tf.variable(0)` 给图添加一个 `tf.Operation` 到图上，操作将存储可以写的 Tensor 值在 `tf.Session.run` 调用前。`tf.Variable` 对象包装这个操作，然后他能被想 tensor 一

样使用，Tensor 将读当前存储的之。tf.Variable 对象有 assign 和 assig_aadd 之类的方法，当方法被执行的时候，更新存储的值。

- 调用 tf.train.Optimizer.minimize 将操作和 tensor 到默认的涂上计算梯度，返回一个 tf.Operation，当运行的时候，用图读设置变量。

多数程序依赖于默认的图，在 TensorFlow API 调用大多数程序仅仅添加操作和 tensor 到默认的图上，并不执行实际的计算。当你通过这些 tf.Tensor 和 tf.Operation 代表你的函数传递给 tf.Session 进行计算。

2.3.3 命名你的操作

tf.Graph 对象给它包含的包含 tf.Operation 对象定义了一个 namespace。TensorFlow 自动为你涂上的才注意选择一个独一无二的名字，而且给操作名字方便程序易读和调试。TensorFlow API 提供了两个操作来覆盖操作的名字：

- 每个 API 函数创建一个新的 tf.Operation 或者返回一个新的 tf.Tensor 时接受一个 name 选项。例如 tf.constant(42.0,name="answer") 创建一个新的操作名字叫 answer，返回一个名字为" answer:0 "的 tf.Tensor。如果默认图已经包含了名字为"answer" 的操作，TensorFlow 将添加"-1","-2" 等等，例如：

```

1 c_0 = tf.constant(0,name="c")#操作的名字为"c"
2 c_1 = tf.constant(2,name="")#操作的名字为"c_1"
3 with tf.name_scope("outer"):
4     c_2 = tf.constant(2,name="c")#操作的名字为outer/c
5     with tf.name_scope("inner"):
6         c_2 = tf.constant(3,name="c")
7     c_4 = tf.constant(4,name="c")#操作名字为"outer/c_1"
8     with tf.name_scope("inner"):
9         c_5 = tf.constant(5,name="c")

```

tf.Tensor 对象隐藏为 tf.Operation 名字，之后 tf.Operation 将产生 tensor 作为输出。一个 tensor 的名字形式”<OP_NAME>:<i>”，这里：

- ”<OP_NAME>” 是产生它的操作的名字。
- ”<i>” 是一个整数操作输出的 tensor 的索引。

2.3.4 放置操作在不同的设备上

如果你想 TensorFlow 用多个不同的设备，tf.device 函数提供了方便的方法请求所有的操作在一个特别的上下文被放置现在相同的设备上。指定格式如下：

```
1 /job:<JOB_NAME>/task:<TASK_INDEX>/device:<DEVICE_TYPE>:<DEVICE_INDEX>
```

这里:

```
1 \item <JOB_NAME>是一个alpha数字，不是以数字开头
2 \item <DEVICE_TYPE>是一个u注册的设备。
3 \item <TASK_INDEX>一个非负整数代表job中的任务的索引
4 \item <JOB_NAME>查看tf.train.ClusterSpec查看更多关于jobs和tasks的解释。
5 \item <DEVICE_INDEX>:一个代表device索引的非负整数，例如为了区别在同一进程中的不同GPU。
```

你不需要制定设备的每一部分，例如，如果你运行在一个单 GPU 的机器上，你也许用 tf.device 添加一些操作到 CPU 和 GPU 上。

```
1 weights = tf.random_normal()
2 with tf.device("/device:CPU:0")
3     img = tf.decode_jpeg(tf.read_file("img.jpg"))
4 with tf.device("/device:GPU:0"):
5     result = tf.matmul(weights, img)
```

如果你用典型的分布式配置部署 TensorFlow，你也许制定 job 的名字和 task ID 放置变量在参数服务器 job("/job:ps")，其他的操作在 worker job("/job:worker")

```
1 with tf.device("/job:ps/task:0"):
2     weight_1 = tf.Variable(tf.truncated_normal([784, 100]))
3     biases_1 = tf.Variable(tf.zeros([100]))
4 with tf.device("/job:ps/task:1"):
5     weight_2 = tf.Variable("/job:ps/task:1")
6     biases_2 = tf.Variable(tf.zeros([10]))
7 with tf.device("/job:worker"):
8     layer_1 = tf.matmul(train_batch, weight_1) + biases_1
9     layer_2 = tf.matmul(train_batch, weight_2) + biases_2
```

tf.device 给你一些灵活度选择防止单个操作或者更广范围的 TensorFlow 图。在一些情况下，有简单的算法。例如 tf.train.replica_device_setter API 可以用 tf.device 防止操作 parallel distributed training. 例如下面的代码段显示 tf.train.replica_device_setter 应用不同的放置策略到 tf.Variable 对象和其他操作:

```
1 with tf.device(tf.train.replica_device_setter(ps_tasks=3)):
2 # tf.Variable objects are, by default, placed on tasks in "/job:ps" in a
3 # round-robin fashion.
4     w_0 = tf.Variable(...) # placed on "/job:ps/task:0"
5     b_0 = tf.Variable(...) # placed on "/job:ps/task:1"
6     w_1 = tf.Variable(...) # placed on "/job:ps/task:2"
```

```

7   b_1 = tf.Variable(...) # placed on "/job:ps/task:0"
8   input_data = tf.placeholder(tf.float32)      # placed on "/job:worker"
9   layer_0 = tf.matmul(input_data, w_0) + b_0 # placed on "/job:worker"
10  layer_1 = tf.matmul(input_data, w_1) + b_1 # placed on "/job:worker"

```

2.3.5 Tensor-like 对象

一些 TensorFlow 操作接受一个或者更多的 tf.Tensor 队形作为参数。例如, tf.matmul 得到 tf.Tensor 对象, tf.add_n 得到一个 tf.Tensor 列表对象。为了方便死用这些函数接受一个 tensor-like 对象在 tf.Tensor, 用 tf.convert_to_tensor 方法转换它为 tf.Tensor, Tensor-like 包含下面的元素类型:

```

1 \item tf.Tensor
2 \item tf.Variable
3 \item numpy.ndarray
4 \item list(tensor-like 对象的列表)
5 \item Python 标量:bool, float, int, str。

```

你可以用 tf.register_tensor_conversion_function。

默认, 每次你相同的 tensor-like 对象 TensorFlow 将创建一个新的 tf.Tensor。如果 tensor-like 对象大 (numpy.ndarray 包含一些训练样本) 当你多次使用你也许会超出内存。为了避免这样, 手动调用 tf.vonvert_to_tensor 在 tensor-like 对象, 用 tf.Tensor 返回。

2.3.6 在 tf.Session 执行图

TensorFlow 用 tf.session 类代表客户程序 (通常是 Python 程序), 通过一个类似的接口在 C++ 也可用。一个 tf.Session 对象提供访问设备在本地机器上, 远程设备用分布式的 TensorFlow 与。它也缓存一些关于你的 tf.Graph 的信息以至于你能高效的运行相同的计算多次。

2.3.7 创建 tf.Session

如果你用低层的 TensorFlow API, 你可以为当前图创建一个 tf.Session

```

1 # 创建一个默认的 session
2 with tf.Session() as sess:
3     # 创建一个远程会话
4     with tf.Session("grpc://example.org:2222"):

```

因为一个 tf.Session 拥有自己的物理资源 (像 GPU 和网络连接), 它是典型用作上下文管理 (with) 自动关闭会话。但是你应该确定调用 tf.Session.close 当你完成后你必须释放资源。

高级 API 像 `tf.train.MonitoredTrainingSession` 或者 `tf.estimator.Estimator` 将创建和管理一个 `tf.Session` 给你。APIs 接受 `target` 和 `config` 参数 (或者作为 `tf.estimator.RunConfig` 的一部分), 含义如下: `tf.Session.__init__` 接受三个参数:

- `target` 如果这个参数为空, 会话仅仅用在本地机器上。然而, 你也许指定 `grpc://` URL 指定 TensorFlow Server 地址让会话访问 server 上的所有设备。查看 `tf.train.Server` 查看 TensorFlow 创建 `yserver` 的详细信息。例如通常 `between-graph replication` 配置, `tf.Session` 在同一进程中作为客户链接 `tf.train.Server`。
- `graph` 默认情况下新的 `tf.Session` 将被限制到仅能在当前的图上运行操作。如果你在你的程序中用多个图, 你可以在创建会话的时候制定 `tf.Graph`。
- `config` 这个参数允许你制定 `tf.ConfigProto` 控制 session 的行为。例如包含一些配置选项。
- `allow_soft_placement` 设置设个参数为 `True` 使用 soft d 设备放置算法, 该算法忽略 `tf.device` 注释尝试放置 CPU-only 操作在 GPU 设备。
- `cluster_def`: 当用分布式的 TensorFlow, 这个选项允许你制定用于计算的机器, 提供不同 job 名字的映射任务索引和网络地址。
- `graph_option.optimizer_options`: 在执行前提供控制优化 TensorFlow 的执行。
- `gpu_options.allow_growth`: 设置为 `True` 改变 GPU 内存分配因至于他的随着内存分配渐渐增长, 而不是启动的时候粉煤尽量多的内存。

2.3.8 用 `tf.Session.run` 执行操作

`tf.Session.run` 是运行 `tf.Operation` 和评估 `tf.Tensor` 的主要操作, 一可以传递一个或者更多的 `tf.Operation` 和 `tf.Tensor` 或者 tensor-like 类型像 `tf.Variable`。这些 `fetch` 决定了 `tf.Graph` 的什用于计算 `fetch` 的所有子图操作的输出。例如下面的代码片段显示了不同的参数可能曹植不同的子图被执行:

```

1 x = tf.constant([[37.0, -23.0], [1.0, 4.0]])
2 w = tf.Variable(tf.random_uniform([2, 2]))
3 y = tf.matmul(x,w)
4 output = tf.nn.softmax(y)
5 init_op = w.initializer
6 with tf.Session() as sess:
7     #初始化变量
8     sess.run(init_op)
9     print(sess.run(output))
10    y_val, output_val = sess.run([y, output])

```

tf.Session.run 也可以用 feeds 词典，词典映射 tf.Tenor(典型的 tf.placeholder()) 对象到合适执行的值 (典型的 Python 标量, 列表, Numpy 数组)。

```

1 x = tf.placeholder(tf.float32, shape=[3])
2 y = tf.square(x)
3 with tf.Session() as sess:
4     print(sess.run(y, {x:[1.0,2.0,3.0]}))
5     print(sess.run(y, {x:[0.0,0.0,5.0]}))
6     #sess.run(y) 运行此代码会 Raise tf.errors.InvalidArgumentError
7     #因为当你计算一个tensor的时候它以来的值应该给定。
8     #sess.run(y,{x:37.0}) Raises ValueError, 因为 37.0 的形状不匹配

```

tf.Session.run 也接受一个选项 options 参数使你指定调用的选项, run_metadata 参数使你收集关于执行的 metadata。例如你可以用下面的选项处理执行信息:

```

1 y = tf.matmul([[37.0, -23.0], [1.0, 4.0]], tf.random_uniform([2, 2]))
2 with tf.Session() as sess:
3     # 定义 sess.run 的选项
4     options = tf.RunOptions()
5     options.output_partition_graphs = True
6     options.trace_level = tf.RunOptions.FULL_TRACE
7     # 定义返回 metadata 的容器
8     metadata = tf.RunMetadata()
9     sess.run(y, options=options, run_metadata=metadata)
10    # 打印每个设备执行的子图
11    print(metadata.partition_graphs)
12    # 打印每个操作的时间
13    print(metadata.step_status)

```

2.3.9 GraphDef 和 MetaGraphDef

TensorFlow 用数据流图作为程序的表示，一个 tf.Graph 包含两个相关的信息:

- 图的结构 (Graph structure): 节点和边指示了操作如何被组合在一起。但是没有描述他们如何被使用，图的结构像集合代码: 查看他们可能传达一些有用的信息，但是它不包含源代码传送的所有信息。
- 图集合 (Graph collections)TensorFlow 提供了一个通常的机制从 tf.Graph 的恢复 metadata 集合。tf.add_to_collection 函数使得你能用 key (tf.GraphKeys 定义的一些标准的 key) 结合列表对象看, tf.get_collection 使得你能结合 key 查看所有的对象一些 TensorFlow 库用如下机制: 当你创建一个 tf.variable, 他被增加到代表全局变量和训练的变量集合中，之后你创建一个 tf.train.Saver 或者 tf.train.Optimizer, 集合中的变量被用作默认参数。

一个图可以用两种形式表示:

- tf.GraphDef: 合适图结构的低级表示,包含所有操作的秒数和他们之间的边。tf.GraphDef 代表使用的低级 APIs, 像 tensorflow:session C++ APIs 通常它要求额外的上下文环境 (如典型的操作的名字) 来充分使用它。tf.Graph.as_graph_def 方法转换一个 tf.Graph 为 tf.GraphDef。
- tf.train.MetaGraphDef: 这是数据流图的高级表示它包含一个 tf.GraphDef, 和一些帮助我们理解图的信息 (像图集合的上下文信息)。tf.train.export_meta_graph 函数转化一个 tf.Graph 为 tf.trainMetaGraphDef。tf.train.Saver 方法也写一个 tf.train.MetaGraphDef, 它可能被用在结合保存的 checkpoint 文件去恢复训练保存点的状态。

通常情况下我们鼓励你用 tf.train.MetaGraphDef 而不是 tf.GraphDef。在一些情况下 tf.GraphDef 是很有用的, 例如当用下 tf.import_graph_def 或者 Graph Transform 这样的低级函数修改图, 但是 tf.train.MetaGraphDef 更好的用于建议高级应用。例如用 tf.train.MetaGraphDef SavedModel library 包装 tf.Graph 和一系列训练模型参数以至于他们能被用于服务。

如果你有 tf.train.MetaGraphDef, tf.train.import_meta_graph 函数将载入默认的图, 调用这个函数有下面两种特征:

- 它将从原始的图集合中恢复图的内容。像 tf.global_variable 和默认的参数像 tf.train.latest_checkpoint 函数可能被用于从类似的 checkpoint 目录寻找最新的 checkpoint

如果你有一个 tf.GraphDef, tf.import_graph_def 函数使得你能载入图进一个已经存在的 pythontf.Graph 对象。为了利用导入的图, 你必须知道在 tf.GraphDef 中操作或者 Tensor 的名字。tf.import_graph_def 函数有两个主要的特性帮你用导入的图:

- 你可以通过传递选项 input_map 参数重新绑定导入图的 tensor 对象到默认的图。例如, input_map 使你获取定义在 tf.GraphDef 导入图的片段, 连接图中的 tensor 和你在代码段中创建的 tf.placeholder。
- 你可以传递他们的名字到 return_elements 列表从导入的图中返回 tf.Tensor 或者 tf.Operation 对象

2.3.10 可视化你的图

TensorFlow 提供一个工具帮助你理解你图中的代码。图的可视化是 tensorBoard 的一个组件它在你的浏览器中生成你的图的结构。最简单的创建可视化的方法是当创建 tf.summary.FileWriter 时传递 tf.Graph

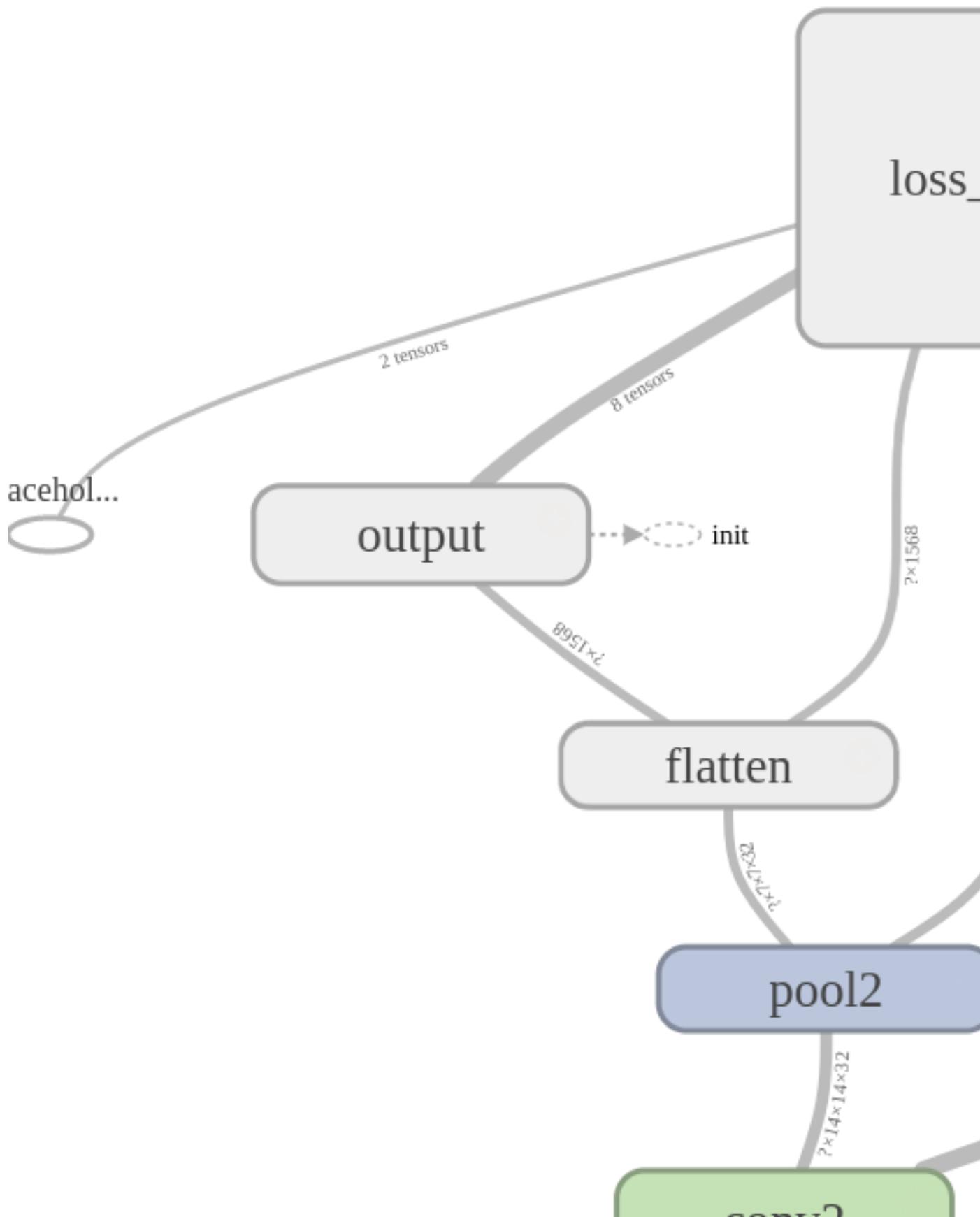
```

1 # Build your graph.
2 x = tf.constant([[37.0, -23.0], [1.0, 4.0]])

```

```
3 w = tf.Variable(tf.random_uniform([2, 2]))
4 y = tf.matmul(x, w)
5 # ...
6 loss = ...
7 train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)
8 with tf.Session() as sess:
9     # sess.graph provides access to the graph used in a tf.Session.
10    writer = tf.summary.FileWriter("/tmp/log /...", sess.graph)
11    # Perform your computation ...
12    for i in range(1000):
13        sess.run(train_op)
14        # ...
15    writer.close()
```

注意当你用 `tf.estimator.Estimator`, 图 (上面的任何总结) 将被自动采集到你创建 estimator 时指定的 `model_dir`。你可以在 TensorBoard 中打开采集, 导航到 Graph, 查看你的图的高级和石化结果。注意典型的 TensorFlow 图, 特别是训练图自动计算梯度, 同一时间有很多节点可视化。图可视化利用 scope 的名字分组相关的操作到高级节点。你可以点击曾色的 + 按钮展开里面的子图。



2.3.11 用多图编程

注意当训练一个模型的时候，通常的方法阻止你的代码是用一个图训练你的模型，另一个图评估或者在你的训练好的模型执行推理。在一些情况下，推理图将不同于训练图，例如像 dropout, batch normalization 在不同的情况下用不同的操作。更进一步默认的程序像 tf.trainSaver 用 tf.Variable 对象的名字识别在不同的 checkpoint 中的变量。当用这种方式编程的时候，你可以完全用 Python 处理建立，执行图，你也可以在同一进程用多个图。TensorFlow 提供了一个默认的图隐含的在相同的上下文环境传递所有的 API 函数。对于一些程序，单个图是足够的，然而 tensorflow 也提供了方法操作默认的节点下面的情况下可能很有用：

- 一个 tf.Graph 为 tf.Operation 对象定义了 tf.Operation 的 namespace，每个图中的操作必须有独一无二的名字。TensorFlow 将第一无二的名字通过添加”_1”, ”_2” 形成，因此如果他们的名字已经被转去了，用多个操作创建图给你更多控制每个操作的名字。
- 默认的图存储关于每个 tf.Operation 和 tf.Tensor 的信息。如果你的程序创建了很多互不连接的子图，用不同的 tf.Graph 建立子图可能更搞笑，因此不相关的状态可能被垃圾收集器收集。

你可以安装一个不同的 tf.Graph 作为默认的图，用 tf.Graph.as_default 上下文管理器：

```

1 g_1 = tf.Graph()
2 with g_1.as_default():
3     c = tf.constant("Node in g_1")
4     sess_1 = tf.Session()
5 g_2 = tf.Graph()
6 with g_2.as_default():
7     d = tf.constant("Node in g_2")
8 sess_2 = tf.Session(graph=g_2)
9 assert c.graph is g_1
10 assert sess_1.graph is g_1
11 assert d.graph is g_2
12 assert sess_2.graph is g_2

```

查看当前默认的图可以使用 tf.get_default_graph 返回一个 tf.Graph 对象。

```

1 # Print all of the operations in the default graph.
2 g = tf.get_default_graph()
3 print(g.get_operations())

```

2.4 数据集

Dataset 模块允许你从简单的，可重用的片段输入 pipeline。例如图像模块的 pipeline 也许集合分布式文件系统的数据，随机扰动每场图片，随机融合选中的图片为一个 batch 来训练，pipeline 的 text 模型能利用元素提取的文本数据，转换他们为查找表 embedding 的标志符将不同长度的序列放在一起成为一个 batch。Dataset API 使得处理大型数据，不同数据格式和复杂的转换变得很容易。一个 Dataset API 包含两个 TensorFlow 抽象。

- 一个 `tf.contrib.data.Dataset` 代表一个元素序列，其中的每个元素代表了一个或者更多的 Tensor 对象。例如在图像 pipeline，一个元素可能是单个的训练样本（一对代表了 label 和属相数据的 tensor）有两种方法创建数据集：
 1. 创建一个源 (`Dataset.from_tensor_slices()`) 从一个或者更多的 `tf.Tensor` 独享构造数据集。
 2. 应用转换格式从一个或者更多的 `tf.contrib.data.Dataset` 对象构造数据集。
- `tf.contrib.data.Iterator` 提供主要的从数据集提取元素的方法当 `Iterator.get_next()` 操作执行的时候从 Dataset 生成下一个元素，典型的行为作为输入 pipeline 和你的模型之间的接口。最简单的迭代器 (iterator) 是”one-shot iterator” 它结合了数据集和迭代。更多高级使用 Nike 一用 `Iterator.initializer` 操作用不同的数据集重新初始化和参数化一个迭代器，例如在一个程序多次迭代训练样本和验证集。

2.4.1 基本的机制

为了开始一个输入 pipeline 你需要定义一个源。例如从内存中的一些 tensor 构造一个数据集。你可以使用 `tf.contrib.data.Dataset.from_tensors()` 或者 `tf.contrib.data.Dataset.from_tensor_slices()`。如果你的输入数据在磁盘上，推荐你用 TFRecord 格式，你可以构造一个 `tf.contrib.data.TFRecordDataset`。

当你有一个 Dataset 对象的时候，你可以通过链式方法调用 `tf.contrib.data.Dataset` 对象转化成新的 Dataset。例如你可以用之前的元素转换作为 `Dataset.map()`（应用一个函数到每个元素）多元素转换像 `Dataset.batch()`。查看文档[tf.contrib.data.Dataset](#)完成列表转换。最常用的方法是消耗从 Dataset 来的值是创建一个迭代器对象，提供访问一个元素的数据集的元素一次（调用 `Dataset.make_one_shot_iterator()`），一个 `tf.contrib.data.Iterator` 提供两个操作：`Iterator.initializer` 重新初始化你的迭代器状态；`Iterator.get_next()` 返回迭代器下一个元素的 Tensor。

2.4.2 数据结构

一个数据及包含有相同结构的元素。一个元素包含一个或者更多称为组建的的 `tf.Tensor` 对象。每个组件有 `tf.DType` 代表在 tensor 中元素的数据类型，`Dataset.output_types` 和

Dataset.output_shapes 属性允许你查看每个数据集元素的组建的类型和形状。这些参数的潜逃结构映射元素的结构 (也许是一个 tensor, 一个 tensor 元组, 或者嵌套的 tensor 元组)

```

1 dataset1 = tf.contrib.data.Dataset.from_tensor_slices(tf.random_uniform([4, 10]))
2 print(dataset1.output_types)#{tf.float32
3 print(dataset1.output_shape)#{(10,)}
4 dataset2 = tf.contrib.data.Dataset.from_tensor_slices(tf.random_uniform([4]),
5 tf.random_uniform([4, 100], maxval=100, dtype=tf.int32))
6 print(dataset2.output_types)#{(tf.float32, tf.int32)
7 print(dataset2.output_shapes)#{(), (100,)}
8 datasets = tf.contrib.data.Dataset.zip((dataset1, dataset2))
9 print(dataset3.output_types)#{(tf.float32, (tf.float32, tf.int32))
10 print(dataset3.output_shapes)#{(10, (), (100,))}
```

给每个元素的组成名字是很方便的, 例如他们代表不同训练样本的特征。另外, 你可以用 collections.namedtuple 或者一个字典映射字符串到 tensor 代表 Dataset 的单个元素。

```

1 dataset = tf.contrib.data.Dataset.from_tensor_slices(
2   "a": tf.random_uniform([4]),
3   "b": tf.random_uniform([4, 100], maxval=100, dtype=tf.int32))
4 print(dataset.output_types)#{'a':tf.float32, 'b':tf.int32}
5 print(dataset.output_shape)#{'a':(), 'b':(100,)}
```

Dataset 转换支持任何的数据结构, 当你用 Dataset.map(), Dataset.flat_map 和 Dataset.filter() 转换应用函数到每个元素, 元素结构决定函数的参数:

```

1 dataset1 = dataset1.map(lambda x: ...)
2 dataset2 = dataset2.flat_map(lambda x, y: ...)
3 # Note: Argument destructuring is not available in Python 3.
4 dataset3 = dataset3.filter(lambda x, (y, z): ...)
```

2.4.3 创建一个迭代器

当你创建一个 Dataset 代表你的输入数据的时候, 下一步是创建一个迭代器从数据集中访问元素, Dataset API 当前支持三种迭代器:

- one-shot
- initilizable
- reinitilizable
- feedable

one-shot 迭代器是迭代器中最简单的形式，支持在数据集上迭代一次不需要初始化。One-shot 处理大多数的基于队列的输入 pipeline，但是他们不支持参数化。用 Dataset.range() 作为例子：

```

1 dataset = tf.contrib.data.Dataset.range(100)
2 iterator = dataset.make_one_shot_iterator()
3 next_element = iterator.get_next()
4 for i in range(10):
5     value = sess.run(next_element)
6     assert i == value

```

initializable 迭代器要求你使用前明确的运行 iterator.initializer 操作。作为不方便的交换，它允许你送入一个或者更多的 tf.placeholder() tensor 初始化你的迭代器，继续用 Dataset.range()：

```

1 max_value = tf.placeholder(tf.int64, shape=[])
2 dataset = tf.contrib.data.Dataset.range(max_value)
3 iterator = dataset.make_initializable_iterator()
4 next_element = iterator.get_next()
5 # Initialize an iterator over a dataset with 10 elements.
6 sess.run(iterator.initializer, feed_dict={max_value: 10})
7 for i in range(10):
8     value = sess.run(next_element)
9     assert i == value
10 # Initialize the same iterator over a dataset with 100 elements.
11 sess.run(iterator.initializer, feed_dict={max_value: 100})
12 for i in range(100):
13     value = sess.run(next_element)
14     assert i == value

```

一个 reinitializable 迭代器可以通过多个不同的 Dataset 对象初始化。例如你可续有一个用随机扰动输入图像提高泛化性的输入 pipeline 一个验证输入 pipeline 在没有修改的数据上评价预测。这些 pipeline 将用有相同结构（每个组件有相同的类型和兼容的形状）的不同的 Dataset 对象

```

1 # Define training and validation datasets with the same structure.
2 training_dataset = tf.contrib.data.Dataset.range(100).map(
3     lambda x: x + tf.random_uniform([], -10, 10, tf.int64))
4 validation_dataset = tf.contrib.data.Dataset.range(50)
5 # A reinitializable iterator is defined by its structure. We could use the
6 # output_types and output_shapes properties of either training_dataset
7 # or validation_dataset here, because they are compatible.
8 iterator = Iterator.from_structure(training_dataset.output_types,
9     training_dataset.output_shapes)
10 next_element = iterator.get_next()

```

```

11 training_init_op = iterator.make_initializer(training_dataset)
12 validation_init_op = iterator.make_initializer(validation_dataset)
13 # Run 20 epochs in which the training dataset is traversed, followed by the
14 # validation dataset.
15 for _ in range(20):
16     # Initialize an iterator over the training dataset.
17     sess.run(training_init_op)
18     for _ in range(100):
19         sess.run(next_element)
20     # Initialize an iterator over the validation dataset.
21     sess.run(validation_init_op)
22     for _ in range(50):
23         sess.run(next_element)

```

一个 feedable 迭代器可以和 tf.placeholder 用在一起调用 tf.Session.run 时选择什么咧带起通过 deed_dict 机制。它提供相同的函数作为重新初始化迭代器，但是当你在不同数据及切换的时候不要求你从数据集开始初始化。例如用相同的训练验证样本你可以用 tf.contrib.data.Iterator.from_string_handle 定义一个 feedable 迭代器允许你在不同的数据集之间切换：

```

1 # Define training and validation datasets with the same structure.
2 training_dataset = tf.contrib.data.Dataset.range(100).map(
3     lambda x: x + tf.random_uniform([], -10, 10, tf.int64)).repeat()
4 validation_dataset = tf.contrib.data.Dataset.range(50)
5 # A feedable iterator is defined by a handle placeholder and its structure. We
6 # could use the output_types and output_shapes properties of either
7 # training_dataset or validation_dataset here, because they have
8 # identical structure.
9 handle = tf.placeholder(tf.string, shape=[])
10 iterator = tf.contrib.data.Iterator.from_string_handle(
11     handle, training_dataset.output_types, training_dataset.output_shapes)
12 next_element = iterator.get_next()
13 # You can use feedable iterators with a variety of different kinds of iterator
14 # (such as one-shot and initializable iterators).
15 training_iterator = training_dataset.make_one_shot_iterator()
16 validation_iterator = validation_dataset.make_initializable_iterator()
17 # The Iterator.string_handle() method returns a tensor that can be evaluated
18 # and used to feed the handle placeholder.
19 training_handle = sess.run(training_iterator.string_handle())
20 validation_handle = sess.run(validation_iterator.string_handle())
21 # Loop forever, alternating between training and validation.
22 while True:
23     # Run 200 steps using the training dataset. Note that the training dataset is

```

```

24 # infinite , and we resume from where we left off in the previous while loop
25 # iteration .
26     for _ in range(200):
27         sess.run(next_element, feed_dict={handle: training_handle})
28     # Run one pass over the validation dataset .
29     sess.run(validation_iterator.initializer)
30     for _ in range(50):
31         sess.run(next_element, feed_dict={handle: validation_handle})

```

2.4.4 消耗迭代器的值

Iterator.get_next() 方法返回一个或多个迭代器的下一个元素 tf.Tensor 对象。每次迭代器被计算的时候他们得到数据集中的下一个元素，在 TensorFlow 中调用 Iterator.get_next() 不会立即前进迭代器。相反你需要在一个 TensorFlow 表达式返回 tf.Tensor 独享，传递表达式的结果给 tf.Session.run 得到表达式的结果个下一个迭代器。如果迭代器到大数据的尾部，执行 Iterator.get_next() 操作将报出 tf.errors.OutOfRangeError。这个点后迭代器将进入不稳定状态，你必须再次初始化它：

```

1 dataset = tf.contrib.data.Dataset.range(5)
2 iterator = dataset.make_initializable_iterator()
3 next_element = iterator.get_next()
4 # Typically result will be the output of a model, or an optimizer's
5 # training operation .
6 result = tf.add(next_element, next_element)
7 sess.run(iterator.initializer)
8 print(sess.run(result)) # ==> "0"
9 print(sess.run(result)) # ==> "2"
10 print(sess.run(result)) # ==> "4"
11 print(sess.run(result)) # ==> "6"
12 print(sess.run(result)) # ==> "8"
13 try:
14     sess.run(result)
15 except tf.errors.OutOfRangeError:
16     print("End of dataset") # ==> "End of dataset"

```

一个常用的模板是创建一个 try-except 块的训练循环包装器：

```

1 sess.run(iterator.initializer)
2 while True:
3     try:
4         sess.run(result)
5     except tf.errors.OutOfRangeError:
6         break

```

如果数据集中的每个元素都有迭代的结构在相同的迭代结果下 `Iterator.get_next()` 将返回一个或者更多的 `tf.Tensor`。

```

1 dataset1 = tf.contrib.data.Dataset.from_tensor_slices(tf.random_uniform([4, 10]))
           )
2 dataset2 = tf.contrib.data.Dataset.from_tensor_slices((tf.random_uniform([4]),
           tf.random_uniform([4, 100])))
3 dataset3 = tf.contrib.data.Dataset.zip((dataset1, dataset2))
4 iterator = dataset3.make_initializable_iterator()
5 sess.run(iterator.initializer)
6 next1, (next2, next3) = iterator.get_next()

```

注意计算任何 `next1`,`next2` 或者 `next3` 将对所有的组件前进迭代器。一个典型的消耗迭代器将不能包含在单个表达式中的所有组件。

2.4.5 读输入数据

消耗 NumPy 数据

如果你的所有数据都适合于存储，一个简单的方法是用 `Dataset.from_tensor_slices()` 转换他们为 `tf.Tensor` 对象创建一个 `Dadaset`。

```

1 # Load the training data into two NumPy arrays, for example using np.load().
2 with np.load("/var/data/training_data.npy") as data:
3     features = data["features"]
4     labels = data["labels"]
5 # Assume that each row of features corresponds to the same row as labels.
6 assert features.shape[0] == labels.shape[0]
7 dataset = tf.contrib.data.Dataset.from_tensor_slices((features, labels))

```

注意上面的带吧将在你的 TensorFlow 图中创建一个嵌入的 `features` 和 `labels` 作为一个 `tf.constant()` 操作。对于小的数据集这是很有用的，但是比较浪费存储，因为数据的内容将被多次复制可能达到 `tf.GraphDef` protocol buffer 的 2GB 限制。

```

1 # Load the training data into two NumPy arrays, for example using np.load().
2 with np.load("/var/data/training_data.npy") as data:
3     features = data["features"]
4     labels = data["labels"]
5     # Assume that each row of features corresponds to the same row as labels.
6     assert features.shape[0] == labels.shape[0]
7     features_placeholder = tf.placeholder(features.dtype, features.shape)
8     labels_placeholder = tf.placeholder(labels.dtype, labels.shape)
9     dataset = tf.contrib.data.Dataset.from_tensor_slices((features_placeholder,
           labels_placeholder))

```

```

10 # [Other transformations on dataset...]
11 dataset = ...
12 iterator = dataset.make_initializable_iterator()
13 sess.run(iterator.initializer, feed_dict={features_placeholder: features,
14                                         labels_placeholder: labels})

```

2.4.6 消耗 TFRecord 数据

一些数据集有一个或者多个文件。tf.contrib.data.TextLineDataset 提供了一个简单的方法从一个或者更多 text 文件提取行给定一个或者更多的文件名 TextLineDataset 将产生一个或者更多的字符串值元素。向 TFRecordDataset, TextLineDataset 接受 filenames 作为一个 tf.Tensor, 因此你可以通过 tf.placeholder 参数化它

```

1 filenames = ["/var/data/file1.txt", "/var/data/file2.txt"]
2 dataset = tf.contrib.data.TextLineDataset(filenames)

```

默认情况下一个 TextLineDataset 产生文件的每一行, 这也许并不是你想要的, 例如一个文件的开头有一些注释。这些行可以用 Dataset.skip() 移除和 Dataset.filter() 转换。为了应用这些转换在每个分割的文件, 我们用 Dataset.flat_map() 为每个文件创建一个迭代的 Dataset

```

1 filenames = ["/var/data/file1.txt", "/var/data/file2.txt"]
2
3 dataset = tf.contrib.data.Dataset.from_tensor_slices(filenames)
4
5 # Use Dataset.flat_map() to transform each file as a separate nested dataset,
6 # and then concatenate their contents sequentially into a single "flat" dataset.
7 # * Skip the first line (header row).
8 # * Filter out lines beginning with "#" (comments).
9 dataset = dataset.flat_map(
10     lambda filename: (
11         tf.contrib.data.TextLineDataset(filename)
12         .skip(1)
13         .filter(lambda line: tf.not_equal(tf.substr(line, 0, 1), "#"))))

```

2.4.7 用 Dataset.map() 处理数据

Dataset.map(f) 通过使用函数 f 作用于输入数据集的每个元素生成一个新的数据集。它通过函数编程语言用 map 函数应用到列表。这个函数 f 接受 tf.tensor 对象代表一个单个的输入元素, 返回一个代表一个数据集中单个元素 tf.Tensor 对象。它通过标准的 TensorFlow 操作转化一个元素为另一个。

2.4.8 解析 tf.Example protocol buffer 消息

一些输入的 pipeline 从 TFRecord 格式的文件提取 tf.train.Example protocol buffer 消息，用 tf.python_io.TFRecordWriter。每个 tf.train.Example 记录包含一个或者多个特征，输入 pipeline 通常转换这些特征为 tensor。

```

1 # Transforms a scalar string example_proto into a pair of a scalar string and
2 # a scalar integer, representing an image and its label, respectively.
3 def _parse_function(example_proto):
4     features = {"image": tf.FixedLenFeature((), tf.string, default_value=""),
5                 "label": tf.FixedLenFeature((), tf.int32, default_value=0)}
6     parsed_features = tf.parse_single_example(example_proto, features)
7     return parsed_features["image"], parsed_features["label"]
8
9 # Creates a dataset that reads all of the examples from two files, and extracts
10 # the image and label features.
11 filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
12 dataset = tf.contrib.data.TFRecordDataset(filenames)
13 dataset = dataset.map(_parse_function)
```

2.4.9 解码图像数据变换大小

当在一个真实世界的图像数据中训练一个神经网络，需要转换不同大小到一个同样的大小，因此他们也许处理为一个固定的尺寸

```

1 # Reads an image from a file, decodes it into a dense tensor, and resizes it
2 # to a fixed shape.
3 def _parse_function(filename, label):
4     image_string = tf.read_file(filename)
5     image_decoded = tf.image.decode_image(image_string)
6     image_resized = tf.image.resize_images(image_decoded, [28, 28])
7     return image_resized, label
8
9 # A vector of filenames.
10 filenames = tf.constant(["/var/data/image1.jpg", "/var/data/image2.jpg", ...])
11
12 # labels[i] is the label for the image in filenames[i].labels = tf.constant([0, 37, ...])dataset =
13 # tf.contrib.data.Dataset.from_tensor_slices((filenames,
14 # labels))dataset = dataset.map(_parse_function)
```

2.4.10 用专门的 Python logic

考虑到性能要求，我们鼓励你尽可能用 TensorFlow 操作处理你的数据。然而当解析你的数据时有是有调用额外的 python 操作处理数据是有用的。为了这么做，在 Dataset.map()

转换中调用 tf.py_func() 操作

```

1 import cv2
2
3 # Use a custom OpenCV function to read the image, instead of the standard
4 # TensorFlow tf.read_file() operation.
5 def _read_py_function(filename, label):
6     image_decoded = cv2.imread(image_string, cv2.IMREAD_GRAYSCALE)
7     return image_decoded, label
8
9 # Use standard TensorFlow operations to resize the image to a fixed shape.
10 def _resize_function(image_decoded, label):
11     image_decoded.set_shape([None, None, None])
12     image_resized = tf.image.resize_images(image_decoded, [28, 28])
13     return image_resized, label
14
15 filenames = ["/var/data/image1.jpg", "/var/data/image2.jpg", ...]
16 labels = [0, 37, 29, 1, ...]
17
18 dataset = tf.contrib.data.Dataset.from_tensor_slices((filenames, labels))
19 dataset = dataset.map(
20     lambda filename, label: tf.py_func(
21         _read_py_function, [filename, label], [tf.uint8, label.dtype]))
22 dataset = dataset.map(_resize_function)

```

2.4.11 简单的批处理

一个最简单的批处理是堆叠数据集中 n 个连续的元素。Dataset.batch() 变换就是这么做的，和 tf.stack() 一样应用元素的每个组件，每个组件 i 所有的元素必须有一个相同的形状 tensor。

```

1 inc_dataset = tf.contrib.data.Dataset.range(100)
2 dec_dataset = tf.contrib.data.Dataset.range(0, -100, -1)
3 dataset = tf.contrib.data.Dataset.zip((inc_dataset, dec_dataset))
4 batched_dataset = dataset.batch(4)
5
6 iterator = batched_dataset.make_one_shot_iterator()
7 next_element = iterator.get_next()
8
9 print(sess.run(next_element)) # ==> ([0, 1, 2, 3], [0, -1, -2, -3])
10 print(sess.run(next_element)) # ==> ([4, 5, 6, 7], [-4, -5, -6, -7])
11 print(sess.run(next_element)) # ==> ([8, 9, 10, 11], [-8, -9, -10, -11])

```

2.4.12 批量的 tensorpadding

上面的方法要要求所有的元素有相同的尺寸，然而一些模型（sequence 模型）中输入数据有不同的形状。为了处理这些情况，`Dataset.padded_batch()` 使你通过制定一个或者更多维（需要 padding）转换不同形状的 tensor 为一个 batch。

```

1 dataset = tf.contrib.data.Dataset.range(100)
2 dataset = dataset.map(lambda x: tf.fill([tf.cast(x, tf.int32)], x))
3 dataset = dataset.padded_batch(4, padded_shapes=[None])
4
5 iterator = dataset.make_one_shot_iterator()
6 next_element = iterator.get_next()
7
8 print(sess.run(next_element)) # => [[0, 0, 0], [1, 0, 0], [2, 2, 0], [3, 3, 3]
9
10 print(sess.run(next_element)) # => [[4, 4, 4, 4, 0, 0, 0],
11 # [5, 5, 5, 5, 5, 0, 0],
12 # [6, 6, 6, 6, 6, 6, 0],
13 # [7, 7, 7, 7, 7, 7, 7]]

```

`Dataset.padded_batch()` 转换允许你为每组件的一维度设置不同的 padding，他根据有变化的长度或者固定的长度，他可以覆盖 padding 的值（默认为 0）。

2.4.13 处理多 epoch

Dataset API 提供了两个主要的方法处理相同数据的多个 epochs，最简单的方法是用 Dataset.repeat() 变换数据集在多个 epoch、例如，在输入中创建一个数据集 10 epochs

```
1 filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
2 dataset = tf.contrib.data.TFRecordDataset(filenames)
3 dataset = dataset.map(...)
4 dataset = dataset.repeat(10)
5 dataset = dataset.batch(32)
```

使用 `Dataset.repeat()` 变换没有参数重复输入将不确定。`Dataset.repeat()` 变换连接他的参数每个 epochs 没有任何结束信号和下一个 epoch 的开始信号。如果你想接收每个 epoch 信号，你可以写一个循环在数据集的末尾捕获 `tf.errors.OutOfRange`。

```
1 filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
2 dataset = tf.contrib.data.TFRecordDataset(filenames)
3 dataset = dataset.map(...)
4 dataset = dataset.batch(32)
5 iterator = dataset.make_initializable_iterator()
6 next_element = iterator.get_next()
```

```

7
8 # Compute for 100 epochs.
9 for _ in range(100):
10     sess.run(iterator.initializer)
11     while True:
12         try:
13             sess.run(next_element)
14         except tf.errors.OutOfRangeError:
15             break
16
17     # [Perform end-of-epoch calculations here.]
```

2.4.14 随机打乱输入数据

Dataset.shuffle() 用和 tf.RandomShuffleQueue 方法随即打乱输入数据集，它保持一个固定的 buffer 平均的从 bugger 选择下一个元素：

```

1 filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
2 dataset = tf.contrib.data.TFRecordDataset(filenames)
3 dataset = dataset.map(...)
4 dataset = dataset.shuffle(buffer_size=10000)
5 dataset = dataset.batch(32)
6 dataset = dataset.repeat()
```

2.4.15 用高级 APIs

tf.train.MonitoredTrainingSession API 简化了一些在分布式方面运行方面的处设置。MonidotedTrainingSession 用 tf.errors.outOfRangeError 作为训练完成的标记,因此用 Dataset API, 我们推荐用 Dataset.make_one_shot_iterator() 例如:

```

1 filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
2 dataset = tf.contrib.data.TFRecordDataset(filenames)
3 dataset = dataset.map(...)
4 dataset = dataset.shuffle(buffer_size=10000)
5 dataset = dataset.batch(32)
6 dataset = dataset.repeat(num_epochs)
7 iterator = dataset.make_one_shot_iterator()
8
9 next_example, next_label = iterator.get_next()
10 loss = model_function(next_example, next_label)
11
12 training_op = tf.train.AdagradOptimizer(...).minimize(loss)
13
```

```
14 with tf.train.MonitoredTrainingSession(...) as sess:  
15     while not sess.should_stop():  
16         sess.run(training_op)
```

为了在 `tf.estimator.Estimator` 的 `input_fn` 中使用一个 `Dataset`, 我们推荐用 `Dataset.make_one_shot_iterator`。例如:

```
1 def dataset_input_fn():  
2     filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]  
3     dataset = tf.contrib.data.TFRecordDataset(filenames)  
4  
5     # Use tf.parse_single_example() to extract data from a tf.Example  
6     # protocol buffer, and perform any additional per-record preprocessing.  
7     def parser(record):  
8         keys_to_features = {  
9             "image_data": tf.FixedLenFeature((), tf.string, default_value=""),  
10            "date_time": tf.FixedLenFeature((), tf.int64, default_value=0),  
11            "label": tf.FixedLenFeature((), tf.int64,  
12                                         default_value=tf.zeros([], dtype=tf.int64)),  
13        }  
14         parsed = tf.parse_single_example(record, keys_to_features)  
15  
16         # Perform additional preprocessing on the parsed data.  
17         image = tf.decode_jpeg(parsed["image_data"])  
18         image = tf.reshape(image, [299, 299, 1])  
19         label = tf.cast(parsed["label"], tf.int32)  
20  
21         return {"image_data": image, "date_time": parsed["date_time"]}, label  
22  
23     # Use Dataset.map() to build a pair of a feature dictionary and a label  
24     # tensor for each example.  
25     dataset = dataset.map(parser)  
26     dataset = dataset.shuffle(buffer_size=10000)  
27     dataset = dataset.batch(32)  
28     dataset = dataset.repeat(num_epochs)  
29     iterator = dataset.make_one_shot_iterator()  
30  
31     # features is a dictionary in which each value is a batch of values for  
32     # that feature; labels is a batch of labels.  
33     features, labels = iterator.get_next()  
34     return features, labels
```

2.5 线程和队列

注意在 TensorFlow1.2 之前我们推荐用多线程，队列输入 pipeline，在 TensorFlow1.2 开始我们推荐使用 tf.contrib.data 模块。tf.contrib.data 提供了一个更加简单的结构构建高效的输入 pipeline，我们已经停止了之前正在开发的多线程和队列输入 pipeline，我们帮依然维护旧的代码的开发者维护文档。

```

1 q = tf.FIFOQueue(3, "float")
2 init = q.enqueue_many(([0., 0., 0.],))
3 x = q.dequeue()
4 y = x+1
5 q_inc = q.enqueue([y])
6 init.run()
7 q_inc.run()
8 q_inc.run()
9 q_inc.run()
10 q_inc.run()
```

Enqueue,EnqueueMany 和 Dequeue 是一个特别的节点。他们的代队列真实值的指针，允许他们改变状态。我们推荐你考虑这些操作的时候用面向对象的理解，事实上在 Python API 中这些操作通过调用队列的方法。

注意 Queue 方法必须运行在相同的设备上，不兼容的设备放置将在创建这些操作的时候被忽略

2.5.1 队列用法

像 tf.FIFOQueue 和 tf.RandomShuffleQueue 是在图上执行异步计算的重要的 TensorFlow 对象。典型的队列输入 pipline 用 RandomShuffleQueue 为训练模型准备输入：

- 多线程准备训练数据和将数据入队
- 训练线程执行训练操作从队列出队 mini-batch

我们推荐使用 Dataset 的 shuffle 和 batch 方法完成这个任务。然而，如果你仍然愿意使用队列版本，你可以在 tf.train.shuffle_batch 中找到完美的实现。

下面展示一个简单的实现，这个函数获取一个 source tensor，capacity 和 batch size 作为参数返回一个批量打乱的出队 tensor。

```

1 def simple_shuffle_batch(source, capacity, batch_size=10):
2     # Create a random shuffle queue.
3     queue = tf.RandomShuffleQueue(capacity=capacity,
```

```

4                         min_after_dequeue=int(0.9*capacity),
5                         shapes=source.shape, dtypes=source.dtype)
6
7 # Create an op to enqueue one item.
8 enqueue = queue.enqueue(source)
9
10 # Create a queue runner that, when started, will launch 4 threads applying
11 # that enqueue op.
12 num_threads = 4
13 qr = tf.train.QueueRunner(queue, [enqueue] * num_threads)
14
15 # Register the queue runner so it can be found and started by
16 # tf.train.start_queue_runners later (the threads are not launched yet).
17 tf.train.add_queue_runner(qr)
18
19 # Create an op to dequeue a batch
20 return queue.dequeue_many(batch_size)

```

当 `tf.train.start_queue_runners` 开始的时候,或者直接通过 `tf.train.MonitoredSession`,`QueueRunner` 将在后台开启进程填充队列, 同时主线程执行 `dequeue_many` 操作从中拉取数据, 现在这些操作不相互依赖, 除非间接地通过队列的内部依赖。简单的用这个函数像这样:

```

1 # create a dataset that counts from 0 to 99
2 input = tf.constant(list(range(100)))
3 input = tf.contrib.data.Dataset.from_tensor_slices(input)
4 input = input.make_one_shot_iterator().get_next()
5
6 # Create a slightly shuffled batch from the sorted elements
7 get_batch = simple_shuffle_batch(input, capacity=20)
8
9 # MonitoredSession will start and manage the QueueRunner threads.
10 with tf.train.MonitoredSession() as sess:
11     # Since the QueueRunners have been started, data is available in the
12     # queue, so the sess.run(get_batch) call will not hang.
13     while not sess.should_stop():
14         print(sess.run(get_batch))

```

输出

```

1 [ 8 10  7  5  4 13 15 14 25  0]
2 [23 29 28 31 33 18 19 11 34 27]
3 [12 21 37 39 35 22 44 36 20 46]

```

对于更多的情况有 `tf.train.MonitoredSession` 提供的自动线程启动和管理是足够的, 在极少的情况下不行, TensorFlow 提供了手动管理你的线程的工具。

2.5.2 手动线程管理

正如我们看到的，TensorFlow Session 是多线程的而且是线程安全的，因此多线程能够容易的在相同的会话和运行操作中使用。然而，不总是很容易实现一个 Python 程序按照要求驱动线程，所有的线程必须能同时停止，特别是必须捕获和报告，队列停止的时候必须被合适的关闭。TensorFlow 提供了两个类:tf.train.Coordinator 和 tf.train.QueueRunner。这两个类帮助多线程一起停止向程序报告异常等待他们停止，QueueRunner 类被用于创建一个线程协作同一队列中的入队 tensor。

2.5.3 Coordinator

tf.train.Coordinator 类管理 TensorFlow 程序的后台线程帮助多线程一起停止，关键的方法是：

- tf.train.Coordinator.should_stop: 如果线程应该被停止返回 True。
- tf.train.Coordinator.request_stop: 请求应该停止的线程。
- tf.train.Coordinator.join: 等待直到指定的线程被停止。

你首先创建一个 Coordinator 对象然后创建一些用于协调的线程。线程通常循环运行当 should_stop 为 True 时停止。任何线程都可以决定计算应该被停止。它仅仅必须调用 request_stop()，should_stop() 返回 True 是其它线程停止。

```

1 # Using Python's threading library.
2 import threading
3
4 # Thread body: loop until the coordinator indicates a stop was requested.
5 # If some condition becomes true, ask the coordinator to stop.
6 def MyLoop(coord):
7     while not coord.should_stop():
8         ...do something...
9         if ...some condition...:
10             coord.request_stop()
11
12 # Main thread: create a coordinator.
13 coord = tf.train.Coordinator()
14
15 # Create 10 threads that run 'MyLoop()'
16 threads = [threading.Thread(target=MyLoop, args=(coord,)) for i in range(10)]
17
18 # Start the threads and wait for all of them to stop.
19 for t in threads:
```

```

20     t.start()
21 coord.join(threads)

```

显然, coordinator 可以管理线程做不同的事。他们不是不许和上面的例子一样。coordinator 也支持捕获和报告异常, 查看[tf.train.Coordinator](#)文档查看更多信息。

2.5.4 QueueRunner

`tf.train.QueueRunner` 类创建一些线程重复执行入队操作。这些线程可以用 coordinator 一起停止, 另外一个队列 runner 将运行一个 closer 操作, 如果在 coordinator 中的队列被报告异常将关闭队列。你可以用一个队列 runner 实现下面的架构, 首先用一个 TensorFlow 为输入样本建立一个图, 添加操作处理将样本送入队列, 添加训练操作从队列出队。

```

1 example = ... ops to create one example...
2 # Create a queue, and an op that enqueues examples one at a time in the queue.
3 queue = tf.RandomShuffleQueue(...)
4 enqueue_op = queue.enqueue(example)
5 # Create a training graph that starts by dequeuing a batch of examples.
6 inputs = queue.dequeue_many(batch_size)
7 train_op = ... use 'inputs' to build the training part of the graph...

```

在 Python 的训练程序中, 创建一个 QueueRunner 将运行一些线程处理入队样本、创建一个 Coordinator 要求 queue runner 用 coordinator 开启它的线程。用 coordinator 写一个训练循环。

```

1 # Create a queue runner that will run 4 threads in parallel to enqueue
2 # examples.
3 qr = tf.train.QueueRunner(queue, [enqueue_op] * 4)
4
5 # Launch the graph.
6 sess = tf.Session()
7 # Create a coordinator, launch the queue runner threads.
8 coord = tf.train.Coordinator()
9 enqueue_threads = qr.create_threads(sess, coord=coord, start=True)
10 # Run the training loop, controlling termination with the coordinator.
11 for step in range(1000000):
12     if coord.should_stop():
13         break
14     sess.run(train_op)
15 # When done, ask the threads to stop.
16 coord.request_stop()
17 # And wait for them to actually do it.
18 coord.join(enqueue_threads)

```

2.5.5 处理异常

线程通过队列 runner 启动做的比仅仅运行入队操作要多。他们捕获处理队列生成的异常，包括用于报告队列被关闭的 tf.errors.OutOfRangeError 异常。一个训练中的程序用一个 coordinator 必须类似的在主循环中捕获和报告异常。下面是上面训练循环的一个改进的例子：

```
1 try:
2     for step in range(1000000):
3         if coord.should_stop():
4             break
5         sess.run(train_op)
6     except Exception, e:
7         # Report exceptions to the coordinator.
8         coord.request_stop(e)
9     finally:
10        # Terminate as usual. It is safe to call coord.request_stop() twice.
11        coord.request_stop()
12        coord.join(threads)
```

2.6 tf.estimator 快速导航

TensorFlow 的高级机器学习 API(tf.estimator) 使得配置, 训练评价多种机器学习模型变得很简单, 在这个导航中, 你讲用 tf.estimator 构造一个神经网络分类器在iris data基于花萼和花瓣的几何特性训练预测花的种类, 你的代码按照如下 5 步执行:

1. 载入 CSV 文件的训练测试数据到 TensorFlowDataset
2. 构造[神经网络分类器](#)
3. 用训练数据训练模型。
4. 评估模型的精度。
5. 分类新的样本

2.6.1 完成神经网络源代码

```
1 from __future__ import absolute_import
2 from __future__ import division
3 from __future__ import print_function
4
5 import os
6 import urllib
7
8 import numpy as np
9 import tensorflow as tf
10
11 # Data sets
12 IRIS_TRAINING = "iris_training.csv"
13 IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"
14
15 IRIS_TEST = "iris_test.csv"
16 IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"
17
18 def main():
19     # If the training and test sets aren't stored locally, download them.
20     if not os.path.exists(IRIS_TRAINING):
21         raw = urllib.urlopen(IRIS_TRAINING_URL).read()
22         with open(IRIS_TRAINING, "w") as f:
23             f.write(raw)
24
25     if not os.path.exists(IRIS_TEST):
```

```

26 raw = urllib.urlopen(IRIS_TEST_URL).read()
27 with open(IRIS_TEST, "w") as f:
28     f.write(raw)
29
30 # Load datasets.
31 training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
32     filename=IRIS_TRAINING,
33     target_dtype=np.int,
34     features_dtype=np.float32)
35 test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
36     filename=IRIS_TEST,
37     target_dtype=np.int,
38     features_dtype=np.float32)
39
40 # Specify that all features have real-value data
41 feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]
42
43 # Build 3 layer DNN with 10, 20, 10 units respectively.
44 classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
45                                         hidden_units=[10, 20, 10],
46                                         n_classes=3,
47                                         model_dir="/tmp/iris_model")
48
49 # Define the training inputs
50 train_input_fn = tf.estimator.inputs.numpy_input_fn(
51     x={"x": np.array(training_set.data)},
52     y=np.array(training_set.target),
53     num_epochs=None,
54     shuffle=True)
55
56 # Train model.
57 classifier.train(input_fn=train_input_fn, steps=2000)
58
59 # Define the test inputs
60 test_input_fn = tf.estimator.inputs.numpy_input_fn(
61     x={"x": np.array(test_set.data)},
62     y=np.array(test_set.target),
63     num_epochs=1,
64     shuffle=False)
65
66 # Evaluate accuracy.
67 accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]
68
69 print("\nTest Accuracy: {:.f}\n".format(accuracy_score))

```

```

69
70 # Classify two new flower samples.
71 new_samples = np.array(
72     [[6.4, 3.2, 4.5, 1.5],
73      [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
74 predict_input_fn = tf.estimator.inputs.numpy_input_fn(
75     x={"x": new_samples},
76     num_epochs=1,
77     shuffle=False)
78
79 predictions = list(classifier.predict(input_fn=predict_input_fn))
80 predicted_classes = [p["classes"] for p in predictions]
81
82 print(
83     "New Samples, Class Predictions:    {}\n"
84     .format(predicted_classes))
85
86 if __name__ == "__main__":
87     main()

```

下面的章节将详细介绍代码。

2.6.2 载入 CSV 数据进入 TensorFlow

Iris data set 包含有 150 行 iris 瓶中的样本:Iris setosa, Iris virginica 和 Iris versicolor。每行的数据包括花萼的长宽, 花瓣的长宽, 花用整数代表 0 表示 Iris setosa,1 表示 Iris versicolor,2 表示 Iris virginica。iris 数据集已经被分成两部分

- 120 个样本的训练集iris_training.csv
- 30 个样本的测试集iris_test.csv

导入需要的模型

```

1 from __future__ import absolute_import
2 from __future__ import division
3 from __future__ import print_function
4
5 import os
6 import urllib
7
8 import tensorflow as tf
9 import numpy as np
10

```



```
11 IRIS_TRAINING = "iris_training.csv"
12 IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"
13
14 IRIS_TEST = "iris_test.csv"
15 IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"
```

如果训练集和测试集没有被存储在本地，下载他们：

```
1 if not os.path.exists(IRIS_TRAINING):
2     raw = urllib.urlopen(IRIS_TRAINING_URL).read()
3     with open(IRIS_TRAINING, 'w') as f:
4         f.write(raw)
5
6 if not os.path.exists(IRIS_TEST):
7     raw = urllib.urlopen(IRIS_TEST_URL).read()
8     with open(IRIS_TEST, 'w') as f:
9         f.write(raw)
```

下一步用 `learn.dataset.base` 中的 `load_csv_with_header()` 方法载入训练数据进入 Dataset, `load_csv_with_header()` 方法接受三个参数:

- filename: CSV 文件的完成的路径加上文件名。
- target_dtype: 接收 numpy datatype 的数据集的目标值。
- feature_dtype: 接收 numpy datatype 类型的数据集的特征值。

这里的目标 (你的训练模型的预测) 是花的种类, 值范围为 0~2, 因此合适的 numpy 数据类型是 np.int。

```

1 # Load datasets.
2 training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
3     filename=IRIS_TRAINING,
4     target_dtype=np.int,
5     features_dtype=np.float32)
6 test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
7     filename=IRIS_TEST,
8     target_dtype=np.int,
9     features_dtype=np.float32)
```

在 tf.contrib.learn 中的 Dataset 是[named tuples](#); 你可以通过 data 和 target 访问特征数据和目标值, 这里 training_set 和 training_set.target 包含训练集的特征数据和目标数据, 对应的 test_set.data 和 test_set.target 包含测试集特征和目标。

2.6.3 构造神经网络分类器

tf.estimator 提供多种预定义方法, 称为 Estimator, 你可以在同国内你的数据”盒, 子”运行训练, 评估操作, 你可以实例化 tf.estimator.DNNClassifier:

```

1 # Specify that all features have real-value data
2 feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]
3
4 # Build 3 layer DNN with 10, 20, 10 units respectively.
5 classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
6                                         hidden_units=[10, 20, 10],
7                                         n_classes=3,
8                                         model_dir="/tmp/iris_model")
```

下面的代码中首先定义模型的特征列, 指定在数据集中的特征的数据类型。所有的特征数据是连续的, 因此 tf.feature_column.number_column 是构造特征列的合适的函数, 数据集中有 4 个特征, 因此我们指定 shape 为 [4] 保持所有的数据, 然后用下面的参数创建 DNNClassifier 分类器模型:

- feature_columns=feature_columns, 特征集合的列。

- hidden_units=[10,20,10], 三个hidden layer包含有 10, ,2,10 个神经元。
- n_classes=3, 三个目标类, 对应三个 iris 种类。
- model_dir=/temp/iris_model: 训练模型中保存 checkpoint 文件的路径

2.6.4 描述训练的输入 pipeline

tf.estimator API 用输入函数创建 TensorFlow 操作为模型生成数据,你可以用 tf.estimator.numpy_input_fn 生成输入 pipeline:

```

1 # Define the training inputs
2 train_input_fn = tf.estimator.inputs.numpy_input_fn(
3     x={"x": np.array(training_set.data)},
4     y=np.array(training_set.target),
5     num_epochs=None,
6     shuffle=True)

```

2.6.5 为 iris 训练集拟合 DNNClassifier

现在我们已经配置好的 classifier 模型, 你可以用 train 方法通过训练数据训练模型。传递 train_input_fn 作为 input_fn, 这里训练步数为 2000:

```

1 # Train model.
2 classifier.train(input_fn=train_input_fn, steps=2000)

```

状态模型每保存在 classifier, 依偎着你可以反复训练, 例如下面是合适的:

```

1 classifier.train(input_fn=train_input_fn, steps=1000)
2 classifier.train(input_fn=train_input_fn, steps=1000)

```

然而, 如果你在训练的时候跟踪模型, 你可以用 TensorFlow SessionRunHook 执行采集操作.

2.6.6 评估模型的精度

你可以在 Iris 训练集上训练你的 DNNClassifier 模型;现在你可以在测试集上用 evaluate 检查它在测试集上个精确度。evaluate 返回一个评估结果的字典, 下面的代码春娣 Irish 测数据给 test_set.data 和 test_set.target 评估和从结果中打印。

```

1 # Define the test inputs
2 test_input_fn = tf.estimator.inputs.numpy_input_fn(
3     x={"x": np.array(test_set.data)},
4     y=np.array(test_set.target),

```

```

5     num_epochs=1,
6     shuffle=False)
7
8 # Evaluate accuracy.
9 accuracy_score = classifier.evaluate(input_fn=test_input_fn)[“accuracy”]
10
11 print(“\nTest Accuracy: {0:f}\n”.format(accuracy_score))

```

这里 num_epochs=1 参数对于 numpy_input_fn 是很重要的。test_input_fn 将在数据上迭代一次然后报出 OutOfRangeError, 这个错误通知分类及停止评估, 因此它将计算输入一次

然后你可以运行完整的脚本, 它将打印出:

```
1 Test Accuracy: 0.966667
```

你的精度结果可能有点不同但是应该大于 90%。

2.6.7 分类新的样本

用 estimator 的 predict() 方法分类新的样本, 例如你有两个新的花的样本:

	花萼长度	花萼宽度	花瓣长度
6.4	3.2	4.5	
5.8	3.1	5.0	

你可以用 predict() 方法预测结果, predict 返回一个词典生成器, 生成器可以容易的被转化成列表, 下面的代码访问和打印预测的类:

```

1 # Classify two new flower samples.
2 new_samples = np.array(
3     [[6.4, 3.2, 4.5, 1.5],
4      [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
5 predict_input_fn = tf.estimator.inputs.numpy_input_fn(
6     x={"x": new_samples},
7     num_epochs=1,
8     shuffle=False)
9
10 predictions = list(classifier.predict(input_fn=predict_input_fn))
11 predicted_classes = [p[“classes”] for p in predictions]
12
13 print(
14     "New Samples, Class Predictions:    {}\n"
15     .format(predicted_classes))

```

你应该得到如下结果

```
1 New Samples , Class Predictions : [ 1 2 ]
```

结果预测样本是 Iris versicolor, Iris virginica。

2.7 用 tf.estimator 创建一个输入函数

在这个导航中向你介绍在 tf.estimator 创建一个输入函数。你将看到如何构造一个 input_fn 去处理和输入数据进你的模型，然后模拟将实现一个 input_ 函数到神经网络回归器训练，评估预测房价数据

2.7.1 用 input_fn 自定义 Pipeline

input_n 被用来传递特征和目标数据到 Estimator 的 train,evaluate,predict 方法。

```
1 import numpy as np
2
3 training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
4     filename=IRIS_TRAINING, target_dtype=np.int, features_dtype=np.float32)
5
6 train_input_fn = tf.estimator.inputs.numpy_input_fn(
7     x={"x": np.array(training_set.data)},
8     y=np.array(training_set.target),
9     num_epochs=None,
10    shuffle=True)
11
12 classifier.train(input_fn=train_input_fn, steps=2000)
```

2.7.2 input_fn 的分解

下面的代码描述了输入函数的基本结构:

```
1 def my_input_fn():
2
3     # Preprocess your data here...
4
5     # ...then return 1) a mapping of feature columns to Tensors with
6     # the corresponding feature data, and 2) a Tensor containing labels
7     return feature_cols, labels
```

输入函数的函数体包含指定处理你的输入数据的逻辑，像数据清洗和特征放大，输入函数必须返回两个包含最终的标签和特征的数据输入进你的模型：

feattrue_cols: 一个包含有映射特征列名字为 Tensor 包含有特征数据的键值 (key/-value) 对。

labels: 一个包含有你的标签的值 (你的模型想要预测的值)

2.7.3 转换特征数据为 Tensor

如果你的 feature/label 数据是一个 python 数据, 或者 pandas dataframe 或者 numpy 数组, 你可以用下面的方法构造 input_fn:

```

1 import numpy as np
2 # numpy input_fn.
3 my_input_fn = tf.estimator.inputs.numpy_input_fn(
4     x={"x": np.array(x_data)},
5     y=np.array(y_data),
6     ...)
```

```

1 import pandas as pd
2 # pandas input_fn.
3 my_input_fn = tf.estimator.inputs.pandas_input_fn(
4     x=pd.DataFrame({"x": x_data}),
5     y=pd.Series(y_data),
6     ...)
```

对于稀疏, 分类数据, 你将需要填入下面三个参数:

- dense_shape: 形状 tensor。每个维度的列表的索引。例如 dense_shape=[3,6] 指定二维 tensor, 3×6 ,dense_shape=[2,3,4] 指定 3 维 $2 \times 3 \times 4$ tensor,dense_shape=[9] 指定包含 9 个元素的一维 tensor。
- indices: 在你的包含有非零值的 tensor 的元素的索引。接受列表, 列表中的每个元素是包含非 0 元素的索引。(例如 [0,0] 代表两维 Tensor 的第 0 行第 0 列。indices=[[1,3],[2,4]] 指定索引为 [1,3],[2,4] 的元素有非零值。)
- values: 一维值得 tensor, values 中的 i 对应 indices 中的 i 和它指定的值。例如给定值 indices=[[1,3],[2,4]], 参数 values=[18,3.6], 指定元素索引 [1,3] 的位置为 18,[2,4] 的值为 3.6。

下面的代码顶一个一个两维 3×5 的 SparseTensor, 索引为 [0,1] 的位置的值为 6, [2,4] 位置的值为 0.5, 其他值为 0。

```

1 sparse_tensor = tf.SparseTensor(indices=[[0, 1], [2, 4]],
2                                 values=[6, 0.5],
3                                 dense_shape=[3, 5])
```

对应的 tensor:

```
1 [[0, 6, 0, 0, 0]
2 [0, 0, 0, 0, 0]
3 [0, 0, 0, 0, 0.5]]
```

2.7.4 传递 input_fn 数据到你的模型

为了输入数据给你的模型训练，你简单的传递你创建的输入函数给你的 train 操作:

```
1 classifier.train(input_fn=my_input_fn, steps=2000)
```

注意 input_fn 参数必须接受一个函数对象（例如 input_fn=input_fn），这意味着如果你在训练调用的时候传递参数给你的 input_fn，不是函数调用的返回值，正如下面的代码一样，你将得到 TypeError:

```
1 classifier.train(input_fn=my_input_fn(training_set), steps=2000)
```

然而如果你想参数化你的输入函数，有其它的方法能做到，你可以实现一个包装器函数不接受参数 input_fn 用它实现你想要的参数输入函数。

```
1 def my_input_fn(data_set):
2     ...
3
4 def my_input_fn_training_set():
5     return my_input_fn(training_set)
6
7 classifier.train(input_fn=my_input_fn_training_set, steps=2000)
```

你同样可以用 Python 的 `function.ppartial` 函数构造一个新的参数固定的函数对象。

```
1 classifier.train(
2     input_fn=functools.partial(my_input_fn, data_set=training_set),
3     steps=2000)
```

第三个选择是用 lambda 表达式包装你的 input_fn 函数传递它给你的 input_fn 参数:

```
1 classifier.train(input_fn=lambda: my_input_fn(training_set), steps=2000)
```

用上面的方法的一个很大的好处是为你的数据集接受参数，你可以通过改变数据集参数传递相同的 input_fn 函数给 evaluate 和 prediction 操作:

```
1 classifier.evaluate(input_fn=lambda: my_input_fn(test_set), steps=2000)
```

这种方法加强的代码的维护性: 不需要定义多的 input_fn 函数 (例如 input_fn_train,input_fn_test,input_fn_给每个操作，最终你可以用 `tf.estimator.inputs` 中的方法从 numpy 或者 pandas 数据集创

建 input_fn。另一个好处是你可以用更多的参数，像 num_epochs 和 shuffle 控制 input_fn 如何在数据上迭代，

```

1 import pandas as pd
2
3 def get_input_fn_from_pandas(data_set, num_epochs=None, shuffle=True):
4     return tf.estimator.inputs.pandas_input_fn(
5         x=pd.DataFrame(...),
6         y=pd.Series(...),
7         num_epochs=num_epochs,
8         shuffle=shuffle)

```

```

1 import numpy as np
2
3 def get_input_fn_from_numpy(data_set, num_epochs=None, shuffle=True):
4     return tf.estimator.inputs.numpy_input_fn(
5         x={...},
6         y=np.array(...),
7         num_epochs=num_epochs,
8         shuffle=shuffle)

```

2.7.5 波士顿房价的神经网络模型

接下来的导航，你将写输入函数处理从[UCI Housing Data Set](#)获取的数据集的子集，传递数据给神经网络回归器预测房价。你讲用于训练的神经网络包含下面的子集[Boston CSV](#)

data sets 包含下面特征数据

特征	描述
CRIM	人均犯罪率
ZN	居住地面积划分为 25000 平方英尺一块
INDUS	非商业用地的一部分
NOX	一氧化氮的浓度为千万分之一
RM	每个房子的房间数
AGE	1940 年前自有居民的比例
DIS	离波士顿就业中心的距离
TAX	每 10000 美元的税率
PTRATIO	学生老师的比率

2.7.6 建立

下载数据集[boston_train.csv](#),[boston_test.csv](#)和[boston_predict.csv](#)

2.7.7 导入的房子数据

```
1 from __future__ import absolute_import
2 from __future__ import division
3 from __future__ import print_function
4
5 import itertools
6
7 import pandas as pd
8 import tensorflow as tf
9
10 tf.logging.set_verbosity(tf.logging.INFO)
```

给 COLUMNS 中的数据定义名字，区别于标签中的特征，定义 FEATURES 和 LABEL，读入 CSV 文件到 pandas DataFrame:

2.7.8 定义特征列创建回归器

下一步是为输入数据创建 FeatureColumn，数据的格式指定用于训练的特征集，因为所有在房价数据集中的特征包含连续的值，你可以用 `tf.contrib.layers.real_valued_column()` 创建他们的 FeatureColumn：

```
1 feature_cols = [tf.feature_column.numeric_column(k) for k in FEATURES]
```

现在初始化一个神经网络回归模型的实体 DNNRegressor,你需要提供两个参数:`hidden_units` 指定每个隐藏层的节点数 (这里的两层, 每层 10 个节点) 和 `feature_columns`: 包含 `FeatureColumns`

2.7.9 构建 input_fn

传递输入数据给 regressor, 写一个 factory 方法接受 pandas DataFrame 返回一个 input_fn:

```

1 def get_input_fn(data_set, num_epochs=None, shuffle=True):
2     return tf.estimator.inputs.pandas_input_fn(
3         x=pd.DataFrame({k: data_set[k].values for k in FEATURES}),
4         y = pd.Series(data_set[LABEL].values),
5         num_epochs=num_epochs,
6         shuffle=shuffle)

```

注意输入数据被传递给 input_fn 的 data_set 参数, 这意味着函数可以处理任何你导入的的 DataFrame:training_set,test_set 和 prediction_set。提供两个额外的参数 num_epochs(控制在数据上的迭代次数) 训练的时候设置为 None, 因此 input_fn 保持返回值知道训练步数到达, 为了平局和测试设置为 1, 因此 input_fn 将在数据上迭代然后抛出OutOfRangeError, 错误将通知 Estimator 停止评估或者预测:shuffle (是否打乱数据)。对于评估和预测, 设置为 False, 因此 input_fn 在数据上顺序迭代, 对于训练设置为 True。

2.7.10 训练回归器

为了训练神经网络回归器, 用 training_set 传递给 input_fn 运行 train:

```

1 regressor.train(input_fn=get_input_fn(training_set), steps=5000)

```

你应该能看到类似的输出, 每 100 步报告训练的损失:

```

1 INFO:tensorflow:Step 1: loss = 483.179
2 INFO:tensorflow:Step 101: loss = 81.2072
3 INFO:tensorflow:Step 201: loss = 72.4354
4 ...
5 INFO:tensorflow:Step 1801: loss = 33.4454
6 INFO:tensorflow:Step 1901: loss = 32.3397
7 INFO:tensorflow:Step 2001: loss = 32.0053
8 INFO:tensorflow:Step 4801: loss = 27.2791
9 INFO:tensorflow:Step 4901: loss = 27.2251
10 INFO:tensorflow:Saving checkpoints for 5000 into /tmp/boston_model/model.ckpt.
11 INFO:tensorflow:Loss for final step: 27.1674.

```

2.7.11 评估模型

下一步看看模型在测试数据及上的性能, 运行 evaluate, 传递 test_set 到 input_fn:

```

1 ev = regressor.evaluate(
2     input_fn=get_input_fn(test_set, num_epochs=1, shuffle=False))

```

从 ev 结果返回损失的，打印：

```

1 loss_score = ev["loss"]
2 print("Loss: {:.f}".format(loss_score))

```

你应该能看到下面的结果：

```

1 INFO:tensorflow:Eval steps [0,1) for training step 5000.
2 INFO:tensorflow:Saving evaluation summary for 5000 step: loss = 11.9221
3 Loss: 11.92209

```

2.7.12 做出预测

最后你可以用模型在给定的预测包含特征数据没有标签的数据集上预测房价

```

1 y = regressor.predict(
2     input_fn=get_input_fn(prediction_set, num_epochs=1, shuffle=False))
3 # .predict() returns an iterator of dicts; convert to a list and print
4 # predictions
5 predictions = list(p["predictions"] for p in itertools.islice(y, 6))
6 print("Predictions: {}".format(str(predictions)))

```

你应该得到包含 6 个房价值 (单位是千美元)

```

1 Predictions: [ 33.30348587  17.04452896  22.56370163  34.74345398  14.55953979
2   19.58005714]

```

```

1 #tensorflow 1.2.1
2 import tensorflow as tf
3 var = tf.Variable(0)
4 add_operation = tf.add(var, 1)
5 update_operation = tf.assign(var, add_operation)
6 with tf.Session() as sess:
7     sess.run(tf.global_variables_initializer())
8     for _ in range(3):
9         sess.run(update_operation)
10        print(sess.run(var))

```

2.7.13 batch normalization

§ 数据 x 为 Tensor。

- mean: 为 x 的均值, 也是一个 Tensor。
 - var: 为 x 的方差, 也为一个 Tensor。
 - offset: 一个偏移, 也是一个 Tensor。
 - scale: 缩放倍数, 也是一个 Tensor。
 - variable_epsilon, 一个不为 0 的浮点数。
 - name: 操作的名字, 可选。

batch normalization 计算方式是：

$$x = (x - \bar{x}) / \sqrt{Var(x) + variable_{epsilon}} \quad (2.1)$$

$$x = x \times scale + offset \quad (2.2)$$

(2.3)

$$\text{均值} : \bar{x} = \frac{1}{m} \sum_{i=1}^m x_i \quad (2.4)$$

$$\text{方差: } \sigma^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \bar{x})^2 \quad (2.5)$$

2.8 常见的激活函数

2.8.1 relu

relu 函数在自变量 x 小于 0 时值全为 0, 在 x 大于 0 时, 值和自变量相等。

```

15     )
16 ax.annotate("",xy=(6,6),xycoords='data',
17             xytext=(10, 6), textcoords='data',
18             arrowprops=dict(arrowstyle="->",
19                             connectionstyle="arc3"),
20
21 )
22 ax.grid(True)
23 plt.xlabel('x')
24 plt.ylabel('relu(x)')
25 plt.savefig('relu.png',dpi = 600)

```

2.8.2 relu6

relu6 函数和 relu 不同之处在于在 x 大于等于 6 的部分值保持为 6。

```

1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3 x = tf.linspace(-10.,10.,100)
4 y = tf.nn.relu6(x)
5 with tf.Session() as sess:
6     [x,y] = sess.run([x,y])
7 plt.plot(x,y,'r',6,6,'bo')
8 plt.title('relu6')
9 ax = plt.gca()
10 ax.annotate("",,
11             xy=(6, 6), xycoords='data',
12             xytext=(6, 4.5), textcoords='data',
13             arrowprops=dict(arrowstyle="->",
14                             connectionstyle="arc3"),
15
16 )
17 ax.grid(True)
18 plt.xlabel('x')
19 plt.ylabel('relu6(x)')
20 plt.savefig('relu6.png',dpi = 600)

```

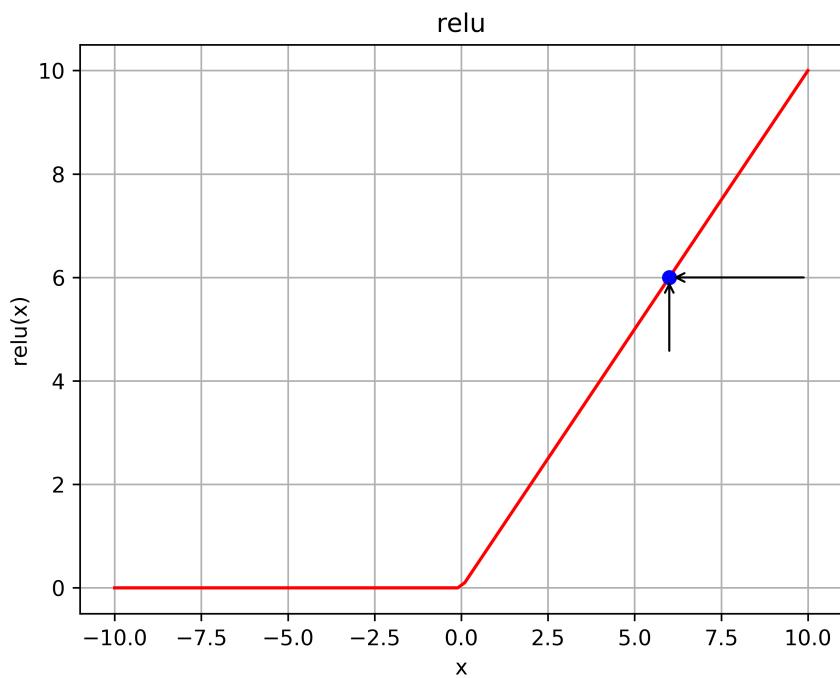


图 2.1: relu

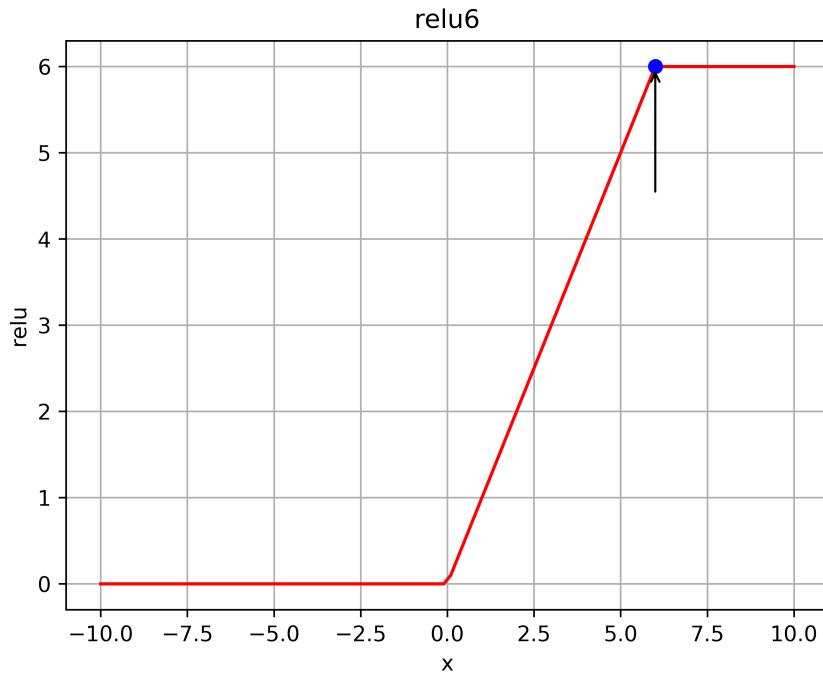


图 2.2: relu6

2.8.3 sigmoid

```

1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3 import matplotlib.patches as mpatches
4 x = tf.linspace(-10.,10.,100)
5 y1 = tf.nn.sigmoid(x)
6 y2 = tf.nn.tanh(x)
7 red_patch = mpatches.Patch(color = 'red',label = 'sigmoid')
8 blue_patch = mpatches.Patch(color = 'blue',label = 'tanh')
9 with tf.Session() as sess:
10     [x,y1,y2] = sess.run([x,y1,y2])
11     plt.plot(x,y1,'r',x,y2,'b')
12     ax = plt.gca()
13     ax.annotate(r"\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}",
14                 xy=(0,0),xycoords="data",
15                 xytext=(1,0),textcoords="data",
16                 arrowprops=dict(arrowstyle="->",
17                                 connectionstyle="arc3"),
18 )
19     ax.annotate(r"\text{sigmoid}(x) = \frac{1}{1+e^{-x}}",

```

```

20     xy=(0, 0.5), xycoords="data",
21     xytext=(1, 0.5), textcoords="data",
22     arrowprops=dict (arrowstyle="->",
23     connectionstyle="arc3"),
24 )
25 plt.xlabel('x')
26 plt.grid(True)
27 plt.legend(handles = [red_patch, blue_patch])
28 plt.savefig('activate.png', dpi=600)

```

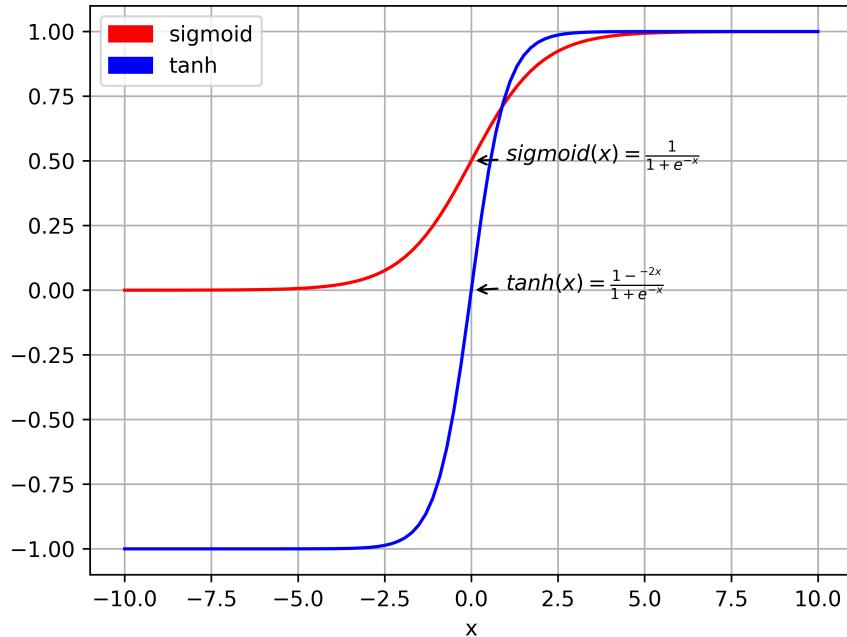


图 2.3: activate_fun

2.8.4 relu 和 softplus

```

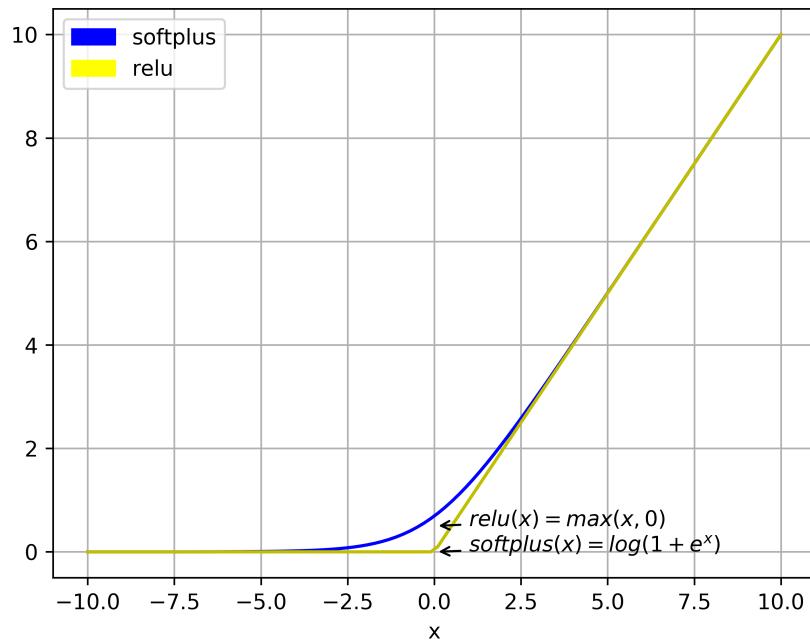
1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3 import matplotlib.patches as mpatches
4 x = tf.linspace(-10., 10., 100)
5 y2 = tf.nn.softplus(x)
6 y3 = tf.nn.relu(x)
7 blue_patch = mpatches.Patch(color = 'blue', label = 'softplus')
8 yellow_patch = mpatches.Patch(color = 'yellow', label = 'relu')

```

```

9  with tf.Session() as sess:
10    [x,y2,y3] = sess.run([x,y2,y3])
11    plt.plot(x,y2,'b',x,y3,'y')
12    ax = plt.gca()
13    plt.xlabel('x')
14    ax.annotate(r"$softplus(x)=\log(1+e^x)$",
15                xy=(0,0),xycoords="data",
16                xytext=(1,0),textcoords="data",
17                arrowprops=dict(arrowsize=2,
18                                connectionstyle="arc3"),
19            )
20    ax.annotate(r"$relu(x)=\max(x, 0)$",
21                xy=(0,0.5),xycoords="data",
22                xytext=(1,0.5),textcoords="data",
23                arrowprops=dict(arrowsize=2,
24                                connectionstyle="arc3"),
25            )
26
27    plt.grid(True)
28    plt.legend(handles = [blue_patch,yellow_patch])
29    plt.savefig('relu_softplus.png',dpi=600)

```



2.8.5 dropout

将神经元以概率 keep_prob 绝对是否被抑制。如果被抑制该神经元的输出为 0 如果不被抑制,该神经元的输出将被放大到原来的 $1/\text{keep_prop}$ 。默认情况下,每个神经元是否被抑制是相互独立的。但是是否被抑制也可以通过 noise_shape 来调节。当 noise_shape[i]=shape(x)[i] 时,x 中的元素相互独立。如果 shape(x)=[k,1,1,n], 那么每个批通道都是相互独立的, 但是每行每列的数据都是关联的, 也就是说要么都为 0, 要么还是原来的值。

```

1 import tensorflow as tf
2 a = tf.constant([[-1., 2., 3., 4.]])
3 with tf.Session() as sess:
4     b = tf.nn.dropout(a, 0.5, noise_shape=[1, 4])
5     print(sess.run(b))
6     c = tf.nn.dropout(a, 0.5, noise_shape=[1, 1])
7     print(sess.run(c))

```

$[[-2. \quad 0. \quad 0. \quad 8.]]$

$[[-0. \quad 0. \quad 0. \quad 0.]]$

当输入数据特征相差明显时, 用 tanh 效果会很好, 但在循环过程中会不断扩大特征效果并显示出来。当特征相差不明显时, sigmoid 效果比较好。同时, 用 sigmoid 和 tanh 作为激活函数时, 需要对输入进行规范化, 否则激活厚的值全部进入平坦区, 隐藏层的输出会趋同, 丧失原来的特征表达, 而 relu 会好很多, 优势可以不需要输入规范化来避免上述情况。因此, 现在大部分卷积神经网络都采用 relu 作为激活函数。

2.9 CNN 常用函数

2.9.1 卷积函数

`tf.nn.conv2d(input,filter,padding,stride=None,diation_rate=Nonei 每 name = None,data_format=None)`

- input: 一个 tensor, 数据类型必须是 float32, 或者是 float64
- filter: 一个 tensor, 数据类型必须和 input 相同。
- strides: 一个长度为 4 的一组证书类型数组, 每一维对应 input 中每一维对应移动的步数, strides[1] 对应 input[1] 移动的步数。
- padding: 有两个可选参数'VALID' (输入数据维度和输出数据维度不同) 和'SAME' (输入数据维度和输出数据维度相同)
- use_cudnn_on_gpu: 一个可选的布尔值, 默认情况下时 True。

- name: 可选，操作的一个名字。

```

1 import tensorflow as tf
2 input_data = tf.Variable(tf.random_normal(shape = [10,9,9,3],mean=0,stddev=1),
3                         dtype = tf.float32)
4 kernel = tf.Variable(tf.random_normal(shape = [2,2,3,2],mean = 0,stddev=1,dtype=
5                         tf.float32))
6
7 y = tf.nn.conv2d(input_data,kernel,strides=[1,1,1,1],padding='SAME')
8 init = tf.global_variables_initializer()
9 with tf.Session() as sess:
10     sess.run(init)
11     print(sess.run(y).shape)

```

输出形状为 [10,9,9,2]。

2.9.2 常见的分类函数

`tf.nn.sigmoid_cross_entropy_with_logits(logits,targets,name=None)`

- logits:[batch_size,num_classes]
- targets:[batch_size,size]
- 输出: loss[batch_size,num_classes]

最后已成不需要进行 sigmoid 操作。

`tf.nn.softmax(logits,dim=-1,name=None)`: 计算 Softmax

$$\text{softmax} = \frac{x^{\logits}}{\text{reduce_sum}(e^{\logits}, dim)}$$

`tf.nn.log_softmax(logits,dim=-1,name = None)` 计算 log softmax

$$\text{logsoftmax} = \logits - \log(\text{reduce_softmax}(\exp(\logits), dim))$$

`tf.nn.softmax_cross_entropy_with_logits(_sentinel=None,labels=None,logits=None,dim=-1,name=None)` 输出 loss:[batch_size] 保存的时 batch 中每个样本的交叉熵。`tf.nn.sparse_softmax_cross_entropy`

- logits: 神经网络最后一层的结果。
- 输入 logits:[batch_size,num_classes],labels:[batch_size], 必须在 [0,num_classes]
- loss[batch], 保存的是 batch 每个样本的交叉熵。

2.10 优化方法

- `tf.train.GradientDescentOptimizer`
- `tf.train.AdadeltaOptimizer`
- `tf.train.AdagradDAOptimizer`
- `tf.train.AdagradOptimizer`
- `tf.train.MomentumOptimizer`
- `tf.train.AdamOptimizer`
- `tf.train.FtrlOptimizer`
- `tf.train.RMSPropOptimizer`

2.10.1 BGD

BGD(batch gradient descent) 批量梯度下降。这种方法是利用现有的参数对训练集中的每一个输入生成一个估计输出 y_i , 然后跟实际的输出 y_i 比较, 统计所有的误差, 求平均后的到平均误差作为更新参数的依据。啊他的迭代过程是:

1. 提取训练集集中所有内容 $\{x_1, \dots, x_n\}$, 以及相关的输出 y_i ;
2. 计算梯度和误差并更新参数。

这种方法的优点是: 使用所有数据计算, 都保证收敛, 并且不需要减少学习率。缺点是每一步需要使用所有的训练数据, 随着训练的进行, 速度会变慢。那么如果将训练数据拆分成一个个 batch, 每次抽取一个 batch 数据更新参数, 是不是能加速训练? 这就是 SGD。

2.10.2 SGD

SGD(stochastic gradient descent): 随机梯度下降。这种方法的主要思想是将数据集拆分成一个个的 batch, 随机抽取一个 batch 计算并更新参数, 所以也称为 MBGD(minibatch gradient descent) SGD 在每次迭代计算 mini-batch 的梯度, 然后对参数进行更新。和 BGD 相比, SGD 在训练数据集很大时也能以较快的速度收敛, 但是它有两个缺点:

1. 需要手动调整学习率, 此外选择合适的学习率比较困难。尤其在训练时, 我们常常想对常出现的特征更快速的更新, 对不常出现的特征更新速度慢些, 而 SGD 更新参数时对所有参数采用一样的学习率, 因此无法满足要求。
2. SGD: 容易收敛到局部最优。

2.10.3 momentum

Momentum 是模拟物理学中的动量概念，更新时在一定程度上保留之前的更新方向，利用当前批次再次微调本次更新参数，因此引入了一个新的变量 v ，作为前几次梯度的累加。因此，momentum 能够更新学习率，在下降初期，前后梯度方向一致时能加速学习；在下降的中后期，在局部最小值附近来回振荡，能够抑制振荡加快收敛。

2.10.4 Nesterov Momentum

标准的 Momentum 法首先计算一个梯度，然后在加速更新梯度的方向进行一个大的跳跃 Nesterov 首先在原来加速的梯度方向进行一个大的跳跃，然后在改为值设置计算梯度值，然后用这个梯度值修正最终的更新方向。

2.10.5 Adagrad

Adagrade 能够自适应的为各个参数分配不同的学习率，能够控制每个维度的梯度方向，这种方法的优点是能实现学习率的自动更改，如果本次更新时梯度大，学习率就衰减得快，如果这次更新时梯度小，学习率衰减得就慢些。

2.10.6 RMSprop

和 Momentum 类似，通过引入衰减系数使得每个回合都衰减一定比例。在实践中，对循环神经网络效果很好。

2.10.7 Adam

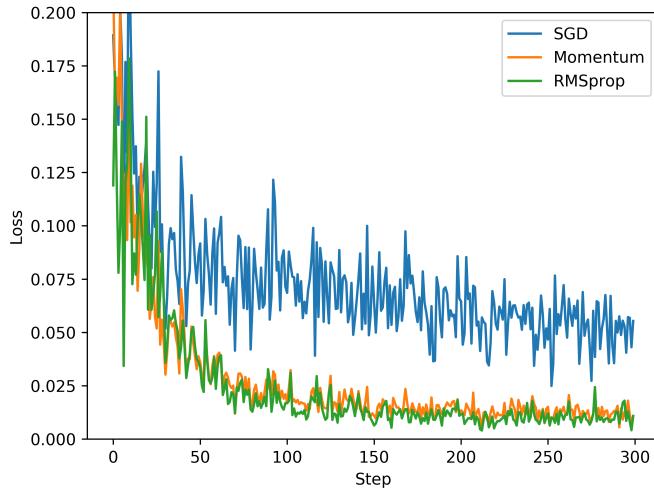
名称来自自适应矩阵 (adaptive moment estimation).Adam 根据损失函数针对每个参数的一阶矩，二阶矩估计动态调整每个参数的学习率。

```

1 import numpy as np
2 import tensorflow as tf
3 import matplotlib.pyplot as plt
4 tf.set_random_seed(0)
5 np.random.seed(0)
6 LR = 0.01
7 BATCH_SIZE = 32
8 x = np.linspace(-1, 1, 100).reshape(-1, 1)
9 noise = np.random.normal(0, 0.1, size=x.shape)
10 y = np.power(x, 2)+noise
11 class Net:
12     def __init__(self, opt, **kwargs):
13         self.x = tf.placeholder(tf.float32, [None, 1])

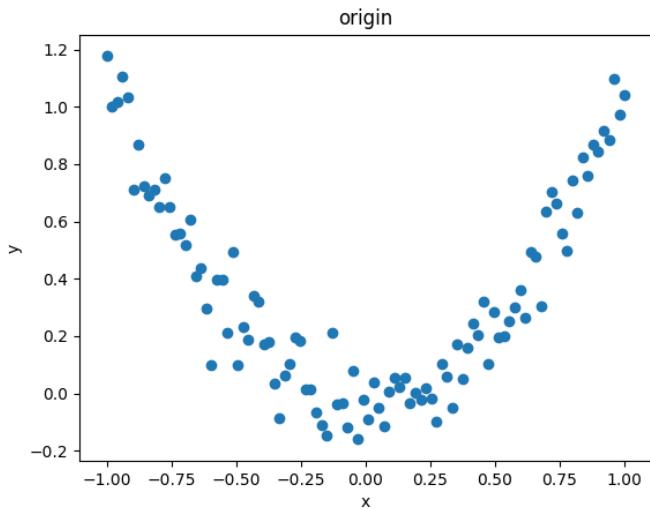
```

```
14     self.y = tf.placeholder(tf.float32,[None,1])
15     l = tf.layers.dense(self.x,20,tf.nn.relu)
16     out = tf.layers.dense(l,1)
17     self.loss = tf.losses.mean_squared_error(self.y,out)
18     self.train = opt(LR,**kwargs).minimize(self.loss)
19 net_SGD = Net(tf.train.GradientDescentOptimizer)
20 net_momentum = Net(tf.train.MomentumOptimizer,momentum=0.9)
21 net_RMSprop = Net(tf.train.RMSPropOptimizer)
22 net_Adam = Net(tf.train.AdamOptimizer)
23 nets = [net_SGD,net_momentum,net_RMSprop,net_Adam]
24 sess = tf.Session()
25 sess.run(tf.global_variables_initializer())
26 losses_his = [[],[],[]]
27 for step in range(300):
28     index = np.random.randint(0,x.shape[0],BATCH_SIZE)
29     b_x = x[index]
30     b_y = y[index]
31     for net,l_his in zip(nets,losses_his):
32         _,l = sess.run([net.train,net.loss],{net.x:b_x,net.y:b_y})
33         l_his.append(l)
34 labels = ['SGD','Momentum','RMSprop','Adam']
35 for i,l_his in enumerate(losses_his):
36     plt.plot(l_his,label=labels[i])
37 plt.legend(loc='best')
38 plt.xlabel('Step')
39 plt.ylabel('Loss')
40 plt.ylim(0,0.2)
41 plt.savefig('Opt.png',dpi=600)
```



2.10.8 构造简单的神经网络拟合数据

原始数据为 $y = x^2$ 的基础上添加随机噪声。原始数据的散点图如下



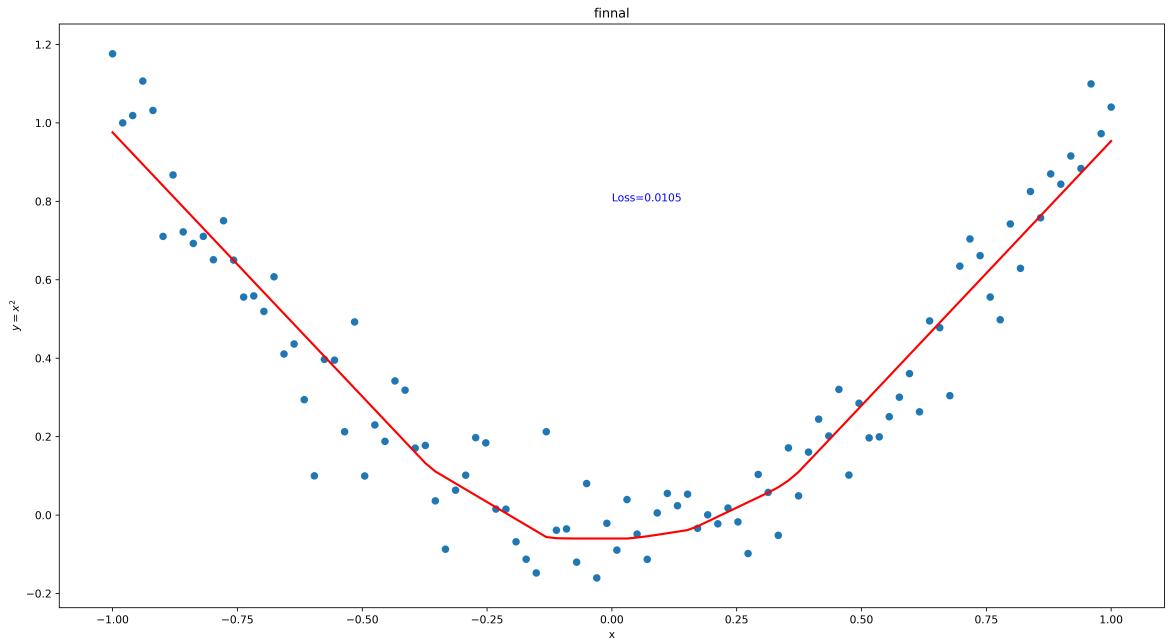
```

1 #tensorflow 1.2.1
2 import tensorflow as tf
3 import matplotlib.pyplot as plt
4 import numpy as np
5 tf.set_random_seed(0)
6 np.random.seed(0)

```

```
7 #生成数据
8 step = 100
9 x = np.linspace(-1,1,step).reshape(-1,1)
10 noise = np.random.normal(0,0.1,size=x.shape)
11 y = np.power(x,2)+noise
12
13 tf_x = tf.placeholder(tf.float32,x.shape)
14 tf_y = tf.placeholder(tf.float32,x.shape)
15 l1 = tf.layers.dense(tf_x,10,tf.nn.relu)
16 output = tf.layers.dense(l1,1)
17
18 loss = tf.losses.mean_squared_error(tf_y,output)
19 optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5)
20 train_op = optimizer.minimize(loss)
21
22 sess = tf.Session()
23 sess.run(tf.global_variables_initializer())
24 plt.ion()
25 for step in range(100):
26     _,l,pred = sess.run([train_op,loss,output],{tf_x:x,tf_y:y})
27     if step%5==0:
28         plt.cla()
29         plt.scatter(x,y)
30         plt.title(r'$y=x^2+noise$')
31         plt.plot(x,pred,'r-',lw=2)
32         plt.text(0,0.8,'Loss=%f' % l,fontdict={'size':10,'color':'blue'})
33         plt.xlabel("x")
34         plt.ylabel(r"$y=x^2$")
35         plt.pause(0.1)
36 plt.ioff()
37 plt.show()
```

最终拟合数据:



2.11 TensorBoard

```

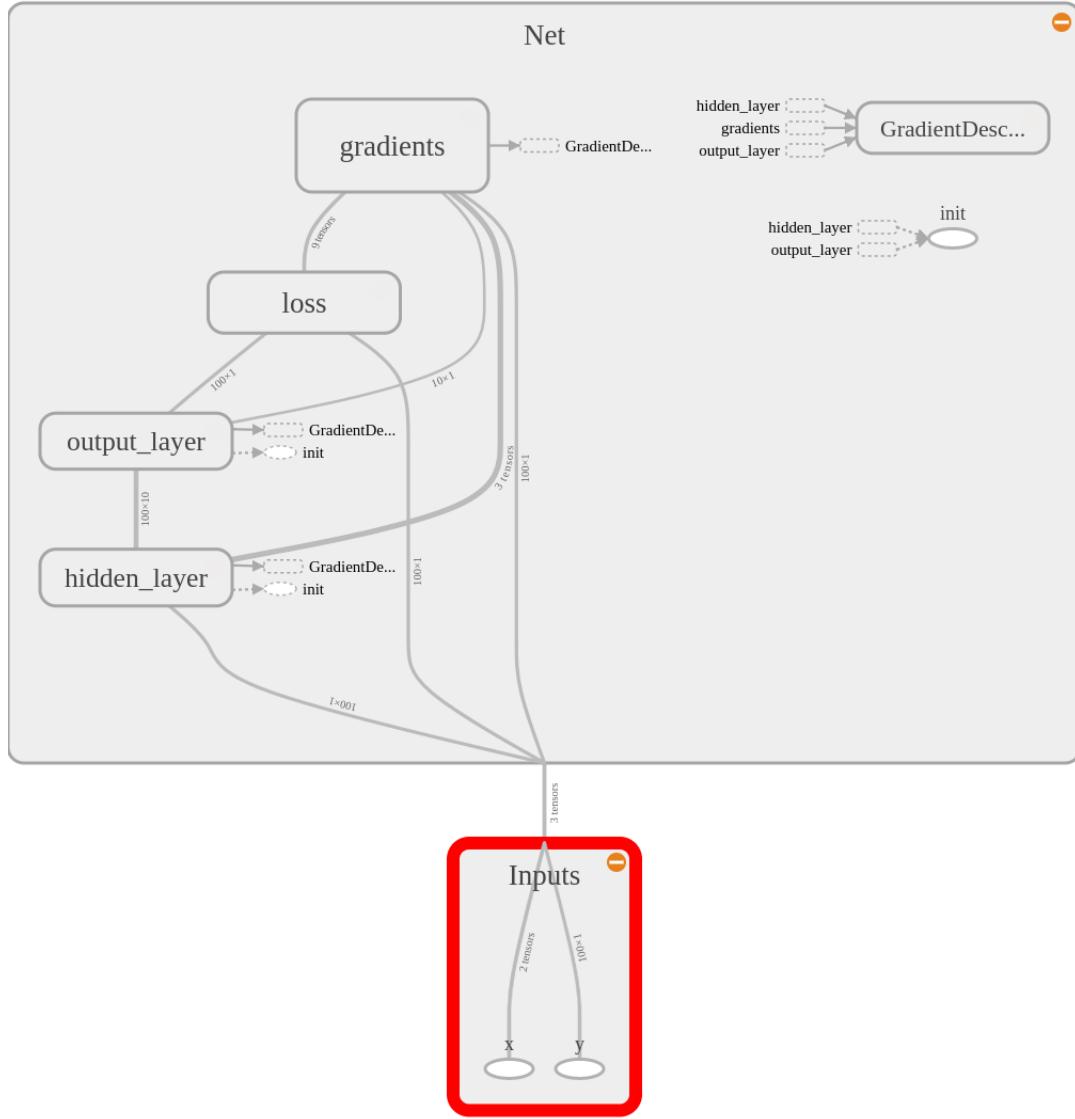
1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3
4 tf.set_random_seed(1)
5 x0 = tf.random_normal((100,2),2,2,tf.float32,0)
6 y0 = tf.zeros(100)
7 x1 = tf.random_normal((100,2),-2,2,tf.float32,0)
8 y1 = tf.ones(100)
9 x = tf.reshape(tf.stack((x0,x1),axis=1),(200,2))
10 y = tf.reshape(tf.stack((y0,y1),axis=1),(200,1))
11 with tf.Session() as sess:
12     x = sess.run(x)
13     y = sess.run(y)
14
15 tf_x = tf.placeholder(tf.float32, x.shape)      # input x
16 tf_y = tf.placeholder(tf.int32, y.shape)        # input y
17
18 # neural network layers
19 l1 = tf.layers.dense(tf_x, 10, tf.nn.relu)       # hidden layer

```

```

20 output = tf.layers.dense(11, 2)                                # output layer
21
22 loss = tf.losses.sparse_softmax_cross_entropy(labels=tf_y, logits=output)
23                                         # compute cost
23 accuracy = tf.metrics.accuracy(          # return (acc, update_op), and create 2
24                                         local variables
24                                         labels=tf.squeeze(tf_y), predictions=tf.argmax(output, axis=1),)[1]
25 optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.05)
26 train_op = optimizer.minimize(loss)
27
28 sess = tf.Session()
29
30                                         # control training and others
31 init_op = tf.group(tf.global_variables_initializer(), tf.
32                                         local_variables_initializer())
33 sess.run(init_op)      # initialize var in graph
34
35 plt.ion()    # something about plotting
36 for step in range(100):
37     _, acc, pred = sess.run([train_op, accuracy, output], {tf_x: x, tf_y: y})
38     if step % 2 == 0:
39         plt.cla()
40         plt.scatter(x[:, 0], x[:, 1], c=pred.argmax(1), s=100, lw=0, cmap='RdYlGn')
41         plt.text(1.5, -4, 'Accuracy=% .2f' % acc, fontdict={'size': 20, 'color': 'red'})
42         plt.pause(0.1)
43 plt.ioff()
44 plt.show()

```



2.11.1 TensorBoard Histogram Dashboard

TensorBoard Histogram Dashboard 显示 TensorFlow 图中的 Tensor 如何随着时间变化。

2.11.2 一个简单的例子

正态分布变量，均值随着和时间移动。TensorFlow 有一个操作 `tf.random_normal` 可以完美的达到这个目的。正如通常情况下 TensorBoard，我们将用 `summary op` 融合数据

据。在这种情况下'tf.summary.histogram'。这里有一个代码段将生成一些包含正态分布直方图数据的总结，这里均值随着增大。

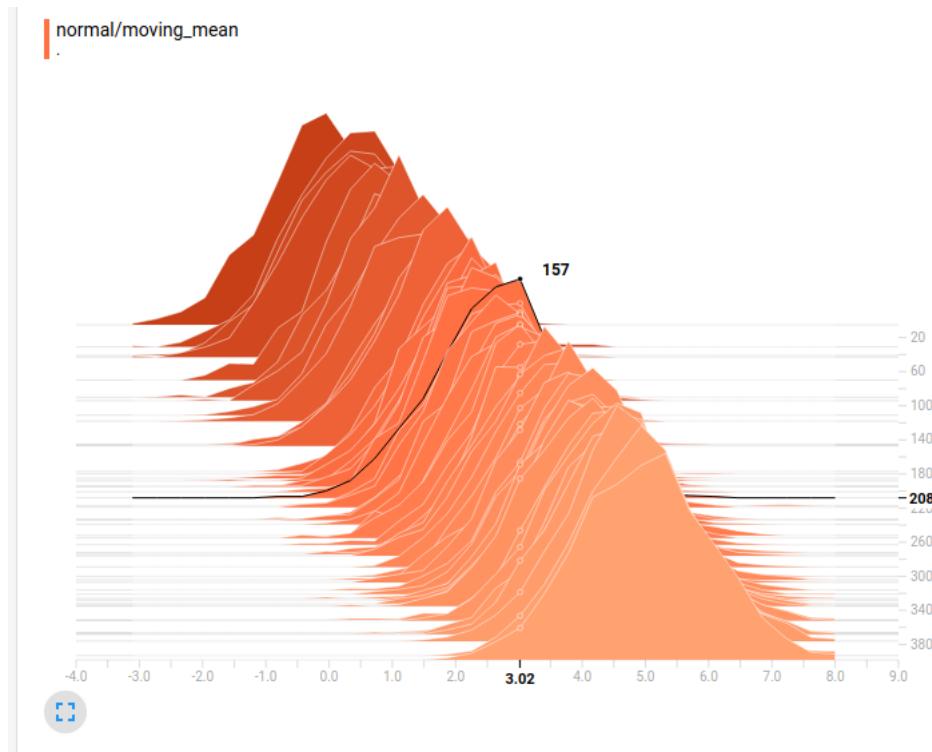
```

1 import tensorflow as tf
2 k = tf.placeholder(tf.float32)
3 mean_moving_normal = tf.random_normal(shape=[1000], mean=(5*k), stddev=1)
4 summaries = tf.summary.histogram('normal/moving_mean', mean_moving_normal)
5 sess = tf.Session()
6 writer = tf.summary.FileWriter('./histogram_example')
7 N = 400
8 for step in range(N):
9     k_val = step/float(N)
10    summ = sess.run(summaries, feed_dict={k:k_val})
11    writer.add_summary(summ, global_step=step)

```

在当前代码中运行下边的代码启动 TensorFlow 载入数据

```
1 tensorboard --logdir=./histogram_example
```



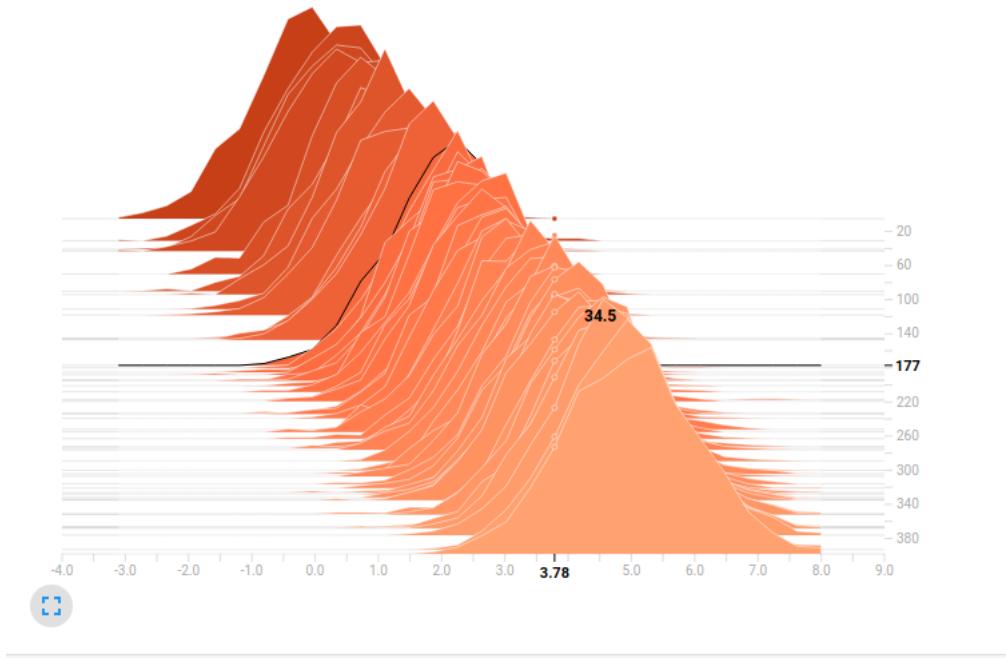
tf.summary.histogram 接受任一尺寸和大小的 Tensor，压缩他们进入直方数据结构组成一些小的数据宽度和数量组层的 bin 将该够，例如我们像组成数 [0.5,1.1,1.3,2.2,2.9,2.99] 成 3

个 bin，我们可以创建三个 bin：一个包含 0 到 1 之间的一切 (0.5)，一个包含 1-2(1.1,1.3) 之间，一个包含 2-3(2.2,2.9,2.99)

TensorFlow 用类是的方法创建 bins，但是不想我们上面的例子，它不创建整数读额 bins，瑞与大型数据，稀疏数据，这样的也许导致上千个 bin，bins 时指数分布时，一些 bins 相比于一些非常大数的 bin 接近于 0。然而，可视化指数分布 bin 时一个技巧，如果高被编码为数量，bin 宽度更大的空间，甚至他们有相同的元素，相比较之下统计数量使得豪赌比较变得可能，直方图采集数据仅均匀的 bins，这可能导致不幸的人工操作。

在直方图可视化器的每一个切片显示为一个单个的直方图。切片安装步数组组织。例老的切片 (e.g. step 0) 比较靠后变为更深，然而新的 slices 接近于前景色，颜色更轻，右边的 y 轴显示了步数。

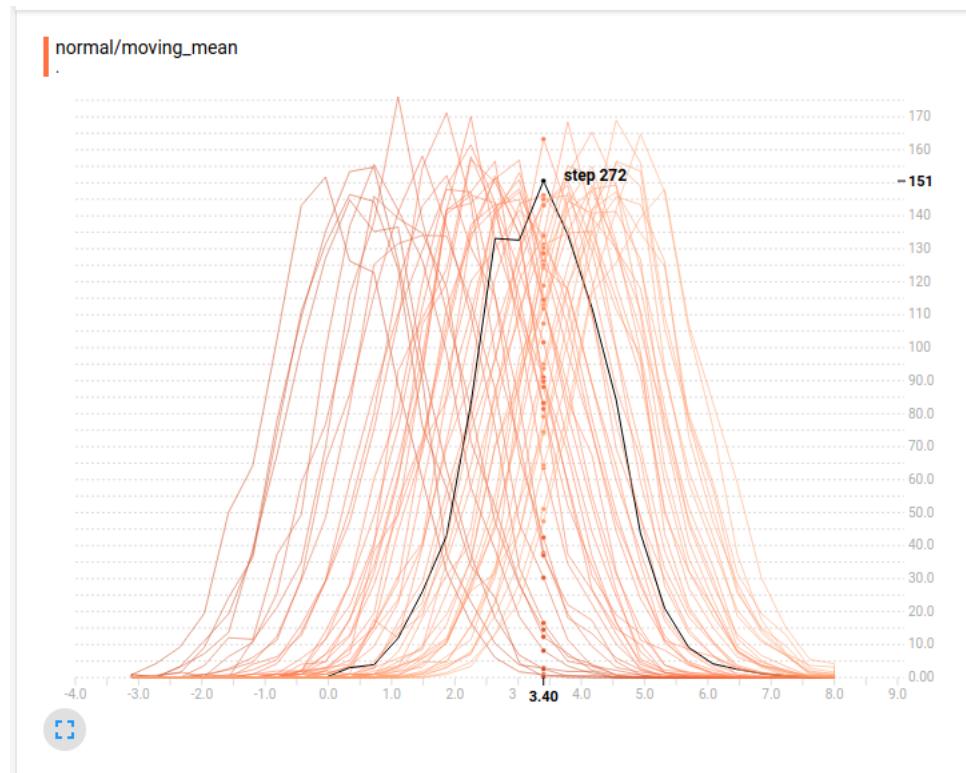
你可以在直方图上滑动鼠标看到更多的详细星系。你如下面的图你可以看到直方图的时间不为 177 有一个 bin 中心在 3.78 有 bin 中有 34.5 个元素。



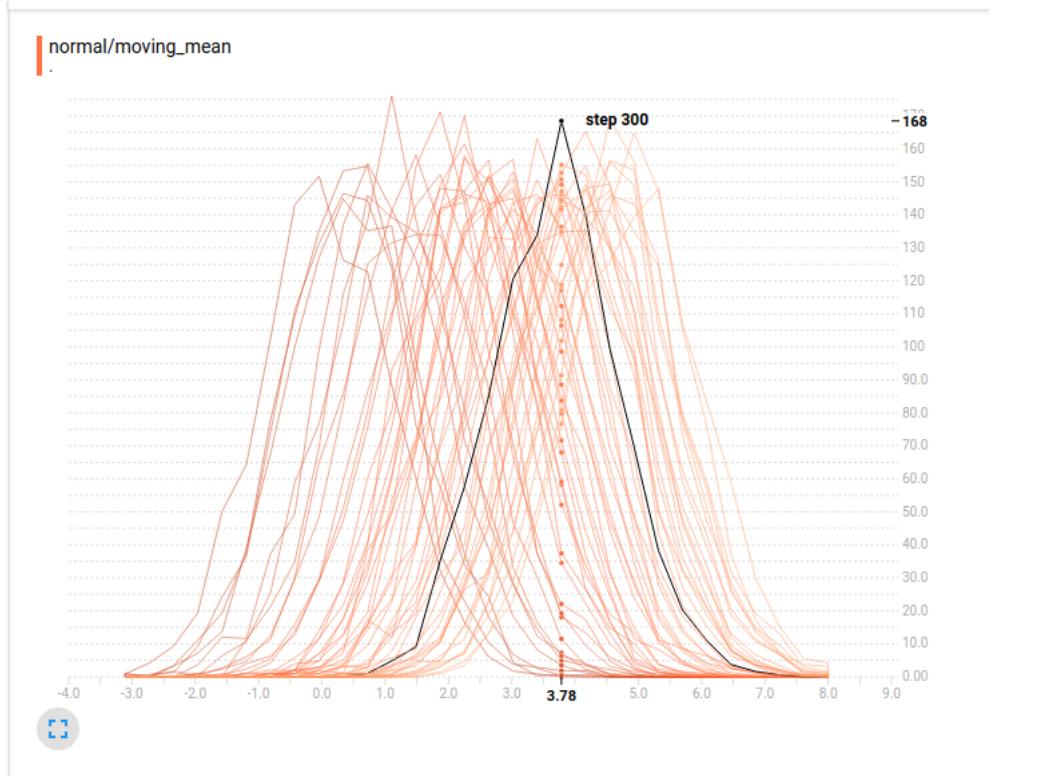
你也许注意到注意到直方切片在统计步数和时间上不总是偶数，这是因为 TensorBoard 用[reservoir sampling](#)保持直方图的子集，为了节约内存，Reservior sampling 保证每个采样有一个相等的可能性被包含进去，但是因为它时一个随机算法，采样并不在每个偶数步发生。

2.11.3 Overlay Mode

控制面板上允许你打开直方图模式为 offset 为 overlay。在 offset 模式下，可视化转动 45 度，因此单个的直方图切片不再展开，而是所有的图共享一个相同的 y 轴上。



现在表上的每个切片被线分开，y 轴显示每个 bucket 项目数量，深色线时老的，早期的时间不，浅色线时最近的新时间不，你可以用鼠标在表上查看更多的信息。



overlay 可视化在你想直接比较不同直方图的数量。

2.11.4 多个分布

直方图控制面板对多分布下的可视化很有用，当我们通过链接两个不同的正态分布构造一个简单的二两分布，代码如下：

```

1 import tensorflow as tf
2 k = tf.placeholder(tf.float32)
3 mean_moving_normal = tf.random_normal(shape=[1000], mean=(3*5), stddev=1)
4 tf.summary.histogram('normal/moving_mean', mean_moving_normal)
5 variance_shrinking_normal = tf.random_normal(shape=[100], mean=0, stddev=1-(k))
6 tf.summary.histogram('normal/shrinking_varance', variance_shrinking_normal)
7 normal_combined = tf.concat([mean_moving_normal, variance_shrinking_normal], 0)
8 tf.summary.histogram('normal/bimodal', normal_combined)
9 summaris = tf.summary.merge_all()
10 sess = tf.Session()
11 writer = tf.summary.FileWriter('./ histgram_example1')
12 N = 400
13 for step in range(N):

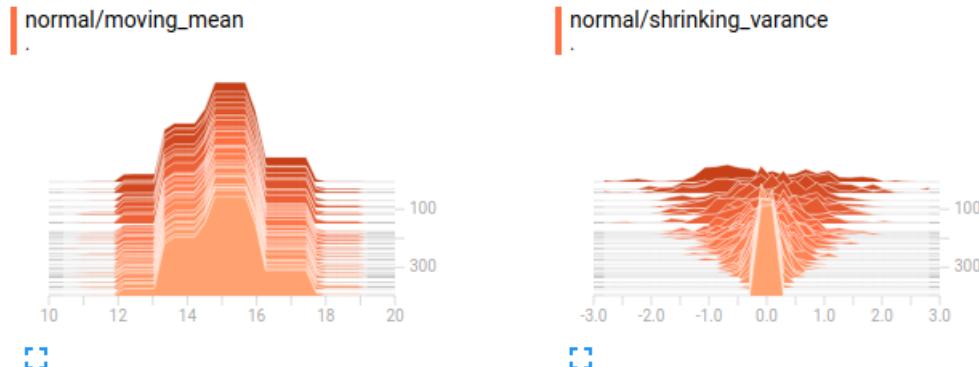
```

```

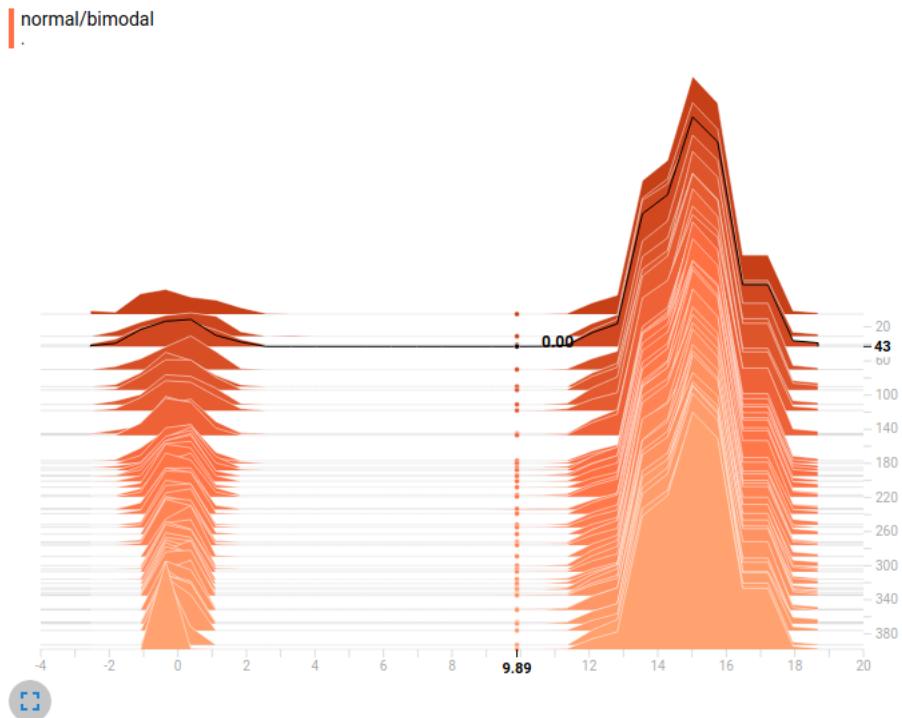
14     k_val = step / float(N)
15     summ = sess.run(summaris, feed_dict={k:k_val})
16     writer.add_summary(summ, global_step=step)

```

上面的例子是滑动平均，现在我们已有一个收缩的变量分布。



当我们链接她们在一起，我们得到一个清晰解释分歧，二进制结构的表格:



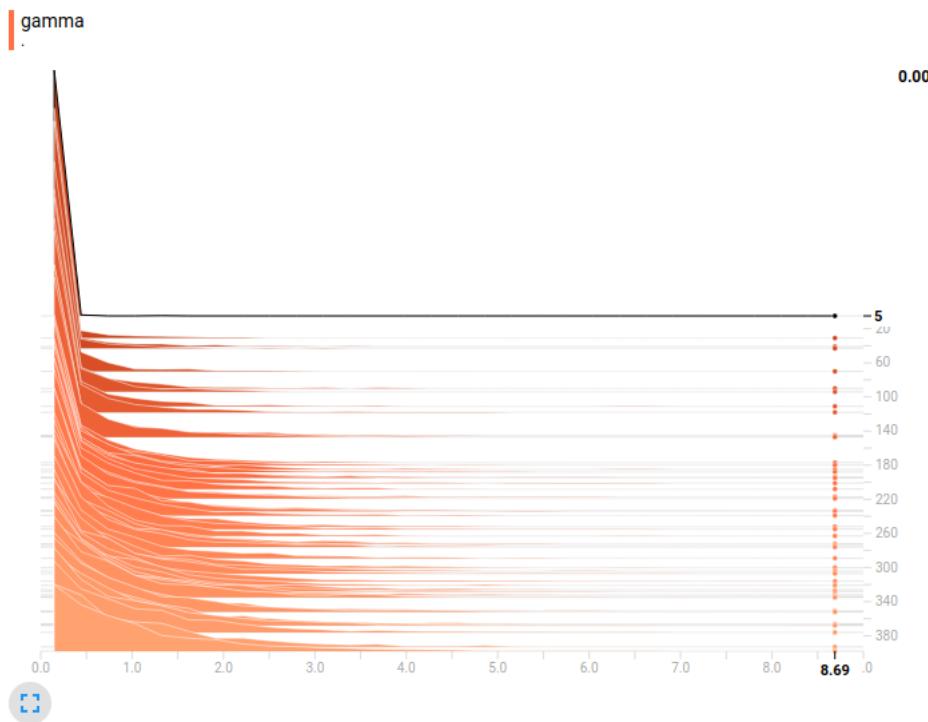
2.11.5 更多分布

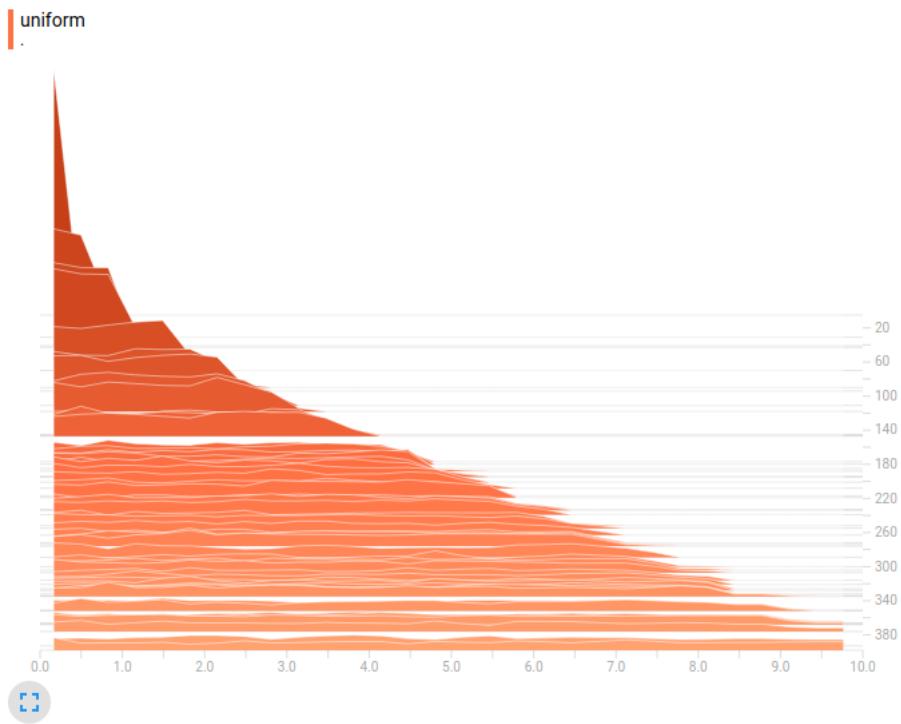
生成可视化更多分布，结合他们到表中：

```

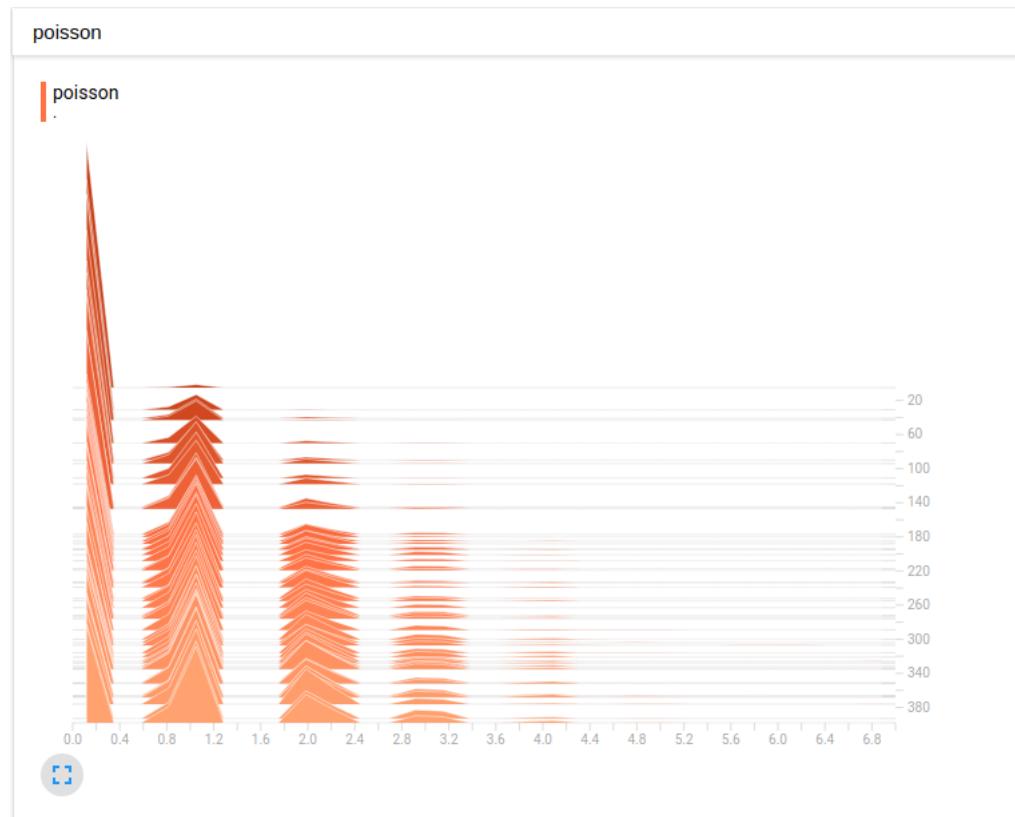
1 import tensorflow as tf
2 k = tf.placeholder(tf.float32)
3 # Make a normal distribution ,with a shift mean
4 mean_moving_normal = tf.random_normal(shape=[1000],mean=(5*k),stddev=1)
5 tf.summary.histogram('normal/moving_mean',mean_moving_normal)
6 variance_shrinking_normal = tf.random_normal(shape=[1000],mean=0,stddev=1-(k))
7 tf.summary.histogram('normal/shrinking_variance',variance_shrinking_normal)
8 normal_combined = tf.concat([mean_moving_normal,variance_shrinking_normal],0)
9 tf.summary.histogram("normal/bimodal",normal_combined)
10 #add gamma distribution
11 gamma = tf.random_gamma(shape=[1000],alpha=k)
12 tf.summary.histogram('gamma',gamma)
13 poisson = tf.random_poisson(shape=[1000],lam=k)
14 tf.summary.histogram('poisson',poisson)
15 #add a uniform distribution
16 uniform = tf.random_uniform(shape=[1000],maxval=k*10)
17 tf.summary.histogram('uniform',uniform)
18 #finnally combine everything together
19
20 all_distributions = [mean_moving_normal,variance_shrinking_normal,gamma,poisson,
21                      uniform]
22 all_combined = tf.concat(all_distributions,0)
23 tf.summary.histogram('all_combined',all_combined)
24 summaries = tf.summary.merge_all()
25 sess = tf.Session()
26 writer = tf.summary.FileWriter('./histogram_example2')
27 N = 400
28 for step in range(N):
29     k_val = step/float(N)
30     summ = sess.run(summaries,feed_dict={k:k_val})
31     writer.add_summary(summ,global_step=step)

```



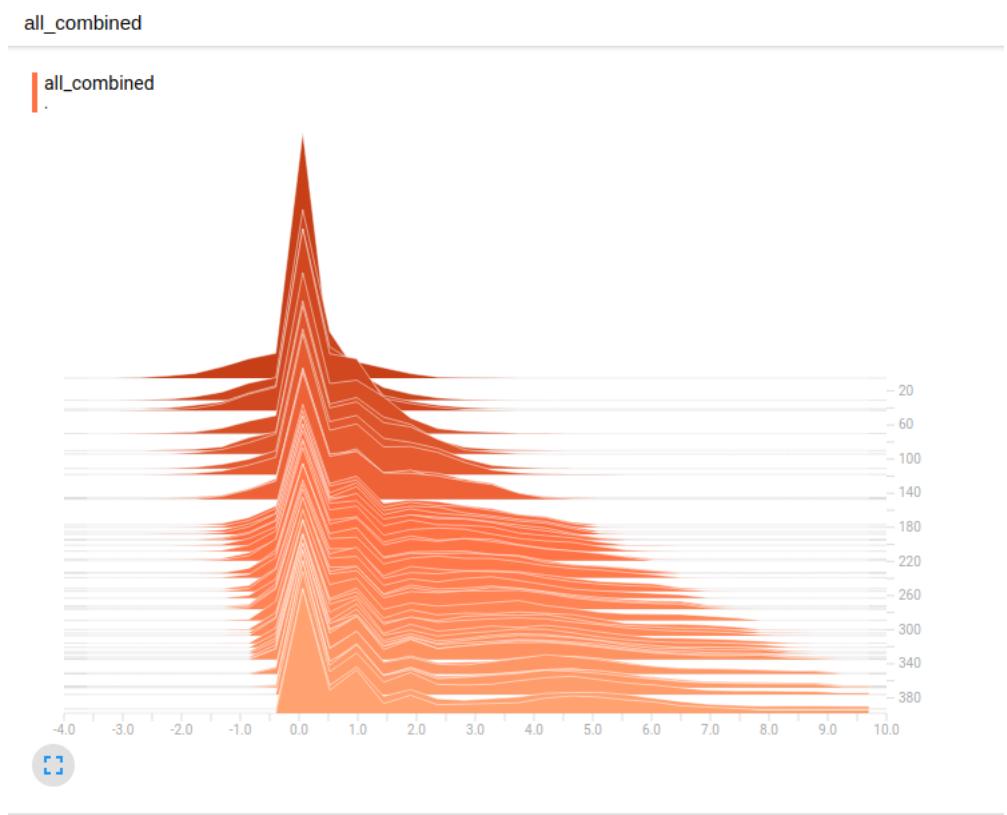


2.11.6 poisson 分布



poisson 分布定义在整数上，因此所有被生成的值都是整数，直方图压缩移动数据到浮点 bins，导致可视化在整数值上显示一点点突起。

2.11.7 结合所有的数据到一张图向上



```
1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3 import numpy as np
4 tf.set_random_seed(0)
5 np.random.seed(0)
6 x = np.linspace(-1, 1, 100).reshape(-1, 1)
7 noise = np.random.normal(0, 0.1, size=x.shape)
8 y = np.power(x, 2) + noise
9 def gendata():
10     t = np.linspace(-1, 1, 100).reshape(-1, 1)
11 def save():
12     print('This is save')
13     tf_x = tf.placeholder(tf.float32, x.shape)
14     tf_y = tf.placeholder(tf.float32, y.shape)
15     l = tf.layers.dense(tf_x, 10, tf.nn.relu)
16     o = tf.layers.dense(l, 1)
```

```

17     loss = tf.losses.mean_squared_error(tf_y,o)
18     train_op = tf.train.GradientDescentOptimizer(learning_rate=0.5).minimize(
19                                     loss)
20     sess = tf.Session()
21     sess.run(tf.global_variables_initializer())
22     saver = tf.train.Saver()
23     for step in range(100):
24         sess.run(train_op,{tf_x:x,tf_y:y})
25     saver.save(sess,'params',write_meta_graph=False)
26     pred,l = sess.run([o,loss],{tf_x:x,tf_y:y})
27     plt.figure(1,figsize=(10,5))
28     plt.subplot(121)
29     plt.scatter(x,y)
30     plt.plot(x,pred,'r-',lw=5)
31     plt.text(-1,1.2,'save loss=%f'%l,fontdict={'size':15,'color':'red'})
32 def reload():
33     print('This is reload')
34     tf_x = tf.placeholder(tf.float32,x.shape)
35     tf_y = tf.placeholder(tf.float32,y.shape)
36     l_ = tf.layers.dense(tf_x,10,tf.nn.relu)
37     o_ = tf.layers.dense(l_,1)
38     loss_ = tf.losses.mean_squared_error(tf_y,o_)
39     sess = tf.Session()
40     saver = tf.train.Saver()
41     saver.restore(sess,'params')
42     pred,l = sess.run([o_,loss_],{tf_x:x,tf_y:y})
43     plt.subplot(122)
44     plt.scatter(x,y)
45     plt.plot(x,pred,'r-',lw=5)
46     plt.text(-1,1.2,'Reload Loss=%f'%l,fontdict={'size':15,'color':'red'})
47     plt.show()
48     save()
49     tf.reset_default_graph()
50     reload()

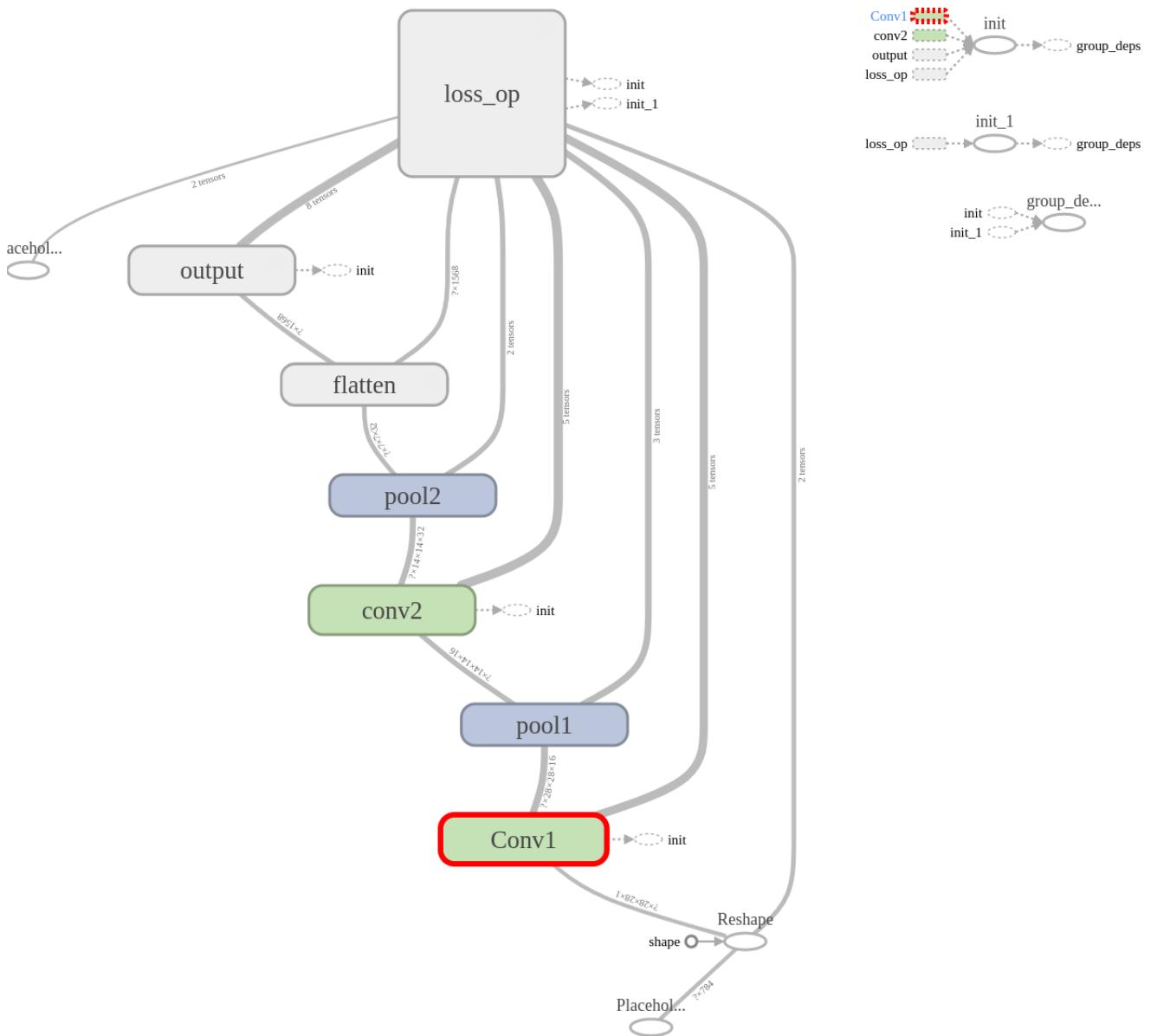
```

2.12 CNN 手写体数据识别

2.12.1 mnist 数据集

手写体数据训练集有 55000 张手写体数据图片。测试集有 10000 张图片。每张图片是大小为 32*32 的灰度图片。卷积神经网络结构：

- 第一层卷积层：卷积核 16 个，卷积核大小为 5×5 ,strides=1,padding 为 SAME，激活函数为 relu(输出大小为 $28 \times 28 \times 16$)。
- 第一层池化层：池化层大小为 2,strides 为 2($14 \times 14 \times 16$)。第二层卷积层：卷积核 32，大小为 5×5 ,strides=1,padding 为 SAME，激活函数为 relu。 $(14 \times 14 \times 32)$
- 第二层池化层：池化层大小为 2,strides 为 2($7 \times 7 \times 32$)。
- flatten:1568。



```

1 import tensorflow as tf
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from tensorflow.examples.tutorials.mnist import input_data
5
6 tf.set_random_seed(0)
7 np.random.seed(0)
8

```

```

9 BATCH_SIZE = 50
10 LR = 0.001
11 mnist = input_data.read_data_sets('/home/hpc/文档/mnist_tutorial/mnist', one_hot
12 = True)
13 test_x = mnist.test.images[:2000]
14 test_y = mnist.test.labels[:2000]
15
16 tf_x = tf.placeholder(tf.float32, [None, 28*28])
17 images = tf.reshape(tf_x, [-1, 28, 28, 1])
18 tf_y = tf.placeholder(tf.int32, [None, 10])
19 with tf.variable_scope('Conv1'):
20     conv1 = tf.layers.conv2d(
21         inputs = images,
22         filters = 16,
23         kernel_size = 5,
24         strides = 1,
25         padding = 'same',
26         activation = tf.nn.relu
27     )
28     tf.summary.histogram('conv1', conv1)
29 with tf.variable_scope('pool1'):
30     pool1 = tf.layers.max_pooling2d(
31         conv1,
32         pool_size=2,
33         strides =2
34     )
35     tf.summary.histogram('max_pool1', pool1)
36 with tf.variable_scope('conv2'):
37     conv2 = tf.layers.conv2d(pool1, 32, 5, 1, 'SAME', activation=tf.nn.relu)
38     tf.summary.histogram('conv2', conv2)
39 with tf.variable_scope('pool2'):
40     pool2 = tf.layers.max_pooling2d(conv2, 2, 2)
41     tf.summary.histogram('max_pool', pool2)
42 with tf.variable_scope('flatten'):
43     flat = tf.reshape(pool2, [-1, 7*7*32])
44 with tf.variable_scope('output'):
45     output = tf.layers.dense(flat, 10)
46 with tf.variable_scope('loss_op'):
47     loss = tf.losses.softmax_cross_entropy(onehot_labels=tf_y, logits=output)
48     train_op = tf.train.AdamOptimizer(LR).minimize(loss)
49     accuracy = tf.metrics.accuracy(labels = tf.argmax(tf_y, axis=1), predictions =
50                                     tf.argmax(output, axis=1),)[1]
51     tf.summary.scalar('loss', loss)

```

```

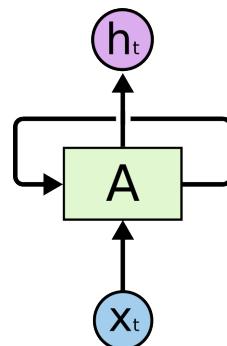
50     tf.summary.scalar('accuracy', accuracy)
51 sess = tf.Session()
52 merge_op = tf.summary.merge_all()
53 init_op = tf.group(tf.global_variables_initializer(), tf.
54                     local_variables_initializer())
55 sess.run(init_op)
56 writer = tf.summary.FileWriter('./log', sess.graph)
57 for step in range(600):
58     b_x, b_y = mnist.train.next_batch(BATCH_SIZE)
59     _, loss_, result = sess.run([train_op, loss, merge_op], {tf_x:b_x, tf_y:b_y})
60     writer.add_summary(result, step)
61     if step%50 == 0:
62         accuracy_, flat_representation = sess.run([accuracy, flat], {tf_x:test_x,
63                                         tf_y:test_y})
64         print('Step:', step, '| train loss: %.4f | loss_: %.2f | test accuracy: %.2f' %
65               accuracy_)
66 test_output = sess.run(output, {tf_x:test_x[:10]})
```

2.13 RNN

人不能抓住每一秒的思考，当你读这篇文章的时候，你能基于你之前的对单词的理解明白文章的每一个单词的意思，你思考的时候不需要丢掉所有的东西，你的思想有持续性。

传统的神经网络很难做到这点，这也是传统神经网络的主要缺点。例如你想分类电影中的不同时间点的事件，传统神经网络用不清楚如何用之前的事情了解新的事件。

RNN 通过循环处理这个问题，允许信息保留。



上面的图表示一个 RNN 单元， A 得到输入 x_t 和输出 h_t ， A 允许信息被循环从一步到下一步，一个循环神经网络可以看成是多个相同单元的复制。铺开 RNN 可以得到这个链式结构揭示了循环神经网络和序列或者列表密切相关，它适用于这种数据。

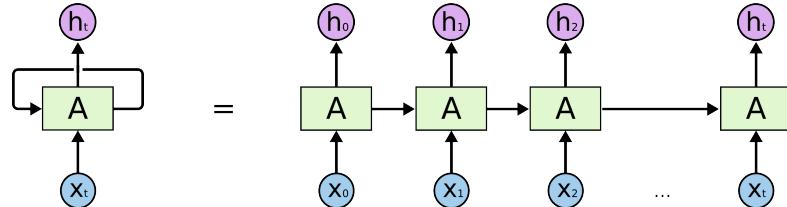
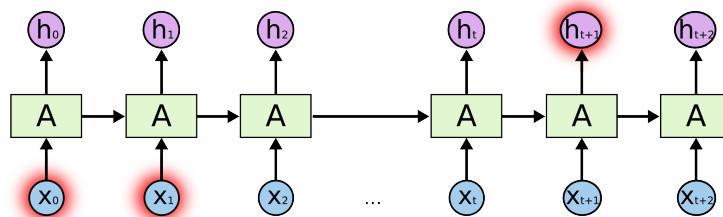


图 2.4: unrolled RNN

2.13.1 The Problem Long-Term Dependencies

语言模型中常用先前的一个词预测下一个词，如果我们尝试预测“the clouds are in the sky”我们不需要很多上下文信息 RNN 通过之前的信息就能学到。但是我们尝试预测这样一个句子“I grew up in France... I speak fluent France”，之前的信息暗示下一个单词可能是语言的名字，如果我们想去缩小语言的范围，我们需要上下文France，可相关信息和这个需要点的间隔很大。理论上 RNN 有能力处理“long-term dependencies”，人能小心的挑



选参数解决这个烦人的问题，然而不幸的是 RNN 似乎不能做到，原因由Hochreiter (1991) [German] and Bengio, et al. (1994)提出。

2.13.2 LSTM 网络

Long Short Term Memory networks 通常简称为 LSTMs 是一个特殊的 RNN，能学习 learning long-term dependencies，他被Hochreiter Schmidhuber (1997)引入，然后被提炼，在大型文体处理上效果很好因而被广泛的使用。

LSTMs 明确的设计去解决 long-term dependency problem。

所有的循环神经网络都有重复的链式形式。在标准的 RNNs，重复的模块有一个非常简单的结构，像 tanh Layer。LSTMs 也有这样类似的结构，但是 congruent 模块有点不同，有一个神经网络层有四个相互作用部分，在上面的图上，每一根线上携带的都是一个向量，从一个输出节点到其他输入，粉色圆圈代表按点操作，黄色盒子是学习好的神经网络层，线融合表示串联，copy 表示将一条线复制一份。

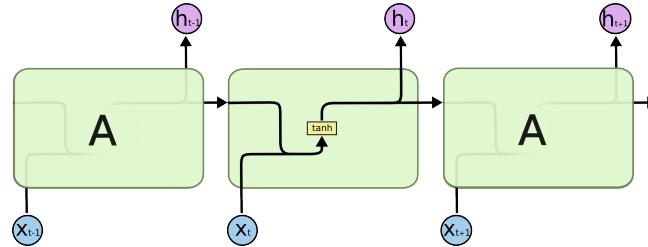


图 2.5: The repeating module in a standard RNN contains a single layer

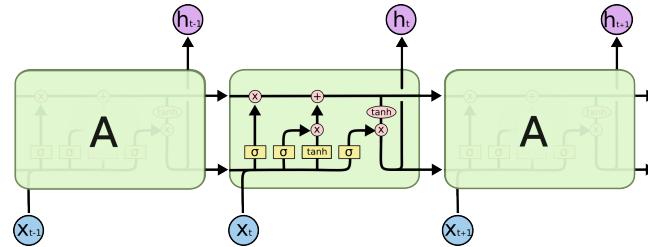


图 2.6: The repeating module in an LSTM contains four interacting layers

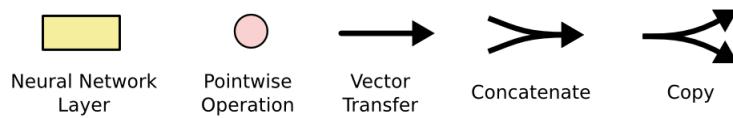
2.13.3 LSTMs 想法的核心

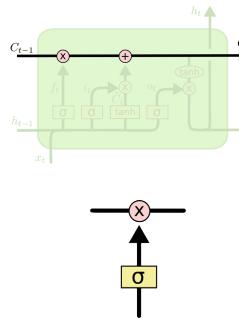
LSTMs 的核心是图像顶部的水平流过的 cell state, cell state 像一个传送带, 它笔直的沿着整条链跑, 和一些次要的线性交互, 很容易实现信息不改变的流动。LSTM 能删除或者增加信息到 cell state, 被控制的结构称为门。门是一种让信息通过的手段, 由一个 sigmoid 神经网络层和 pointwise 惩罚操作组成。sigmod Layer 输出 0 到 1 之间的数, 描述多少组件应该被通过, 0 表示不允许通过 1 表示让一切通过, LSTMs 有三个门, 保护和控制 cell state。

2.13.4 一步步的设置

第一步是 LSTMs 决定什么信息应该被传送, 这个决定每一个称为忘记门的 sigmoid layer 组成, 通过 h_{t-1} 和 x_t 输出 0 到 1 之间的数给当前的 $C_{t-1,1}$ 表示完全保持, 0 表示丢弃。

对于上面的语言模型, cell state 也许包含 the gender of the present subjects, 以至于正确的带名字能被使用, 当我们看一个新的 subject, 我们想图忘记 the gender of the old subject。下一步是决定什么新的信息将被存储在 cell state 中, 这分为两部分





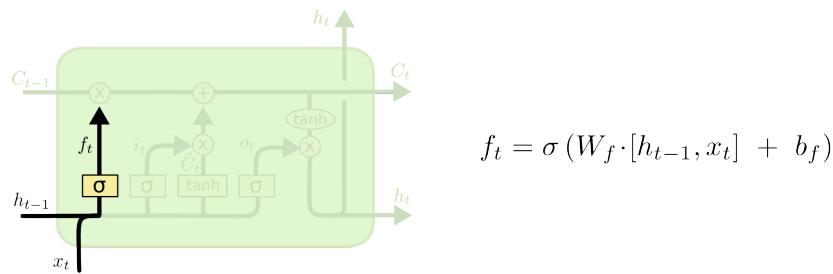
1. Sigmoid layer 调用 input gate layer 决定更新哪个值。
2. tanh layer 创建一个可能被添加到 state 新的候选向量。 \tilde{C}_t

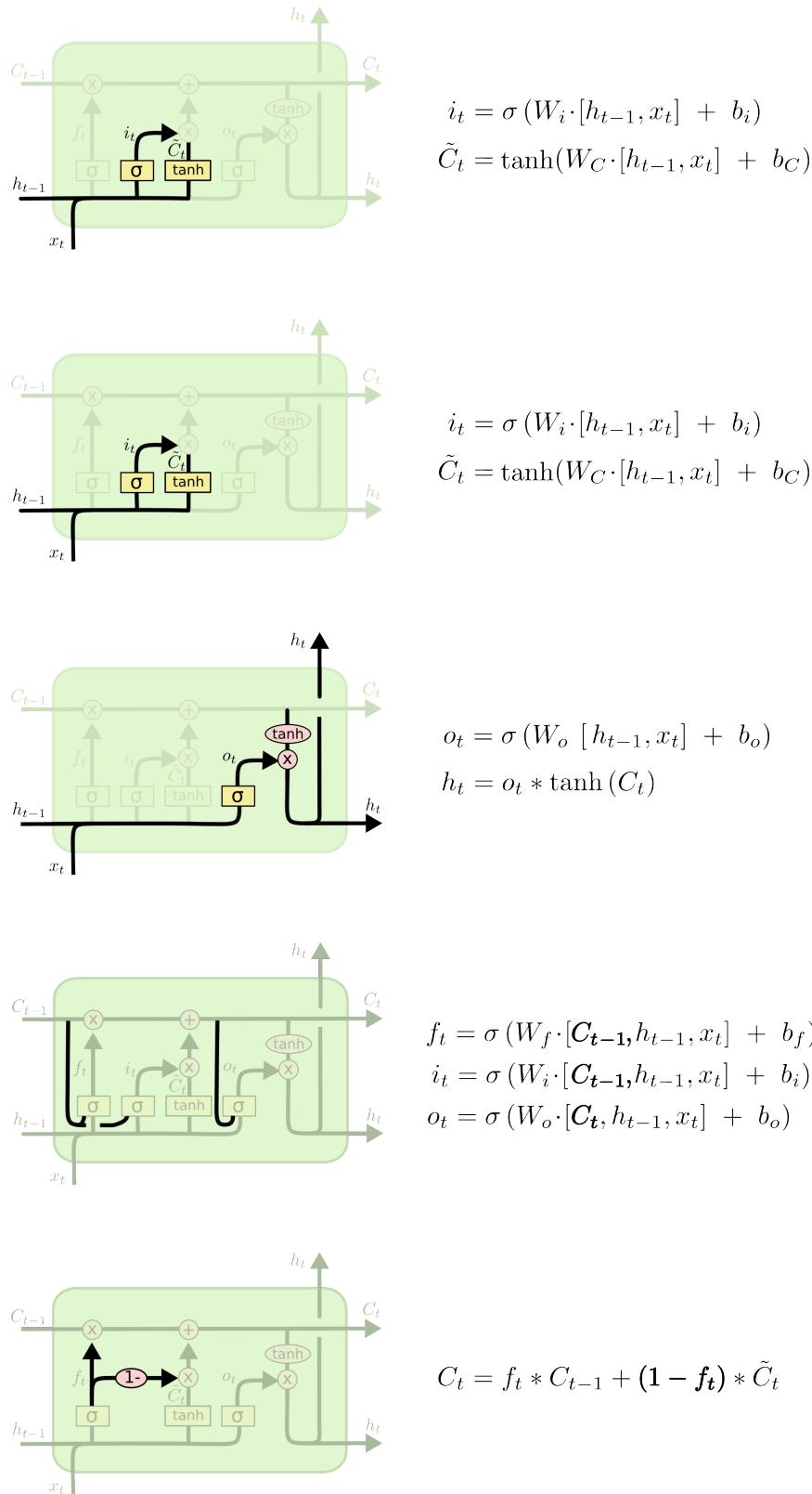
下一步我们结合两个不走创建一个更新状态。, 在我们的语言模型例子中, 我们想要增加 gender of the new subject 到 cell state 取代我们将要忘记的数据 现在更新老的 cell state C_{t-1} 到新的 cell state C_t , 我们用老的 c_{t-1} 乘上 f_t 忘记我们之前决定忘记的事, 然后我们增加 $i_t * \tilde{C}_t$. 这是新的候选值, 表示我们更新每个状态值的规模。在例子中的语言模型, 我们删掉了一个老的 subject's gender 增加新的信息。最后我们需要决定我们输出什么, 输出取决于我们的 cell state, 但是将被过滤, 所限我们运行 sigmoid layer 决定我们将输出那一部分。然后我们放通过 tanh 将 cell state 映射到-1,1, 然后乘上 sigmoid 门的输出, 以至于我们仅仅输出我们决定输出的部分。

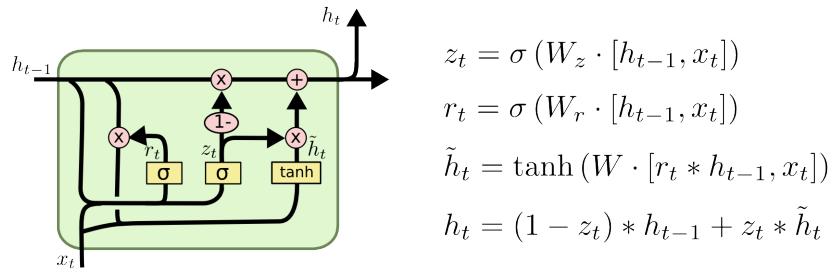
对于语言模型的例子, 因为它仅仅看 subject, 它也许想输出关于动词的信息, 例子中的下一个, 例如, 它也许输出是否 subject 是单数或者复数, 以至于从一个动词应该能知道接下来应该是动词的什么形式。

2.13.5 LSTM 的多种变体

[Gers Schmidhuber \(2000\)](#), 它增加了 peephole connections, 这一位置我们让 gate layer 通过 cell state 上面的图增加了 peepholes 到所有的门, 但是一些论文给出一些 peepholes 和 not others. 另一个变体用两个 forget 和输入门。而不是分别决定忘记或者添加信息, 我们一起决定, 我们需要输入一些值是忘记, 我们仅仅忘记老的值输入新值到 state 一个更







引人注目的变体是 Gate Recurrent Unit 或者称为 (GRU), 由Cho, et al. (2014)引入, 它结合忘记和输入门为一个单独的更新们, 它也融合 cell state 和 hidden state 做了些改变, 这结果模型比标准的 LSTM 模型简单, 现在也越来越流行。这些仅仅是非常流行的 LSTM 变体, 有一些其它的像Yao, et al. (2015)的 Depth Gated RNNs, 用完全不同的方法处理 long-term dependencies, 像Koutnik, et al. (2014)的 Clockwork RNNs。

那个算法是最好的? 他们的差别大吗? Greff, et al. (2015) 做了一些比较了一些流行的变体, 发现他们基本相同。Jozefowicz, et al. (2015)比较了超过 1 万中架构, 找到了一些在确定问题上比 LSTMs 好的架构。

2.13.6 向量字表示

Vector Representation of Words

通常图像或音频系统处理的是由图片中所有单个原始像素点强度值或者音频中功率谱密度的强度值, 把它们编码成丰富、高维度的向量数据集。对于物体或语音识别这一类的任务, 我们所需的全部信息已经都存储在原始数据中 (显然人类本身就是依赖原始数据进行日常的物体或语音识别的)。然后, 自然语言处理系统通常将词汇作为离散的单一符号, 例如”cat”一词或可表示为 Id537, 而”dog”一词或可表示为 Id143。这些符号编码毫无规律, 无法提供不同词汇之间可能存在的关联信息。换句话说, 在处理关于”dogs”一词的信息时, 模型将无法利用已知的关于”cats”的信息 (例如, 它们都是动物, 有四条腿, 可作为宠物等等)。可见, 将词汇表达为上述的独立离散符号将进一步导致数据稀疏, 使我们在训练统计模型时不得不寻求更多的数据。而词汇的向量表示将克服上述的难题。向量空间模型 (VSMs) 将词汇表达 (嵌套) 于一个连续的向量空间中, 语义近似的词汇被映射为相邻的数据点。向量空间模型在自然语言处理领域中有着漫长且丰富的历史, 不过几乎所有利用这一模型的方法都依赖于 分布式假设, 其核心思想为出现于上下文情景中的词汇都有相类似的语义。采用这一假设的研究方法大致分为以下两类: 基于计数的方法 (e.g. 潜在语义分析), 和 预测方法 (e.g. 神经概率化语言模型)。

其中它们的区别在如下论文中又详细阐述 Baroni :et al, 不过简而言之: 基于计数的方法计算某词汇与其邻近词汇在一个大型语料库中共同出现的频率及其它统计量, 然后将这些统计量映射到一个小型且稠密的向量中。预测方法则试图直接从某词汇的邻近词汇对其

进行预测，在此过程中利用已经学习到的小型且稠密的嵌套向量。

Word2vec 是一种可以进行高效率词嵌套学习的预测模型。其两种变体分别为：连续词袋模型（CBOW）及 Skip-Gram 模型。从算法角度看，这两种方法非常相似，其区别为 CBOW 根据源词上下文词汇（'the cat sits on the'）来预测目标词汇（例如，'mat'），而 Skip-Gram 模型做法相反，它通过目标词汇来预测源词汇。Skip-Gram 模型采取 CBOW 的逆过程的动机在于：CBOW 算法对于很多分布式信息进行了平滑处理（例如将一整段上下文信息视为一个单一观察量）。很多情况下，对于小型的数据集，这一处理是有帮助的。相形之下，Skip-Gram 模型将每个“上下文-目标词汇”的组合视为一个新观察量，这种做法在大型数据集中会更为有效。本教程余下部分将着重讲解 Skip-Gram 模型。

处理噪声的对比训练

神经概率化语言模型通常使用极大似然法 (ML) 进行训练，其中通过 softmax function 来最大化当提供前一个单词 h (代表"history")，后一个单词的概率 w_t (目标词概率)

$$P(w_t|h) = \text{softmax}(\text{score}(w_t, h)) = \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\}}$$

当 $\text{score}(w_t, h)$ 计算了文字 w_t 和 上下文 h 的相容性（通常使用向量积）。我们使用对数似然函数来训练训练集的最大值，比如通过：

$$J_{ML} = \log P(w_t|h) = \text{score}(w_t, h) - \log(\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\})$$

这里提出了一个解决语言概率模型的合适的通用方法。然而这个方法实际执行起来开销非常大，因为我们需要去计算并正则化当前上下文环境 h 中所有其它 V 单词 w' 的概率得分，在每一步训练迭代中。

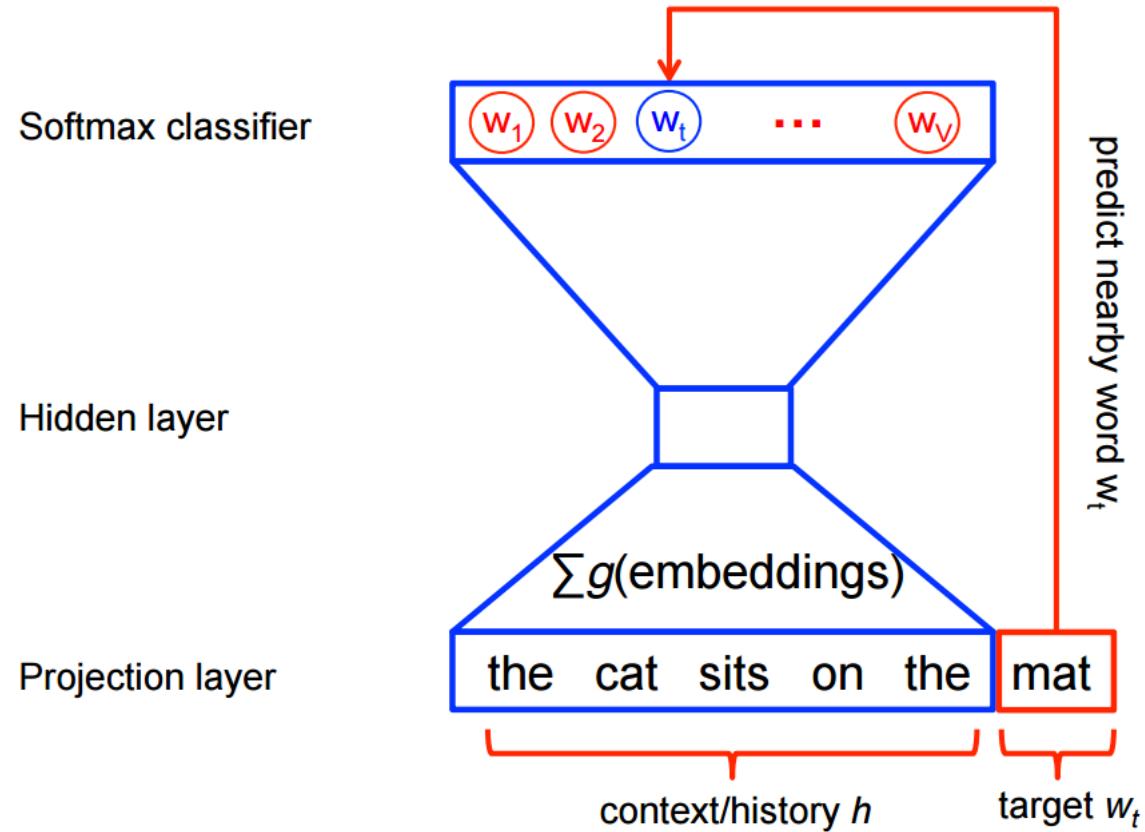


图 2.7: CBOW 方法

从另一个角度来说，当使用 word2vec 模型时，我们并不需要对概率模型中的所有特征进行学习。而 CBOW 模型和 Skip-Gram 模型为了避免这种情况发生，使用一个二分类器（逻辑回归）在同一个上下文环境里从 k 虚构的（噪声）单词 \hat{w} 区分真正的目标单词 w_t ，下面详细参数 CBOW 模型，对于 Skip-Gram 模型只要简单的反向操作即可。

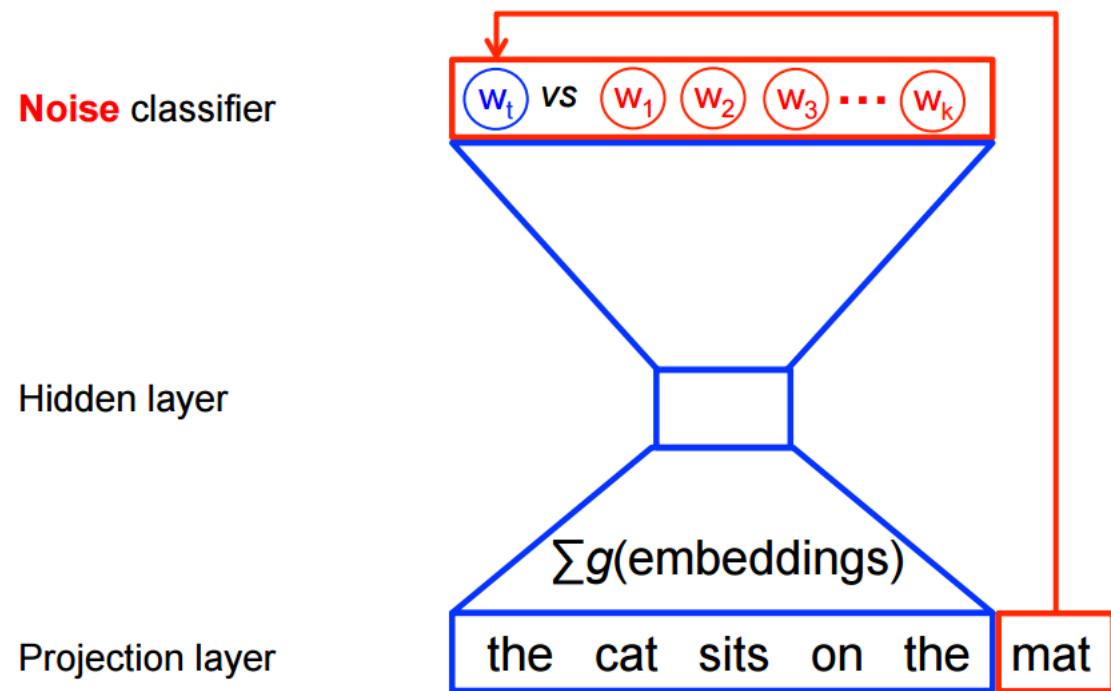


图 2.8: Skip-Gram

从数学的角度来说，我们的目标是对每个样本最大化:

$$J_{NEG} = \log Q_\theta(D = 1|w_t, h) + k \underset{\hat{w} \sim P_{noise}}{E} [\log Q_\theta(D = 0|\hat{w}, h)]$$

其中 $Q_\theta(D = 1|w, h)$ 代表的是当前上下文 h ，根据所学得嵌套向量 θ 目标单词 w 使用二分类逻辑回归计算得出的概率。在实践中，我们通过在噪声分布中绘制比对文字来获得近似的期望值（通过计算蒙特卡洛平均值）。

当真实地目标单词被分配到较高的概率，同时噪声单词的概率很低时，目标函数也就达到最大值了。从技术层面来说，这种方法叫做**负抽样**，而且使用这个损失函数在数学层面上也有很好的解释：这个更新过程也近似于 softmax 函数的更新。这在计算上将会有很大的优势，因为当计算这个损失函数时，只是有我们挑选出来的 k 个 噪声单词，而没有使用整个语料库 V 。这使得训练变得非常快。我们实际上使用了与**noise-contrastive estimation (NCE)**介绍的非常相似的方法，这在 TensorFlow 中已经封装了一个很便捷的函数 `tf.nn.nce_loss()`。

Skip-gram 模型

下面来看一下这个数据集

the quick brown fox jumped over the lazy dog

我们首先对一些单词以及它们的上下文环境建立一个数据集。我们可以以任何合理的方式定义‘上下文’，而通常上这个方式是根据文字的句法语境的（使用语法原理的方式处理当前目标单词可以看一下这篇文献 [Levy et al.](#)，比如说把目标单词左边的内容当做一个‘上下文’，或者以目标单词右边的内容，等等。现在我们把目标单词的左右单词视作一个上下文，使用大小为 1 的窗口，这样就得到这样一个由(上下文, 目标单词)组成的数据集：

([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...

前文提到 Skip-Gram 模型是把目标单词和上下文颠倒过来，所以在这个问题中，举个例子，就是用'quick' 来预测'the' 和'brown'，用'brown' 预测'quick' 和'fox'。因此这个数据集就变成由(输入, 输出)组成的：

(quick, the), (quick, brown), (brown, quick), (brown, fox), ...

目标函数通常是对整个数据集建立的，但是本问题中要对每一个样本（或者是一个 batch_size 很小的样本集，通常设置为 $16 \leq \text{batch_size} \leq 512$ ）在同一时间执行特别的操作，称之为[随机梯度下降 \(SGD\)](#)。我们来看一下训练过程中每一步的执行。

假设用 t 表示上面这个例子中 quick 来预测 the 的训练的单个循环。用 num_noise 定义从噪声分布中挑选出来的噪声（相反的）单词的个数，通常使用一元分布， $P(w)$ 。为了简单起见，我们就定 num_noise=1，用 sheep 选作噪声词。接下来就可以计算每一对观察值和噪声值的损失函数了，每一个执行步骤就可表示为：

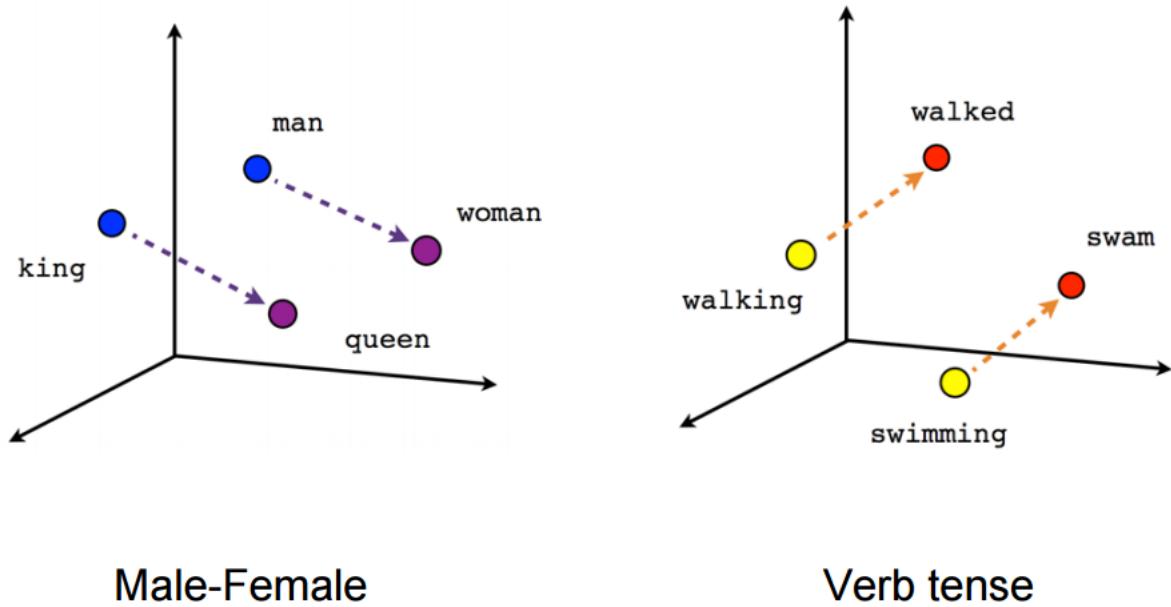
$$J_{NEG}^{(t)} = \log Q_\theta(D=1|the, quick) + \log(Q_\theta(D=0|sleep, quick))$$

整个计算过程的目标是通过更新嵌套参数 θ 来逼近目标函数（这个例子中就是使目标函数最大化）。为此我们要计算损失函数中嵌套参数 θ 的梯度，比如

$$\frac{\partial}{\partial} J_{NEG}$$

（幸好 TensorFlow 封装了工具函数可以简单调用！）。对于整个数据集，当梯度下降的过程中不断地更新参数，对应产生的效果就是不断地移动每个单词的嵌套向量，直到可以把真实单词和噪声单词很好得区分开。

我们可以把学习向量映射到 2 维中以便我们观察，其中用到的技术可以参考[t-SNE 降维技术](#)。当我们用可视化的方式来观察这些向量，就可以很明显的获取单词之间语义信息的关系，这实际上是非常有用的。当我们第一次发现这样的诱导向量空间中，展示了一些特定的语义关系，这是非常有趣的，比如文字中 male-female, gender 甚至还有 country-capital 的关系，如下方的图所示（也可以参考 [Mikolov et al., 2013](#) 论文中的例子）。



这也解释了为什么这些向量在传统的 NLP 问题中可作为特性使用，比如用在对一个演讲章节打个标签，或者对一个专有名词的识别（看看如下这个例子 [Collobert et al.](#) 或者 [Turian et al.](#)）。

不过现在让我们用它们来画漂亮的图表吧！

这里谈得都是嵌套，那么先来定义一个嵌套参数矩阵。我们用唯一的随机值来初始化这个大矩阵。

```
1 embeddings = tf.Variable(
2     tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

对噪声-比对的损失计算就使用一个逻辑回归模型。对此，我们需要对语料库中的每个单词定义一个权重值和偏差值。（也可称之为输出权重与之对应的输入嵌套值）。定义如下：

```
1 nce_weights = tf.Variable(
2     tf.truncated_normal([vocabulary_size, embedding_size],
3                         stddev=1.0 / math.sqrt(embedding_size)))
4 nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
```

我们有了这些参数之后，就可以定义 Skip-Gram 模型了。简单起见，假设我们已经把语料库中的文字整型化了，这样每个整型代表一个单词（细节请查看 `_basic.py`）。Skip-Gram 模型有两个输入。一个是一组用整型表示的上下文单词，另一个是目标单词。给这些输入建立占位符节点，之后就可以填入数据了。

```

1 train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
2 train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])

```

然后我们需要对批数据中的单词建立嵌套向量，TensorFlow 提供了方便的工具函数。

```

1 embed = tf.nn.embedding_lookup(embeddings, train_inputs)

```

好了，现在我们有了每个单词的嵌套向量，接下来就是使用噪声-比对的训练方式来预测目标单词。

```

1 loss = tf.reduce_mean(
2     tf.nn.nce_loss(nce_weights, nce_biases, embed, train_labels,
3                     num_sampled, vocabulary_size))

```

我们对损失函数建立了图形节点，然后我们需要计算相应梯度和更新参数的节点，比如说在这里我们会使用随机梯度下降法，TensorFlow 也已经封装好了该过程。

```

1 optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0).minimize(loss)

```

训练过程

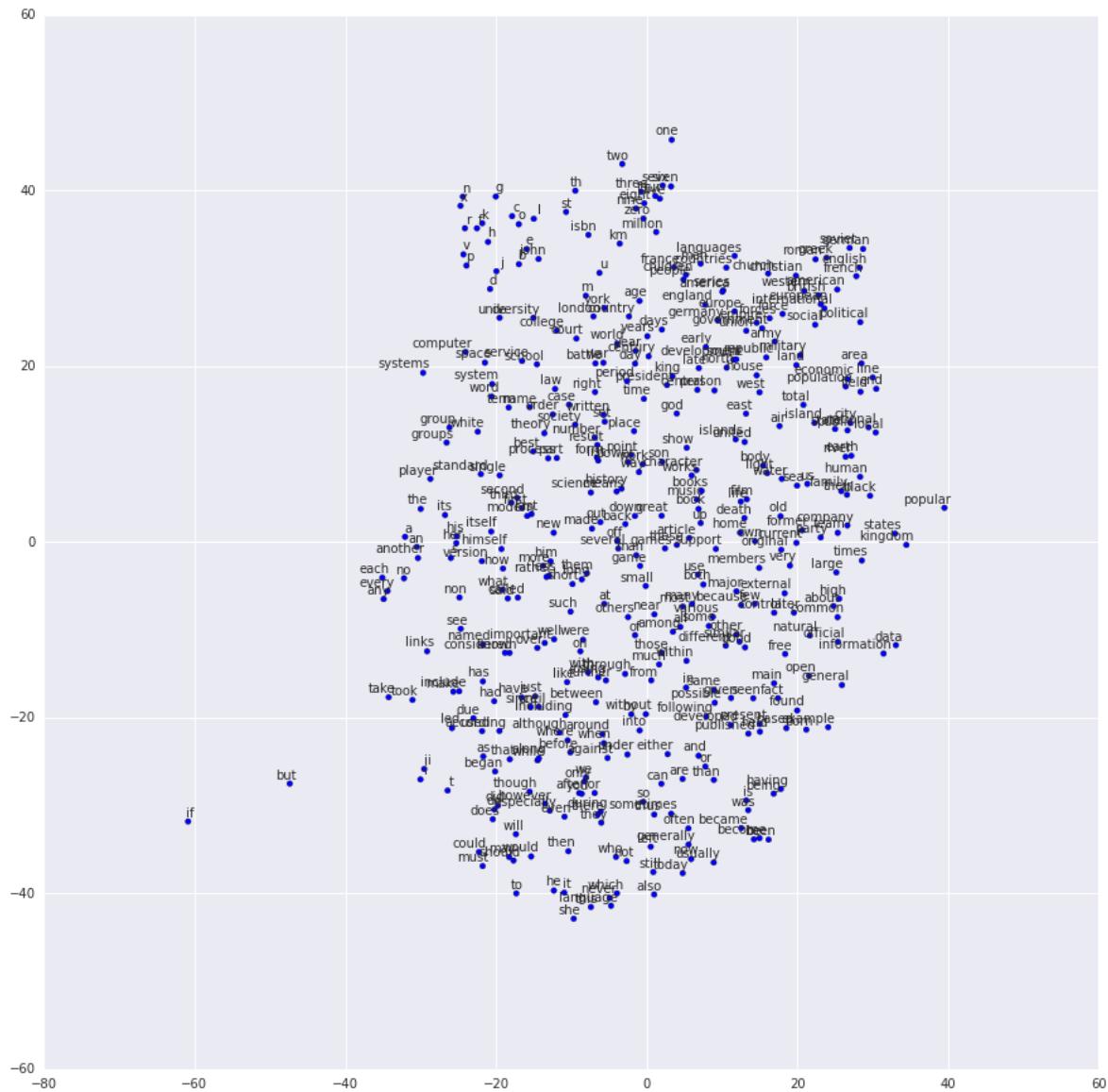
训练的过程很简单，只要在循环中使用 `feed_dict` 不断给占位符填充数据，同时调用 `session.run` 即可。

```

1 for inputs, labels in generate_batch(...):
2     feed_dict = {training_inputs: inputs, training_labels: labels}
3     _, cur_loss = session.run([optimizer, loss], feed_dict=feed_dict)

```

嵌套学习结果可视化



Et voilà! 与预期的一样，相似的单词被聚类在一起。对 word2vec 模型更复杂的实现需要用到 TensorFlow 一些更高级的特性，具体是实现可以参考[word2vec.py](#)

嵌套学习的评估：类比推理

词嵌套在 NLP 的预测问题中是非常有用且使用广泛地。如果要检测一个模型是否是可以成熟地区分词性或者区分专有名词的模型，最简单的办法就是直接检验它的预测词性、

语义关系的能力，比如让它解决形如 king is to queen as father is to ? 这样的问题。这种方法叫做类比推理，可参考 Mikolov and colleagues，数据集下载地址为:[questions-words.txt](#)。To see how we do this evaluation 如何执行这样的评估，可以看 `build_eval_graph()` 和 `eval()` 这两个函数在下面源码中的使用 [word2vec.py](#)

超参数的选择对该问题解决的准确性有巨大的影响。想要模型具有很好的表现，需要有一个巨大的训练数据集，同时仔细调整参数的选择并且使用例如二次抽样的一些技巧。不过这些问题已经超出了本教程的范围。

优化实现

以上简单的例子展示了 TensorFlow 的灵活性。比如说，我们可以很轻松得用现成的 `tf.nn.sampled_softmax_loss()` 来代替 `tf.nn.nce_loss()` 构成目标函数。如果你对损失函数想做新的尝试，你可以用 TensorFlow 手动编写新的目标函数的表达式，然后用控制器执行计算。这种灵活性的价值体现在，当我们探索一个机器学习模型时，我们可以很快地遍历这些尝试，从中选出最优。

一旦你有了一个满意的模型结构，或许它就可以使实现运行地更高效（在短时间内覆盖更多的数据）。比如说，在本教程中使用的简单代码，实际运行速度都不错，因为我们使用 Python 来读取和填装数据，而这些在 TensorFlow 后台只需执行非常少的工作。如果你发现你的模型在输入数据时存在严重的瓶颈，你可以根据自己的实际问题自行实现一个数据阅读器，参考 新的数据格式。对于 Skip-Gram 模型，我们已经完成了如下这个例子 [word2vec.py](#)。

如果 I/O 问题对你的模型已经不再是个问题，并且想进一步地优化性能，或许你可以自行编写 TensorFlow 操作单元，详见 添加一个新的操作。相应的，我们也提供了 Skip-Gram 模型的例子 [optimized.py](#)。请自行调节以上几个过程的标准，使模型在每个运行阶段有更好的性能。

2.13.7 RNN

此教程将展示如何在高难度的语言模型中训练循环神经网络。该问题的目标是获得一个能确定语句概率的概率模型。为了做到这一点，通过之前已经给出的词语来预测后面的词语。我们将使用 PTB(Penn Tree Bank) 数据集，这是一种常用来衡量模型的基准，同时它比较小而且训练起来相对快速。

语言模型是很多有趣难题的关键所在，比如语音识别，机器翻译，图像字幕等。它很有意思—可以参看 [here](#)。

本教程的目的是重现 [Zaremba et al., 2014](#) 的成果，他们在 PTB 数据集上得到了很棒的结果。

下载及准备数据

本教程需要的数据在 data/ 路径下，来源于 Tomas Mikolov 网站上的[PTB 数据集](#)

该数据集已经预先处理过并且包含了全部的 10000 个不同的词语，其中包括语句结束标记符，以及标记稀有词语的特殊符号 (<unk>)。我们在 reader.py 中转换所有的词语，让他们各自有唯一的整型标识符，便于神经网络处理。

LSTM

模型的核心由一个 LSTM 单元组成，其可以在某时刻处理一个词语，以及计算语句可能的延续性的概率。网络的存储状态由一个零矢量初始化并在读取每一个词语后更新。而且，由于计算上的原因，我们将以 batch_size 为最小批量来处理数据。

基础的伪代码就像下面这样：

```

1 lstm = rnn_cell.BasicLSTMCell(lstm_size)
2 state = tf.zeros([batch_size, lstm.state_size])
3
4 loss = 0.0
5 for current_batch_of_words in words_in_dataset:
6     output, state = lstm(current_batch_of_words, state)
7
8     logits = tf.matmul(output, softmax_w) + softmax_b
9     probabilities = tf.nn.softmax(logits)
10    loss += loss_function(probabilities, target_words)
```

截断反向传播

为使学习过程易于处理，通常的做法是将反向传播的梯度在（按时间）展开的步骤上照一个固定长度 (num_steps) 截断。通过在一次迭代中的每个时刻上提供长度为 num_steps 的输入和每次迭代完成之后反向传导，这会很容易实现。

一个简化版的用于计算图创建的截断反向传播代码：

```

1 words = tf.placeholder(tf.int32, [batch_size, num_steps])
2
3 lstm = rnn_cell.BasicLSTMCell(lstm_size)
4 initial_state = state = tf.zeros([batch_size, lstm.state_size])
5
6 for i in range(len(num_steps)):
7     output, state = lstm(words[:, i], state)
8
9     # ...
```

```
11 final_state = state
```

下面展现如何实现迭代整个数据集：

```
1 numpy_state = initial_state.eval()
2 total_loss = 0.0
3 for current_batch_of_words in words_in_dataset:
4     numpy_state, current_loss = session.run([final_state, loss],
5         feed_dict={initial_state: numpy_state, words: current_batch_of_words})
6     total_loss += current_loss
```

输入

在输入 LSTM 前，词语 ID 被嵌入到了一个密集的表示中（查看 矢量表示教程）。这种方式允许模型高效地表示词语，也便于写代码：

```
1 # embedding_matrix 张量的形状是: [vocabulary_size, embedding_size]
2 word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

嵌入的矩阵会被随机地初始化，模型会学会通过数据分辨不同词语的意思。

损失函数

我们想使目标词语的平均负对数概率最小 $loss = -\frac{1}{N} \sum_{i=1}^N \ln p_{target_i}$ 实现起来并非很难，而且函数 sequence_loss_by_example 已经有了，可以直接使用。

论文中的典型衡量标准是每个词语的平均困惑度（perplexity），计算式为

$$e^{-\frac{1}{N} \sum_{i=1}^N \ln p_{target_i}} = e^{loss}$$

同时我们会观察训练过程中的困惑度值（perplexity）

多个 LSTM 层堆叠

要想给模型更强的表达能力，可以添加多层 LSTM 来处理数据。第一层的输出作为第二层的输入，以此类推。

类 MultiRNNCell 可以无缝的将其实现：

```
1 lstm = rnn_cell.BasicLSTMCell(lstm_size)
2 stacked_lstm = rnn_cell.MultiRNNCell([lstm] * number_of_layers)
3
4 initial_state = state = stacked_lstm.zero_state(batch_size, tf.float32)
5 for i in range(len(num_steps)):
6     # 每次处理一批词语后更新状态值。
7     output, state = stacked_lstm(words[:, i], state)
```

```
8      # 其余的代码.  
9  
10     # ...  
11  
12 final_state = state
```

编译并运行代码

首先需要构建库，在 CPU 上编译：

```
1 bazel build -c opt tensorflow/models/rnn/ptb:ptb_word_lm
```

如果你有一个强大的 GPU，可以运行

```
1 bazel build -c opt --config=cuda tensorflow/models/rnn/ptb:ptb_word_lm
```

运行模型：

```
1 bazel-bin/tensorflow/models/rnn/ptb/ptb_word_lm \  
2   --data_path=/tmp/simple-examples/data/ --alsologtosterr --model small
```

教程代码中有 3 个支持的模型配置参数：“small”，“medium” 和“large”。它们指的是 LSTM 的大小，以及用于训练的超参数集。

模型越大，得到的结果应该更好。在测试集中 small 模型应该可以达到低于 120 的困惑度（perplexity），large 模型则是低于 80，但它可能花费数小时来训练。

Chapter 3

Tensorflow 进阶

3.1 模型存储和加载

- 生成 checkpoint 文件, 扩展名一般为.ckpt, 通过在 tf.train.Saver 对象上调用 Saver.saver() 生成。它包含权重和其他程序中定义的变量, 不包含 图的结构。如果需要在另一个程序中使用, 需要重建图形结构, 并告诉 Tensorflow 如何处理这些权重。
- 生成 (graph proto file), 这是一个二进制文件, 扩展名一般是.pb, 用 tf.train.write_graph() 保存每, 只包含图形结构, 不包含权重, 然后使用 tf.import_graph_def() 加载 图形。

3.2 用 GPU

在 Tensorflow 中 CPU, GPU 用字符串表示

- "cpu:0": 机器上的 CPU
- "gpu:0": 机器上的 GPU
- "gpu:1": 机器上的第二块 GPU

如果 TensorFLow 操作有 GPU 和 CPU 实现, GPU 将被优先指定, 例如 matmul 有 CPU 和 GPU 内核, 在系统上有 cpu:0 和 gpu:0,gpu:0 将优先运行 matmul。布置采集设备

找到你的操作和 tensor 上的设备, 创建一个会话 log_device_placement 配置设置为 True

```

1 import tensorflow as tf
2 a = tf.reshape(tf.linspace(-1.,1.,12),(3,4))
3 b = tf.reshape(tf.sin(a),(4,3))
4 c = tf.matmul(a,b)
5 with tf.Session() as sess:
6     print(sess.run(c))

```

输出参数:

```

[[ 0.87280041  0.44710392  0.00666773]
 [ 0.43973413  0.44710392  0.4397341 ]
 [ 0.00666779  0.44710392  0.87280059]]

```

3.2.1 手工配置设备

如果你想将你的操作运行在指定的设备中而不由 tensorflow 是自动为你选择, 你可以用 tf.device 创建一个设备, 左右的操作将在同一个设备上指定。

```

1 import tensorflow as tf
2 with tf.device('/cpu:0'):
3     a = tf.constant([1.,2.,3.,4.,5.,6.],shape = (2,3),name = 'a')
4     b = tf.reshape(a,shape=(3,2))
5     c = tf.matmul(a,b)
6     with tf.Session(config = tf.ConfigProto(log_device_placement=True)) as sess:
7         print(sess.run(c))

```

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: TITAN Xp, pci bus id: 0000:06:00.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: TITAN Xp, pci bus id: 0000:05:00.0
Reshape: (Reshape): /job:localhost/replica:0/task:0/cpu:0
MatMul: (MatMul): /job:localhost/replica:0/task:0/cpu:0
Reshape/shape: (Const): /job:localhost/replica:0/task:0/cpu:0
a: (Const): /job:localhost/replica:0/task:0/cpu:0
[[ 22.  28.]
 [ 49.  64.]]
```

正如你看到的 a,b 被复制到 cpu:0, 因为设备没有明确指定, Tensorflow 将选择操作和可用的设备 (gpu:0)

3.2.2 允许 GPU 的内存增长

默认情况下 Tensorflow 将映射所有的 CPUs 的显存到进程上, 用相对精确的 GPU 内存资源减少内存的碎片化会更高效。通常有些程序希望分贝可用内存的一部分, 或者增加内存的需要两。在会话中 tensorflow 提供了两个参数 控制它。第一个参数是 allow_growth 选项, 根据运行情况分配 GPU 内存: 它开始分配很少的内存, 当 Session 开始运行 需要更多 GPU 内存是, 我们同感 Tensorflow 程序扩展 GPU 的内存区域。注意我们不释放内存, 因此这可能导致更多的内存碎片。为了开启这个选项, 可以通过下面的设置

```
1 config = tf.ConfigProto()
2 config.gpu_option.allow_growth = True
3 sess = tf.Session(config=config, ...)
```

第二种方法是 per_process_gpu_memory_fraction 选项, 决定 GPU 总体内存中多少应给被分配, 例如你可以告诉 Tensorflow 分配 40% 的 GPU 总体内存。

```
1 config = tf.ConfigProto()
2 config.gpu_option.per_process_gpu_memory_fraction = 0.4
3 sess = tf.Session(config = config)
```

如果你想限制 Tensorflow 程序的 GPU 使用量, 这个参数是很有用的。

在多 GPU 系统是使用 GPU

如果你的系统上有超过一个 GPU, 你的 GPU 的抵消的 ID 将被默认选中, 如果你想运行在不同的 GPU 上, 你需要指定 你想要执行运算的 GPU

```
1 import tensorflow as tf
2 with tf.device('/gpu2:0'):
3     a = tf.constant([1., 2., 3., 4., 5., 6.], shape=(2, 3), name='a')
4     b = tf.reshape(a, shape=(3, 2))
5     c = tf.matmul(a, b)
6     with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
7         print(sess.run(c))
```

如果你指定的设备不存在, 你将个到一个 InvalidArgumentError:

```
InvalidArgumentError (see above for traceback): Cannot assign a device for operation 'Reshape': Operation was explicitly assigned to /device:GPU:2 but available devices are [ /job:localhost/replica:0/task:0/cpu:0, /job:localhost/replica:0/task:0/gpu:0, /job:localhost/replica:0/task:0/cpu:1 ]. Make sure the device specification refers to a valid device.
[[Node: Reshape = Reshape[T=DT_FLOAT, Tshape=DT_INT32, _device="/device:GPU:2"](a, Reshape/shape)]]
```

如果你想 Tensorflow 在万一指定的设备不存在时自动选择一个存在的设备，你可以在创建会话时配置中设置 allow_soft_placement 为 True

```
1 with tf.device('/gpu:2'):
2     a = tf.constant([1., 2., 3., 4., 5., 6.], shape=[3, 2], name='a')
3     b = tf.constant([1., 2., 3., 4., 5., 6.], shape=[2, 3], name='b')
4     c = tf.matmul(a, b)
5 with tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
6                                     log_device_placement=True)) as sess:
7     print(sess.run(c))
```

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: TITAN Xp, pci bus id: 0000:06:00.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: TITAN Xp, pci bus id: 0000:05:00.0
Reshape: (Reshape): /job:localhost/replica:0/task:0/gpu:0
MatMul: (MatMul): /job:localhost/replica:0/task:0/gpu:0
Reshape/shape: (Const): /job:localhost/replica:0/task:0/gpu:0
a: (Const): /job:localhost/replica:0/task:0/gpu:0
[[ 22.  28.]
 [ 49.  64.]]
```

用多 GPU

如果你想在多张 GPU 上运行 Tensorflow，你可以在 multi-tower fashion 上构造你的模型，每个 tower 被指定到不同的 GPU 上。例如：

```
1 c = []
2 for d in ['/gpu:0', '/gpu:1']:
3     with tf.device(d):
4         a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
5         b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
6         c.append(tf.matmul(a, b))
7     with tf.device('/cpu:0'):
8         sum = tf.add_n(c)
9     # Creates a session with log_device_placement set to True.
10 sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
11                     log_device_placement=True))
12     # Runs the op.
13     print(sess.run(sum))
14 sess.close()
```



3.3 如何利用 Inception 的最后一层重新训练新的分类

现代的认知模型可能有上百万个参数可能需要花几周训练，Transfer 学习是通过完整的像 ImageNet 一样的模型通过已经存在的权重简化数周工作分类的技术。在这个例子中我们将创新训练最终层不修改其它层。详细信息你可以查看[这篇论文](#)。

尽管不完整的训练，但是对于一些应用却惊人的高效，可以在笔记本上训练 30 分钟，不要求 GPU。这个导航将显示给你如何在自己的图像运行示例脚本解释一些控制训练需要的脚本。

3.3.1 训练花

在开始训练前你需要设置图像教网络你想认识的新的类别。接下来的张杰解释如何准备你的图像，但是我们创建一个授权的归档的花的文件使得训练更轻松。为了得到花的图像，运行下面的代码：

```
1 cd ~  
2 curl -O http://download.tensorflow.org/example_images/flower_photos.tgz  
3 tar xzf flower_photos.tgz
```

当你有图像后，你从你的 TensorFlow 源文件目录构建重新训练器

```
1 bazel build --config opt tensorflow/examples/image_retraining:retrain
```

可以通过下面运行：

```
1  bazel build tensorflow/examples/image_retraining:retrain
```

如果你有一个机器支持 AVX 设备集 (最近几年的常用的 x86 CPUs) 你可以通过架构提高 building 运行速度

```
2  bazel build --config opt tensorflow/examples/image_retraining:retrain
```

训练器可以是这样:

```
3  bazel-bin/tensorflow/examples/image_retraining/retrain --image_dir ~/flower_photos
```

这个脚本载入先前 inception v3 模型, 删除顶层, 在新的 flower photos 训练新的模型。在原始 ImageNet 类中没有一种花完整网络被完整的训练过了, transfer 学习是低层已经被训练好 区别不修改任何不同对象。

3.3.2 瓶颈

训练花费 30 分钟甚至更长时间取决于你的机器的速度。第一个时期分析所有磁盘上的图像和计算他们的瓶颈, 瓶颈是一个信息对术语 我们经常在最后一层前一层, 倒数第二层已经训练区别输出要求分类的值, 这意味着这必须是有意义的, 因此对于分类器它必须包含足够的信息在一些小的值得集合中做选择, 这意味着我们的最终层训练可以在新的类中工作证明在 ImageNet 中 1000 类对于区别新的对象是有用的。因为每个图像在训练和计算花费时间瓶颈时被多次使用, 他的速度达到缓存起的瓶颈因此不能被重复计算。默认他们存储在/tmp/bottleneck 陌路, 如果你仍然会脚本他们将被重用, 因此你不是必须再次等待这部分。

3.3.3 训练

当瓶颈计算完成时, 实际顶层训练开始。你讲看到输出, 显示精度, 可用精度, 交叉熵。训练精度显示在当前训练批中多少被分类正确, 验证训练精度从图像数据集随机选中精度的值, 不同之处在于训练精度基于网络已经学习到的参数, 在训练中可能过拟合到为噪声。验证精确度用不在训练集中的数据性能测量精确度, 如果训练精确度很高, 测试精确度很低说明网络过拟合训练图像存储的部分参数没有用。交叉熵损失函数查看学习进程处理的增么样, 训练对象使得损失尽可能小, 因此你可以分辨出如果学习起作用, 忽略损失噪声损失保持下降的趋势。默认脚本运行 4000 步, 每一步从训练集中随机选择 10 张图像找到缓冲器的瓶颈, 输入数据仅最终层预测。预测然后比较实际 label 和真实值差距反向传递误差。当你继续的时候你应该看到精确度的提高。你应该能看到精确度在 90% 到 95% 之间, 通过提取值将随机的一次次训练, 这个数完全训练好模型后基于在给定测试集中正确标签的百分比。

3.3.4 用 TensorBoard 可视化

包含 TensorBoard 总结的脚本很容易理解，调试，优化。例如，你可以可视化图和统计，例如在训练中权重和精度变化：

```
4 tensorboard --logdir /tmp/retrain_logs
```

TensorBoard 运行后导航到 localhost:6006 查看 TensorBoard，脚本将默认采集 TensorBoard 总结到 /tmp/retrain_logs，你可以通过 summaries_dir 标志指定采集目录。

3.3.5 用重新训练的模型

脚本将用训练好的最后一层写出你在一个 Inception v3 版本到 /tmp/output_graph.pb 文件在 /tmp/output_labels.txt 包含标签，两个格式见 [C++ and Python image classification examples](#)，因此你可以立即开始新的模型。你去带最顶层，你将需要在脚本中指定新的名字，例如你用 label_image，用 output_layer=final_result。你可以用下面的代码重新训练图：

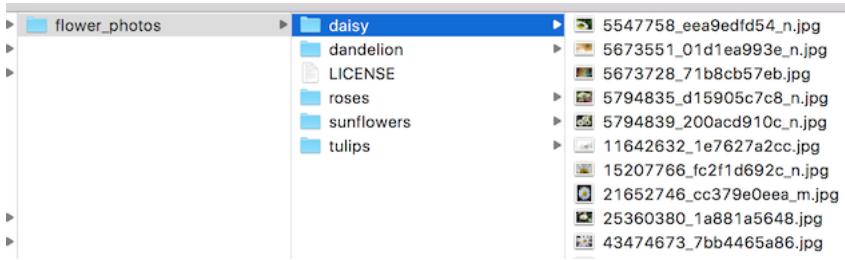
```
5 bazel build tensorflow/examples/image_retraining:label_image && \
6 bazel-bin/tensorflow/examples/image_retraining/label_image \
7 --graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
8 --image=$Home/flower_photos/daisy/21625746_cc379eea_m.jpg
```

你应该看到花的标签的列表在大说书情况下 daisy 在顶层（尽管重新训练的模型被可能会一点不同），你可以用在你的图片上--image 参数用 c++ 代码作为模板整合你自己的应用。如果你想在自己的 Python 程序中用你训练好的模型，上面的 [label_image script](#) 如果你发现默认的 Inception v3 模型对你的应用太大或者太慢，看看 [Other Model Architectures section](#)

3.3.6 在你自己的分类上训练

如果你已经成功的让脚本在花的例子上工作，你可以教他认识其他你想它认识的东西。理论上你需要设置一个子文件夹，命名分类，每个文件夹包含分类的图像。如果你传递子文件夹的根文件夹作为参数给--image_dir，脚本像上面训练花一样训练。

实际上它会花一点时间得到你想要的精度，下面是一些常见的问题。



3.3.7 创建一个训练图像集合

首先我们需要查看收集到的图像，常见的问题是训练过程中数据的输入。

为了训练能起作用，每个你想要识别的图像你必须至少手机 100 张图片，你收集到的图片越多，训练的精确度可能越好。例如你拍摄一些蓝色的房间，另一些是绿色的房间模型的预测最终基于背景颜色，没有对象特征被考虑。为了避免这种情况，拍摄不同颜色的，没有一些实际能看到的特征。如果你想了解更多这类问题你需要读[tank recognition](#) 如果你想考虑你用的分类。分隔大的数据集发现一些不同的物理形式为小的可以通过视觉区分的数据集，例如你可以用'vehicle' 可以用来替代'car'，'motobike' 和'truck'，考虑你有一个开放的世界还是封闭的世界将是很有价值的，在封闭的世界你唯一需要考虑的是识别已有的对象，例如一个植物识别的 app 你应该知道用户可能拍摄的花的图片，英雌你必须决定花的种类，相比之下一个巡逻机器人可能通过摄像头看到不同的事物。在这种情况下你想要分类器报告是否确认他看到的，这可能很难，但是你经常收集一些典型的和主体对象不相关的背景图像，，你可能会让它增加一些图片文件夹中未知的分类。检查确保你的图像被正确的标记也是很重要的。经常用生成的标签对于你的目的来说是不可靠的，例如你用 #daisy 命名一个叫 Daisy 的人。如果你想你的图像如果你了解你的图像，扫除任何错误将可能导致最后精确度提高。

3.3.8 训练步骤

如果你为你的图片感到高兴，你可以通过修改学习进程中的细节提升你的结果。最简单的方法是用`-how_many_training_steps`。默认是 4000. 但是如果你增加到 8000，他的训练时间将增加到两倍。精确度提高的比率显示你训练的越长一些点将停止，但是你可以试验什么时候达到你的模型的限制。

3.3.9 扭曲

随机通过变形，剪裁，变化输入图像的亮度是一个提高结果的常用方法，这样扩展了训练数据的大小，帮助网络学习 真是生活分类器所有的扭曲，在脚本中使用扭曲最大的缺点是缓冲瓶颈不再有用，因此输入图像将不能重用。这意味着训练京城可能花费更多时间，因此我推荐当着作为一个调节方法调节你的模型到合理。你可以传递`-random_crop`,

random_scale 和 random_brightness 给脚本扭曲图片。百分比值用来控制图片上扭曲用多少部分。, 合理的值时 5 或者 10.-flip_left_right 将在水平方向随机的镜像图像的一半, 有助有你的应用能理解翻转的图像。例如如果你想识别字母这将不是一个好的办法, 因为翻转它们会毁掉原来的含义。

3.3.10 超参数

你可以调整一些参数查看是否对你的结果有帮助, -learning_rate 控制最终层训练更新的幅度。直观理解, 如果这个值变小训练时间将变长, 但是他可能对精度有帮助, 你需要小心试验得到查看什么对于你的 case 生效了。-train_batch_size 控制每一训练步多少图像被检查, 因为学习率应用到每批上, 如果你有更大的批得到相同的效果你将需要减小它。

3.3.11 训练, 验证, 测试集

当你为你的脚本指定图像文件夹时, 文件夹被分成不同的数据集。最大的数据集是训练集, 训练集包含用于训练网络的数据, 用于更新权重。你也许很想知道为什么我们不用所有的图像训练? 一个道德潜在的问题是当我们做机器学习算法时我们的模型会记住接近正确答案的不相关信息, 你可以想想你的图像可能记住了一些照片的背景, 通过标签匹配对象, 它在训练时所有的图像可能产生一个好的结果, 但是不能再新的图像产生好的结果因为他不能泛化对象的特征, 仅仅在训练图像的时候记住了一些不重要的特征。这个问题被称为过拟合, 为了避免过拟合我们保持我们的一些数据不再训练进程中, 因此模型不能记住他们, 我们用这些图像作为检察确保过拟合没有发生, 当我们在看模型在这些数据上有一个好的精度说明过拟合没有发生, 通常 80% 的数据被用来作为训练集 10% 的数据集用来验证最后 10% 的数据用做测试集预测分类器在真实世界的性能, 通过-testing_percentage 和-validation_percentage 标志用来控制比例。通常你应该能留下一些值作为默认, 不应该找到任何好处训练调整他们。注意这个脚本用图象的文件名区分训练集, 验证集, 测试集中的图像 (不是一个随机的函数), 这样保证运行时图片不会再训练集和测试集之间移动, 因为当用于训练模型的图像被验证集中的图像取代时可能会出现一些问题。你也需注意到了在迭代过程中验证正确度的波动。多数波动是验证集的子集的随机性引起的, 选择的验证集用来验证精确度。波动能被最大程度减少, 花费的训练时间增长, 通过选择-validation_batch_size=-1 用整个验证集计算精度。当训练结束后你将能检查测试集中错误分类图像, 这可以通过增加-print_misclassified_test_images 标记, 这对于找到那些什么类型的图片让模型困惑 (很难区别的) 是很有帮助的例如你也许发现了一些种类一些常见的图像角度是特别难识别的, 这样是鼓励你增加更多类型的分类训练子类, 检查催眠五分类图片也指出输入数据中的错误, 向错误标签, 其质量魔术的照片。然而, 你应该避免测试集固定点单个误差, 因为他们仅仅反映在训练集中更多的问题。

3.3.12 更对模型架构

这个脚本默认用 Inception v3 模型架构作为预先训练脚本。这是一个好的开始的地方，因为它提供了高精度的训练结果，但是如果你想部署你的模型到手机设备或者其他资源限制的环境你也许想要这种精确度换区更小的文件尺寸和更快的速度。为了帮助这个 `retrain` 在[移动架构](#)上支持 30 个不同的变量。这里有一些比 Inception v3 更小精度的，但是可以得到更小的文件大小（下载小于兆字节）运行快乐几倍。为了训练这个模型，传递`architecture` 标志，例如：

```
1 python tensorflow/examples/image_retraining/retrain.py \
2   --image_dir ~/flower_photos -- architecture mobilenet_0.25_128_quantized
```

这将在 /temp/ 创建一个 941KB 模型文件 `output_graph.pb`。MobileNet 的 25% 的参数，占据 128×128 大小的输入图像，权重在磁盘中量化为 8 位，你可以选择'1.0', '0.75', '0.50', '0.25' 控制权重参数的数量，因此文件尺寸（和一些扩展速度），'224', '192', '160' 或者 '128' 对于输入图像的尺寸，更小的尺寸更快的速度，选项'`_quantized`' 预示着是否文件应该包含 8 位或者 32 位浮点权重。速度和大小好处带来的是精确度的损失，但是对于一些用途来说是不重要的，他可以通过训练数据提高、例如用扭曲在花数据集允许你得到得到 80% 的精度，甚至 0.25/128、`quantized` 图。如果你在你的程序或者 `label_image` 中用 MobileNet 模型，你讲需要一个输入一个指定大小的图像转换一个浮点让位到 '`input`' tensor，典型的 24 位乳香范围 [0,255] 你必须用 `(image-128.)/128` 转化它到 [-1,1] 范围。

3.4 TF layer 向导：建立一个卷积神经网络

TensorFlow [layers module](#) 是一个用于轻松建立神经网络的高级 API，它提供了一个方法促进创建 `dense`（全连接）层和卷积层，增加激活函数，应用 `dropout` 规则。在这个导航中，你讲学习如何用 `layers` 建立一个卷积神经网络模型识别手写体数据集。[手写体数据集](#) 包含 0-9，60000 个训练样本 10000 个测试样本，图像格式为 28×28

3.4.1 开始

创建文件 `cnn_mnist.py`，在手写体程序中添加如下代码：

```
1 from __future__ import absolute_import
2 from __future__ import division
3 from __future__ import print_function
4
5 # Imports
6 import numpy as np
7 import tensorflow as tf
```

```
8 tf.logging.set_verbosity(tf.logging.INFO)
9
10 # Our application logic will be added here
11
12
13 if __name__ == "__main__":
14     tf.app.run()
```

正如你看到的，你将增加，构造，训练，评估卷积神经网络，最终代码可以点击[这里](#)

3.4.2 介绍卷积神经网络

卷积神经网络是当前最先进的用于图像分类任务的模型架构。CNNs 应用一些滤波器从原始的图像像素中提取高级特征，这个模型可能被用在分类。CNN 包含三个组件：

- **卷积层** 应用指定数量的卷积滤波器在图像上。对于每一个子区域，layer 执行一系列数学操作生成一个单个值在输出 feature map，卷积层然后应用 relu 激活函数输出非线性。
- **池化层** 下采样卷积层的图像数据，减小 feature map 的维度从而减小处理时间。常用池化算法是最大池化 (提取 feature map 子区域) 保留最大值，丢掉其它值。
- Dense layers(**全连接层**) 在通过卷积层和下采样层特征提取执行分类。在全连接层，每一个节点连接到前面的节点。

通常 CNN 有一个卷积模块组成，每个层有卷积模块和池化模块组成。最新的卷积模块有一个或者更多的全连接层链接执行分类。最终 CNN 的全连接层包含每个目标类的一个单个节点 (所有模型可能预测的类)，用 softmax 函数生成一个 0-1 的值 (所有值的和维 1)。我们可以解释给定图像和目标的相似情况。

3.4.3 建立 CNN MNIST 分类器

用 CNN 架构建立模型分类 MNIST 数据集。

1. 卷积层 1: 应用 5×5 卷积核 (提取 5×5 像素的区域)，用 relu 激活函数。
2. 池化层 1: 执行最大池化 2×2 stride=2(指定的池化区域不重叠)
3. 卷积层 2: 应用 64 个 5×5 的卷积核，激活函数为 relu。
4. 池化层 2: 再次执行最大池化操作 (卷积核 2×2) stride=2。
5. Dense 1: 1024 个神经元，dropout=0.4。

6. Dense2:10 个神经元 0-9。

打开 `cnn_mnist.py` 增加下面的符合 TensorFlow's Estimator api 接口的 `cnn_model_fn` 函数。`cnn_mnist.py` 接受 `mnist` 特征数据, 标签, 模型作为参数, 配置 CNN, 返回预测, 损失, 训练操作。

下面的章节函数深入 `tf.layers` 代码创建每一层, 如何计算 loss, 配置训练操作, 生成预测。auguries 你已经体验过 CNN 设 TensorFlow Estimators, 你可以跳到[Training and Evaluating the CNN MNIST Classifier](#)

3.4.4 输入层

这个方法为二维图像数据创建见卷积和池化, 输入 tensor 的形状为 [batch_size,image_width,image_height]

- `batch_size`: 在训练过程执行提图下降的样本数据的子集大小。
- `image_width`: 样本图像的宽。
- `image_height`: 样本图像的高。
- `channels`: 样本图像的颜色通道, 对于彩色图想, 通道为 3, 对于单色图像通道为 1.

在这里, 我们的 MNIST 数据集由 28×28 像素的单色照片组成, 因此输入层的形状为 `[batch_size,28,28,1]`, 转变我们的 feature map 到这个形状, 你可以执行操作:

```
1 input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])
```

这里的-1 表示输入的 `features["x"]` 的值的 batch size 应该被动态计算, 保持所有的其它维度为常数。这允许我们将 `batch_size` 作为一个可以调节的超参数。例如, 如果我们输入样本到我们的 batchs 是 5 的模型, `features["x"]` 将包含 $3920(5 \times 28 \times 28)$ 值 (每一个值代表一个像素点), `input_layer` 形状将为 `[5,28,28,1]`, 类似的如果我们样本的 batchs 是 1000, `features["x"]` 将包含 78400 个值, `input_layer` 形状将为 `[100,28,28,1]`。

3.4.5 第一层卷积层

在我们的卷积层我想用 32 个 5×5 的卷积核到输入层, 用 ReLU 激活函数, 我们一可用 `conv2d()` 方法创建这个层:

```
1 conv1 = tf.layers.conv2d(
2     inputs=input_layer,
3     filters=32,
4     kernel_size=[5, 5],
5     padding="same",
```

```

6     activation=tf.nn.relu
7 )

```

inputs 参数指定我们的输入 tensor(形状为 [batch_size,image_width,image_height,channels]), 这里, 我们链接我们的第一个吉安基层到输入层, 形状为 [batch_size,28,28,1] 注意: 如果传递参数 data_format=channels_first,conv2d() 接受 [channels,batch_size,image_width,image_height] 形状的数据。

filter 参数制定卷积核的个数, 这里卷积核为 32 个。kernel_size 制定卷积核的维度为 [width,height] (这里 [5,5]) padding 参数制定两个值:valid(默认), 和 same。制定输出 tensor 应该有和输入特征是偶然相同的形状, 我们设置 padding=same, 说明 TensorFlow 增加 0 值到输出 tensor 的边缘波啊池宽度和高度为 28(没有 padding 5×5 卷积 28×28 将生成 24×24 tensor, 在 28×28 用 5×5 提取出 24×24 个位置)。activation 参数指定应用到输出的激活函数, 这里我们只顶 tf.nn.relu。conv2d() 的输出形状为 [batch_size,28,28,32]: 和输入有相同的宽度和高度, 但是有 32 个通道保持每个卷积核的输出。

3.4.6 池化层 1

链接我们创建的卷积层和池化层, 我们在 layers 中用 max_pooling2d() 方法构造执行最大池化, 卷积核 filter 大小为 2×2 , stride 为 2。

```

1 pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

```

再次, inputs 制定输入 tensor, 形状为 [batch_size,image_width,image_height,channels], 这里我们的输入 tensor 是第一层卷积层的输出 conv1, 形状为 [batch_size,28,28,32]

pool_size 指定最大池化 filter 的大小作为 [width,height] (这里是 [2,2]) 如果两个维度相等你可以指定 pool_size=2。strides 参数制定 stride 的大小, 这里我们设置 strides 为 2, 表示通过 filter 提取子区域的时候宽度和高度都是 2 像素。如果你想设置不同的 width 和 height, 你可以制定一个元祖或者列表。

我们的输出特征是偶然和 max_pooling2d(pool1, 形状为 [batch_size,14,14,32]) 相乘: 2×2 减少宽度和高度到 50%。

3.4.7 二层卷积和池化

我们用 conv2d() 和 max_pooling2d() 链接卷积和池化。对于卷积层 2, 我们配置 64 个 5×5 的卷积核, 激活函数为 ReLU, 池化层 2, 我们用和池化层一眼个间隔:

```

1 conv2 = tf.layers.conv2d(
2     inputs=pool1,
3     filters=64,

```

```

4     kernel_size=[5, 5],
5     padding="same",
6     activation=tf.nn.relu)
7
8 pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

```

卷积层用 pool1 作为输入，生成 tensor conv2。conv2 形状为 [batch_size, 14, 14, 64]，和 pool1 的宽和高相等，64 个通道因为 64 个卷积核。

池化层 2 那 conv2 作为输入，生成 pool2 作为输出，pool2 形状 [batch_size, 7, 7, 64]（减少 conv2 50% 的宽度和高度）

3.4.8 Dense layer

我们添加 dense 层（1024 个神经元和 ReLU 激活函数）到 CNN 生成卷积/池化层提取的特征分类，我们将 flatten 我么呢 feature map(pool2) 到形状 [batch_size, features]，因此我们的 tensor 有两维，上面的形状变成了 [batch_size, 7 × 7]：

```

1 pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])

```

现在哦我们用 dense 方法链接我们的 dense:

```

1 dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)

```

inputs 参数制定输入 tensor：我们的 flattened 的 feature map pool2_flat。units 参数指定 dense 层的神经元的数量。activation 参数获取激活函数，这里我们依然是用 tf.nn.relu。为了改进我们的模型，我们也应用 dropout 方法正则化 dense 层。

```

1 dropout = tf.layers.dropout(
2     inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

```

inputs 参数和上面一样，rate 参数制定 dropout 比率，这里用 0.4 表示 40% 的元素将在训练中被随机丢弃。training 参数得到一个 bool 值制定是否模型在训练模式下运行，dropout 仅仅在 training 为 True 时执行。这里我们检查是否 mode 传递给我们 cnn_model_fn 的模型函数是 TRAIN 模式。输出形状为 [batch_size, 1024]

3.4.9 Logits Layers

在我们神经网络的最后一层是 logits 层，然会预测的原始值。我们用 10 个神经元创建一个 dense layers，激活函数哦认为线性激活函数。

```

1 logits = tf.layers.dense(inputs=dropout, units=10)

```

我们最终输出 CNN 的 tensor，logits 形状为 [batch_size, 10]。

3.4.10 常见的预测

logits 层返回我们预测的原始值 (形状 [batch_size,10])。让我们转化这些原始值到我们的模型函数能返回的两种个不同的格式。

- predicted class: 数字 0-9。
- probabilities: 对于每个可能的目标类的概率。

对于更定的例子，我们的预测类是在相关行 logits 列有最大的值。我们可以用该 tf.argmax 函数找到这个元素的索引。

```
1 tf.argmax(input=logits, axis=1)
```

input 参数制定需要提取最大值的 tensor, axis 参数制定输入 tensor 沿着哪个轴寻找最大值。这里我们写着 1 轴寻找最大值。我们可以用 softmax 生成概率。

```
1 tf.nn.softmax(logits, name="softmax_tensor")
```

我们融合我们的预测到一个字典中，返回一个 EstimatorSpec 对象。

```
1 predictions = {
2     "classes": tf.argmax(input=logits, axis=1),
3     "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
4 }
5 if mode == tf.estimator.ModeKeys.PREDICT:
6     return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)
```

3.4.11 计算 Loss

对于训练和评估阶段，我们需要定义损失函数衡量我们的模型的预测如何接近目标类。对于想 MNIST 的多个分类问题，cross entropy 是典型的被用做损失度量。下面的代码计算交叉熵返回 TRAIN 或者 EVAL 模式：

```
1 onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
2 loss = tf.losses.softmax_cross_entropy(
3     onehot_labels=onehot_labels, logits=logits)
```

我们的 labels tensor 包含一个预测列表，像 [1,9,...]，为了计算交叉熵，你需要转换 labels 为相关的one-hot encoding

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

womenyoingtf.one_hot 函数执行转换。tf.one_hot() 有两个参数：

- one-hot tensor 有值的位置，如上面 1，表示位置索引为 1 的地方有 1
- depth:one-hot tensor 的深度，目标类的数量，这里 depth 为 10，

下面的代码为我们的 labels 创建一个 one-hot tensor，onehot_labels:

```
1 onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
```

因为 labels 包含值从 0-9,indices 是我们的 labels tensor，值变为证书。depth 是 10 因为我们有 10 个可能的目标类。下一步我们计算 onehot_labels 的交叉熵和我们的 logits 层的 softmax 预测。tf.losses.softmax_cross_entropy() 得到 onehot_labels 和 logits 作为参数。在 logits 上执行 softmax 激活函数，返回损失的标量 tensor:

```
1 loss = tf.losses.softmax_cross_entropy(
2     onehot_labels=onehot_labels, logits=logits)
```

3.4.12 配置训练操作

在先前的操作中我们为我们的 CNN 定义了损失作为 logits 层和 layers 的 softmax cross-entropy。让我们配置我们的模型在训练落成中优化 loss。我们将用 0.001 学习率和 SGD 作为优化算法

```
1 if mode == tf.estimator.ModeKeys.TRAIN:
2     optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
3     train_op = optimizer.minimize(
4         loss=loss,
5         global_step=tf.train.get_global_step())
6     return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)
```

3.4.13 增加评估度量

为了增加度量到我们的模型，我们在 EVAL 定义了 eval_metric_ops 字典:

```
1 eval_metric_ops = {
2     "accuracy": tf.metrics.accuracy(
3         labels=labels, predictions=predictions[ "classes" ])}
4 return tf.estimator.EstimatorSpec(
5     mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

3.5 训练评估 CNN MNIST 分类器

我们已经构建了 MNIST CNN 模型函数，现在我们准备训练评估它。

3.5.1 载入训练和测试数据

增加 main() 函数到 cnn_mnist.py 载入训练数据和测试数据。

```

1 def main(unused_argv):
2     # Load training and eval data
3     mnist = tf.contrib.learn.datasets.load_dataset("mnist")
4     train_data = mnist.train.images # Returns np.array
5     train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
6     eval_data = mnist.test.images # Returns np.array
7     eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)
```

我们存储训练数据 train_data(55000 张原始图像的像素值) 训练 train_labels(每张图片 0-9) 作为 numpy 数组。类似的我们存储评估数据 (10000 张)eval_data 和 eval_labels。

3.5.2 创建 Estimator

下一步创建一个 Estimator(一个用于执行高级模型训练, 评估, 推理的 TensorFlow 类), 增加下面代码到 main() 中。

```

1 # Create the Estimator
2 mnist_classifier = tf.estimator.Estimator(
3     model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model")
```

model_fn 参数指定用于训练, 评估, 预测的模型函数, 我们传递 cnn_model_fn, models_dir 参数制定模型数据的保存目录为 /tmp/mnist_convnet_model。

3.5.3 建立 Logging Hook

因为 CNN 可能花一会训练, 让我们设置一些采集以至于我们在训练时能跟踪进层。我们用 TensorFlow 的 tf.train.SessionRunHook 创建一个 tf.train.LoggingTensorHook 采集从 softmax 层来的概率值, 增加下面代码到 main():

```

1 # Set up logging for predictions
2 tensors_to_log = {"probabilities": "softmax_tensor"}
3 logging_hook = tf.train.LoggingTensorHook(
4     tensors=tensors_to_log, every_n_iter=50)
```

我们存储一个我们想要采集进 tensors_to_log 的 tensor 词典。每个 key 是我们选择的 label, 将在采集输出被打印, 相关的 label 是 TensorFlow 图的 Tensor 的名字, 这里我们的概率可以在 softmax_tensor 中找到, 我们给我们 softmax 操作的名字在 cnn_model_fn 生成概率。

下一步我们创建 LoggingTensorHook, 传递 tensor_to_log 到 tensors 参数, 我们设置 every_n_iter=50, 制定训练的时候每 50 步采集概率。

3.5.4 选练模型

现在我们准备好训练我们的模型，我们通过创建 train_input_fn 和在 mnist_classifier 调用 train()，增加下面到 main()

```

1 # Train the model
2 train_input_fn = tf.estimator.inputs.numpy_input_fn(
3     x={"x": train_data},
4     y=train_labels,
5     batch_size=100,
6     num_epochs=None,
7     shuffle=True)
8 mnist_classifier.train(
9     input_fn=train_input_fn,
10    steps=20000,
11    hooks=[logging_hook])

```

在 Numpy_input_fn 调用的时候，我们传递训练特征数据和标签给 x 和 y。我们设置 batch_size 是 100（模型训练的时候每次最小批次是 100 个样本）。num_epochs=None 意味着模型将训练直到指定步数到达。我们也设置 shuffle=True 打乱训练数据，在训练调用的时候，我们设置 steps=20000(这意味着模型总共训练 20000 次) 我们传递 looging_hook 去 hooks 参数，以至于它将在训练期间被触发。

3.5.5 评估模型

当训练结束是我们想要在测试及评估我们的模型，我们可以调用 evaluate 方法，在 model_fn 指定 eval_metric_ops 参数度量方法:

```

1 # Evaluate the model and print results
2 eval_input_fn = tf.estimator.inputs.numpy_input_fn(
3     x={"x": eval_data},
4     y=eval_labels,
5     num_epochs=1,
6     shuffle=False)
7 eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
8 print(eval_results)

```

为了创建 eval_input_fn，我们设置 num_epochs=1，因此模型评估在一个时期评估数据返回结果。我们也设置 shuffle=False 通过数据序列迭代。

3.5.6 运行模型

下面是采集的输出:

```
1 INFO:tensorflow:loss = 2.36026, step = 1
2 INFO:tensorflow:probabilities = [[ 0.07722801  0.08618255  0.09256398, ...]]
3 ...
4 INFO:tensorflow:loss = 2.13119, step = 101
5 INFO:tensorflow:global_step/sec: 5.44132
6 ...
7 INFO:tensorflow:Loss for final step: 0.553216.
8
9 INFO:tensorflow:Restored model from /tmp/mnist_convnet_model
10 INFO:tensorflow:Eval steps [0,inf) for training step 20000.
11 INFO:tensorflow:Input iterator is exhausted.
12 INFO:tensorflow:Saving evaluation summary for step 20000: accuracy = 0.9733, los
13 {'loss': 0.090227105, 'global_step': 20000, 'accuracy': 0.97329998}
```

我们在测试集上获得了 97.3% 的精确度。

Chapter 4

扩展

这个章节解释开发者如何增加功能到 TensorFlow。

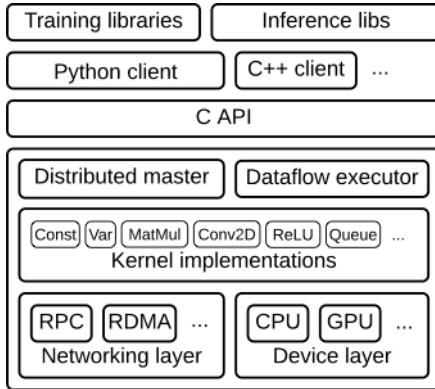
4.1 TensorFlow 架构

我们设计 TensorFlow 是为了大规模分布式训练和推理，但是它也能灵活的支持一些新的机器学习模型实验和系统级别的优化。

这个文件描述了这个系统架构使得结合这个规模和灵活度成为可能。假设你熟悉 TensorFlow 基本的一些概念，像计算图，操作绘画。这个文档适合于那些想用当前 API 不支持的一些方法扩展 TensorFlow，想要优化 TensorFlow 的硬件工程师，在法规莫分布式系统上实现机器学习系统或者是任何想要了解 TensorFlow 的 hood 的人。读完它后你应该能读和修改 TensorFlow 核心代码。

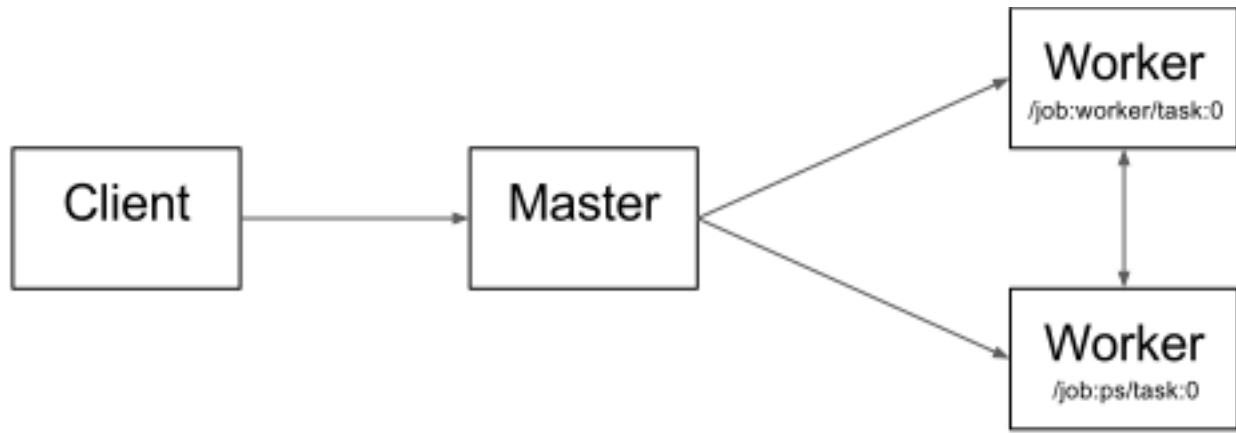
4.2 概述

TensorFlow 运行时是一个跨平台的库，下图画出了常用的架构，C API 分隔用户代码和核心代码。



- Client
- 定义计算作为数据流图。
- 用 session 初始化图。
- Distributed Master
 - 从图中修剪一个子图作为定义的参数给 Session.run()
 - 分开不同的子图为多个部分在不同的进程和设备上运行。
 - 分配图块到 worker service。
- Worker services
 - 调度图上的操作在可用的硬件平台 (CPUs, GPUs) 上执行。
 - 发送和接收 worker service 的操作结果。
 - 内核实现。
- 执行单个图操作的计算。

下图说明逐渐的交互。”job:worker/task:0” 和”/job:ps/task:0” 两个任务在 workers 上。”PS” 代表”parameter server”: 一个负责存储更新模型参数的任务。另一个任务优化参数时发送更新到这些参数，类似的在任务之间的分隔是不被要求的，但是它通常用于分配的训练。



注意 Distributed Master 和 Worker Service 仅仅存在于分布的 TensorFlow。, 单进程版本的 TensorFlow 包含一个特别的 Session 实现能做任何 Distributed master 能做的不仅仅是和本地进程通信。

下面的章节表述了 TensorFlow 的核心。

4.2.1 Client

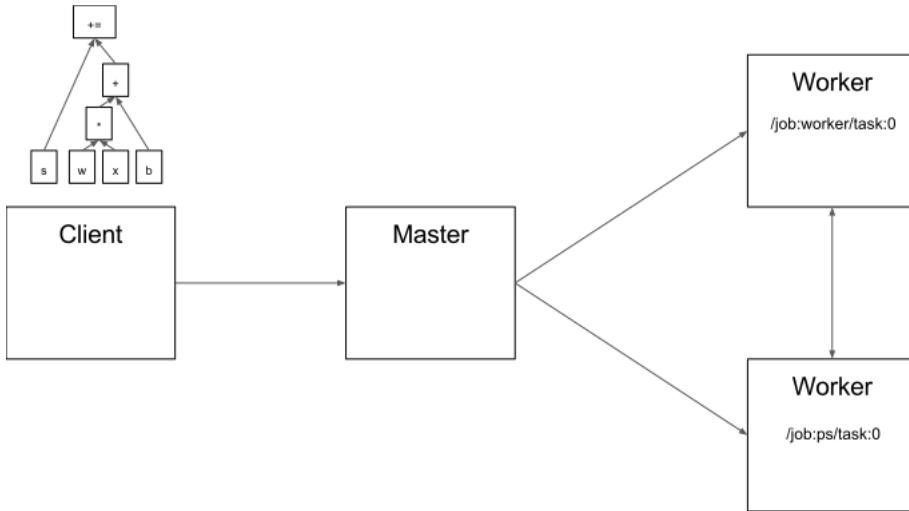
用户写 TensorFlow 程序构造计算图。这个图给需既可以组成单个操作或者用一个像 Estimators API 的方便的库组成神经网络乘和其他高级抽象。TensorFlow 支持多种用户语言，但是我们优先使用 Python 和 C++，仅仅是因为我们的内部用户熟悉他们。当特征被建立好后我们将他们接入 C++。因此用户可以得到一个对所有语言优化的实现。大多数的训练库仅仅支持 Python，但是 C++ 支持更高效的推理。用户创建一个绘画，发送图的定义到 distributed master 作为 tf.GraphDef 协议缓冲区。然后客户评估图上的一个节点或者多个节点，评估触发一个 distributed master 的调用初始化计算。

在下图中，客户建立一个图，应用权重 (w) 到特征向量 (x)，增加偏置 (b) 保存结果。

4.2.2 Distributed master

- 修剪图得到子图计算用户的节点请求。
- 对于每一个加入的设备，分隔图获得子图。
- 缓存这些块以至于他们能用在自序列中。

因为 master 查看每一步的计算，它用像常用的子表达式消除和常数折叠的标准的优化。它然后执行优化的子图。下图显示一个可能的分隔。distributed master 有组合的模型参数为了放置他们在参数服务器上。这里图的边缘被分隔，distributed master 发送接收节点在不同的任务间传送信息。下面的 distributed master 传输子图到分布的任务。



4.2.3 Worker Service

任务中的 worker service。

- 处理 master 的请求。
- 调度内核执行包含本地子图的操作
- 任务间的直接通信。

我们优化 worker service 为了能用更小的花销以支撑大的图。我们当前的实现能实现每秒执行上万张子图，使得大量的副本快速的训练。worker service 布置内核到本地设备上然后通过利用多 CPU 多 GPU 尽可能的并行执行。我们为源和目的设备指定发送和接收操作。

- 用 cudaMemcpyAsync() API 在本地 CPU 和 GPU 之间转换，覆盖计算和数据的转化。
- 用对等的 DMA 在不同的本地 GPU 之间转化避免通过主 CPU 的高昂代价。

对于任务间的转化，TensorFlow 用多个协议，报错：

- gRPC over TCP
- RDMA over Converged Ethernet

我们对于 NVIDIA 的多 GPU 通信 NCCL 库有初步的支持，查看 tf.contrib.nccl

4.3 内核实现

运行包含超过 200 个标准操作白扩数学，数组操作，控制流，状态管理操作。每一个操作对不同的设备有优化，一些操作内核用 Eigen::Tensor 实现，用 C++ 模板生成在多核 CPU 和 GPUs 上生成高效的并行代码，然而我们优先像像 CuDNN 这类更高效实现的库。我们也实现了量化，能在移动设备和高流通数据中心应用上更快地推理，用 gemmlowp 地精读矩阵库加速量化计算。

如果很难或者抵消的表达子计算作为操作的组成，用户可以注册额外的京城通过 C++ 提供更高效的实现，，我们推荐你 duit 一些重要的操作像 ReLU 和 Sigmoid 和相关的梯度注册你的融合内核，XLA 变压器有意额实验性是实现自动内核融合。

Chapter 5

Performance

这个导航包含一个优化你的 TensorFlow 代码的集合。对于 Tensorflow 用户来说这是最好的应用，正如在这个文档中最好的时间，高性能模式为在不同的硬件上创建模型文档链接到示例代码。

5.1 最好的实践

尽管优化实现不同类型的模型可能不同，下面是通过 tensorflow 实现性能的几个最好的方式，尽管这些暗示在基于图像的模型，我们将增加一些笑技巧到所有类型的模型。下面列出了最好实践的关键：

- 从原来码编译安装
- 利用队列读取数据
- 在 CPU 上预处理
- 用 NCHW 图像格式
- 在 GPU 上放共享参数
- 用融合的批处理规范

下面章节时处理的详细信息。

5.2 从源代码创建安装

为了安装最优化的 TensorFlow 版本，通过源代码编译安装 Tensorflow。从原来码编译优化目标硬件确保最新的 CUDA 平台和 CuDNN 库被用高性能安装。

对于多数稳定的实验，从最新版的[latest release](#)分支编译。为了得到最新性能改变接受一些稳定性风险，从[master](#)编译。

如果你需要在不同的目标硬件平台上编译 TensorFlow，交叉编译最优化目标平台。下面的目录是一个例子高数 bazel 为指定平台编译

```
1 # This command optimizes for Intel's Broadwell processor
2 bazel build -c opt --copt=-march="broadwell" --config=cuda //tensorflow/tools/
                                         pip_package:build_pip_package
```

环境，构建，安装技巧

- 编译最高级别的[GPU 支持](#)，e.g. P100: 6.0, Titan X (pascal): 6.2, Titan X (maxwell): 5.2, and K80: 3.7.
- 安装最新版的 CUDA 平台和 cuDNN 库
- 确保你的 gcc 版本支持对目标 cpu 所有的优化，推荐最小的 gcc 版本为 4.8.3
- TensorFlow 在启动时检查是否已经在 cpu 上编译优化过，如果优化不被包含，TensorFlow 将 chxuan 警告，e.g. AVX, AVX2 和 FMA 设备不被包含。

5.2.1 利用队列读取数据

在利用 GPUs 时性能很差或者没有设置高效的 pipeline 导致缺乏数据，确保设置输入 pipeline 高效利用队列和流数据，一种识别 GPU 处于饥饿状态的方法时生成和查询时间线。一个相信的时间线指南不存在，但是一个快速生成时间线的例子在[XLA JIT](#)部分存在，另一个检查是否 GPU 被充分使用时运行 nvidia-smi 查看，如果 GPU 利用没有达到 100% 这样 GPU 没有足够的快的得到数据。

除非指定一个特殊的情形或者示例代码，没有从 Python 变量给予数据到会话，e.g.

```
1 # Using feed_dict often results in suboptimal performance when using large
      inputs.
2 sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

5.2.2 在 CPU 上的预处理

将预处理操作放在 CPU 上可能对性能提升很重要，当预处理发生在 GPU，数据流使从 CPU->GPU(预处理)->CPU->GPU(训练)。这数据被限制在 CPU 和 GPU 之间，当预处理被放在 CPU 上，数据流是 CPU(预处理)->GPU(训练)。另一个好处是在 CPU 上预处理释放 GPU 时间让其集中训练。

将预处理放在 CPU 上可能导致对 sample/sec 处理速度 6 倍以上的处理性能增加，将导致训练时间缩短为原来的 $\frac{1}{6}$ ，确保预处理在 CPU 上，按照如下操作：

```
1 with tf.device('/cpu:0'):
2     # function to get and process images or data.
3     distorted_inputs = load_and_distort_images()
```

5.2.3 用大文件

在一些情形下，CPU 和 GPU 可能通过 I/O 操作获取数据时对数据处于饥饿状态。如果你正用一些小文件形成输入数据集，你也许被你的文件系统限制了速度。如果你在 SSD 上而不是 HDD 上存储你的输入数据你的训练循环运行更快。如果是这样你应该通过创建一些大的 TFRecord 文件预处理你的输入数据。

5.2.4 用 NCHW 图像数据格式

图像数据格式涉及到图像的批量表示。TensorFlow 支持 NHWC(TensorFlow 默认) 和 NCHW(cuDNN 默认)，N 时图像的批数，H 时图像垂直方向的像素数量，W 是水平方向的像素，C 时图像的通道数，尽管 cuDNN 能处理上面两种格式，但是它处理默认格式更快。最好的实现是用 NCHW 和 NHWC 构建模型正如通常在 GPU 上用 NCHW 训练然后在 CPU 上用 NHWC 推断。

TensorFlow 用这两个格式是的一个简单的历史因为它在 CPUs 上运行快点，然后 TensorFlow 团队发现当 NVIDIA cuDNN 库时 NCHW 运行更好。当即用户推荐在他们的模型中支持两种格式，在很长一段时期，我们计划重写图转化两种格式。

5.2.5 用融批规范

当用批规范 `tf.contrib.layers.batch_norm` 设置属性 `fused=True`:

```
1 bn = tf.contrib.layers.batch_norm(
2     input_layer, fused=True, data_format='NCHW'
3     scope=scope, **kwargs)
```

在没有融合批规范计算几个单独的操作。融合批规范结合单个操作进入内核，运行更快。

5.3 性能向导

5.4 好性能模型

5.5 Benchmark

5.6 如何用 TensorFlow 量化神经网络

现代神经网络已经被开发出了，最大的挑战是让他们工作！这意味着在训练中的精确度和速度被优先考虑，浮点时是保留精确度的最简单的方法，GPUs 擅长简爱素这些计算，因此没有太多的注意被放在其它数据格式上。

这些天我们做了一些模型部署在商业应用上，训练的计算要求随着研究人员的数量增加，对于推断的需要正在扩张。这意味着推断效率变成的一些团队最麻烦的问题。

这是量化出现了，它覆盖了一些存储数字和计算执行在更多兼容的 32bit 浮点数。我们将关注固定点下面我将说宁更多细节。

5.6.1 为什么做量化工作

训练神经网络通过对权值小的推动，这些小的推动需要浮点精度工作。

预先训练模型和运行推断有很大不同，一个深度网络的神奇的量化时他们像是复制高级别的噪声在他们的输入。如果考虑识别一个你拍摄照片中的的对象，网络必须在它和训练样本中忽视 CCD 上的噪声，光线改变其它不重要的差异在它和训练样本被看到前，之一梨放在重要的类似的事上。这个能力意味着他们需要退待地精读计算作为另一个源噪声，产生精度结果升值数值格式抓住更少信息。

5.6.2 为什么量化

神经网络模型可能占据一些磁盘空间，原始的 AlexNet 腹地使能数据占据超过 200MB。大多数的这些大小被神经网络连接的权重占据，因为经常单个模型有上百万个神经元。因为他们时有一些不同的浮点数，简单的压缩格式像 zip 不能很好的压缩他们，他们被安排仅一个大的层，每一层的权重趋向于一定范围的正态分布，比如-3.0 到 6.0。

最简单的量化动机是通过存储每一层的最大值和最小值缩小文件大小。然后压缩每个浮点值为 8 位代表最接近到 256 内的真实整数。例如范围-3.0-6.0,0 代表-3,255 代表 6.0,128 代表 1.5。我在之后将进行确切的计算，因此有一些细节，但是这意味着你可以得到缩小文件尺寸 75% 的好处，然后你可以通过导入后在不更改任何存在的浮点数代码然后转换为浮点数。

另一个原因是量化前通过在 8 位输入输出减少你运行前的你需要推理计算资源，获取 8 位值仅仅需要浮点数 25% 的内存带宽，因此你可以更充分使用缓存避免 RAM 存取瓶颈，你也可以用 SIMD 操作在每个时钟周期做更多操作。在 yxiieqingkuangxia 你将有一个 DSP 芯片可以激素 8 为计算得到更多好处。

移动嗯计算到 8 为将帮助你更快地运行模型，用更少的电量，同时它也为不能运行浮点代码的嵌入式系统打开了开了一扇门，因此可以应用到 IoT 世界。

5.6.3 为什么不直接训练低精度

我们正在一些更低深度上做了一些实验，结果似乎显示你需要高于 8 位处理反向传播和梯度，这使得实现训练变得更复杂推理混乱。我们已经有一致的浮点数模型使用，因此能直接方便的转换他们。

5.6.4 你能如何量化你的模型

TensorFlow 支持生成 8 位计算，它也有一些通过用量化计算推理转换训练模型浮点数到相应的图上。例如这里你可以转换最新的 GoogleLeNet 模型用 8 位版本计算：

```
1 curl http://download.tensorflow.org/models/image/imagenet/
2 inception-2015-12-05.tgz -o /tmp/inceptionv3.tgz
3 tar xzf /tmp/inceptionv3.tgz -C /tmp/
4 bazel build tensorflow/tools/quantization:quantize_graph
5 bazel-bin/tensorflow/tools/quantization/quantize_graph \
6   --input=/tmp/classify_image_graph_def.pb \
7   --output_node_names="softmax" --output=/tmp/quantized_graph.pb \
8   --mode=eightbit
```

你将在原始的操作上运行一个新的模型，但是 8 位计算在内部，所有的权重被量化。如果你查看文件尺寸，你将看到大约是原来的 1/4(23MB 对比 91MB)，你可以用相同的输入输出运行这个模型，你将得到相应的结果，这里是代码：

```
1 # Note: You need to add the dependencies of the quantization
2 # operation to the
3 #       cc_binary in the BUILD file of the label_image program:
4 #
5 #       //tensorflow/contrib/quantization:cc_ops
6 #       //tensorflow/contrib/quantization/kernels:quantized_ops
7
8 bazel build tensorflow/examples/label_image:label_image
```

```

9  bazel-bin/tensorflow/examples/label_image/label_image \
10 --image=<input-image> \
11 --graph=/tmp/quantized_graph.pb \
12 --labels=/tmp/imagenet_synset_to_human_label_map.txt \
13 --input_width=299 \
14 --input_height=299 \
15 --input_mean=128 \
16 --input_std=128 \
17 --input_layer="Mul:0" \
18 --output_layer="softmax:0"

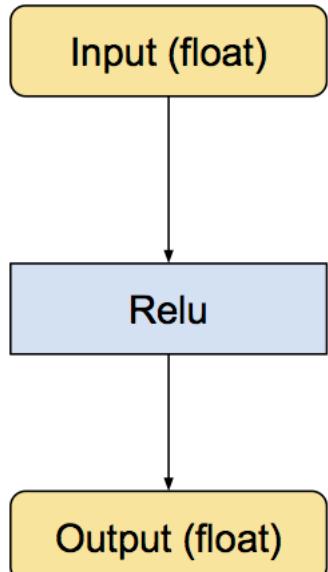
```

你将看到最新的量化图的运行，输出和原始输出十分类似。

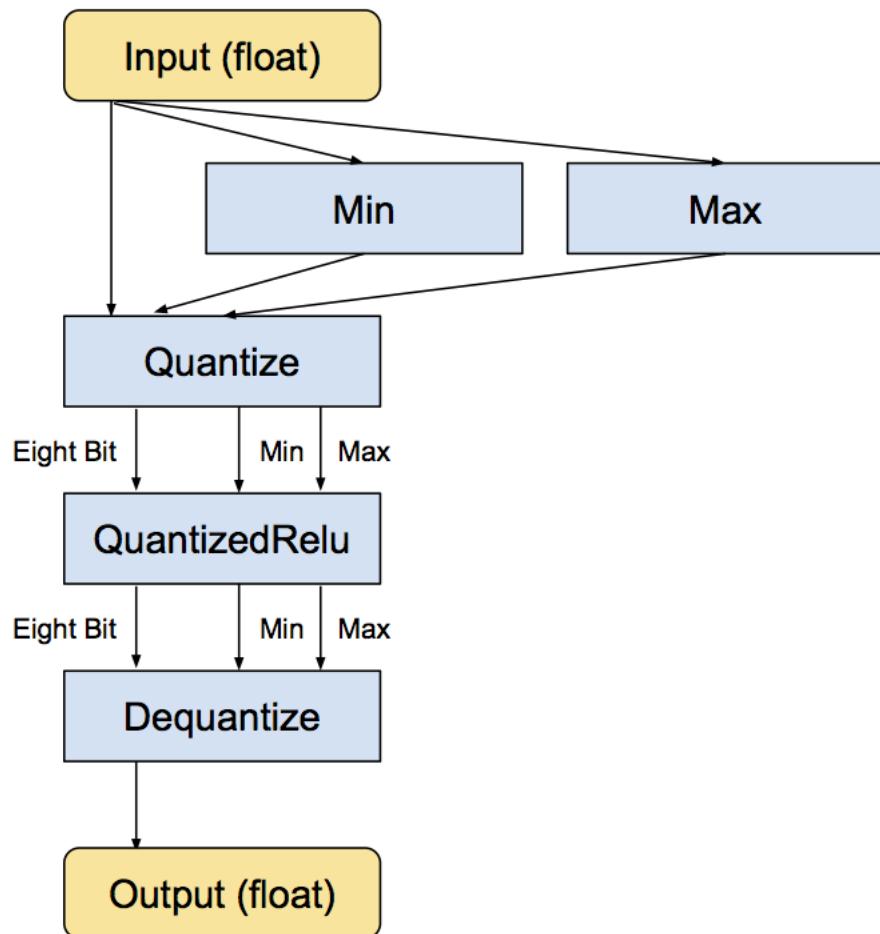
你可以运行相同的处理在你的模型上报春为 GraphDefs，输入输出的名字用在你的网络请求上。我推荐你首先通过 freeze_graph 脚本，转化检查点为常数存储在文件中。

5.6.5 如何量化处理工作

我们在推理过程中通过写 8 位量化本本操作实现两话，这包含卷积，矩阵相乘，激活函数，吃花草做和链接，转化脚本首先对所有的操作量化。有一些小的子图之前有转化函数之后在浮点数和 8 位数之间移动，下面是一个例子，首先原始 Relu 操作输入输出浮点数。

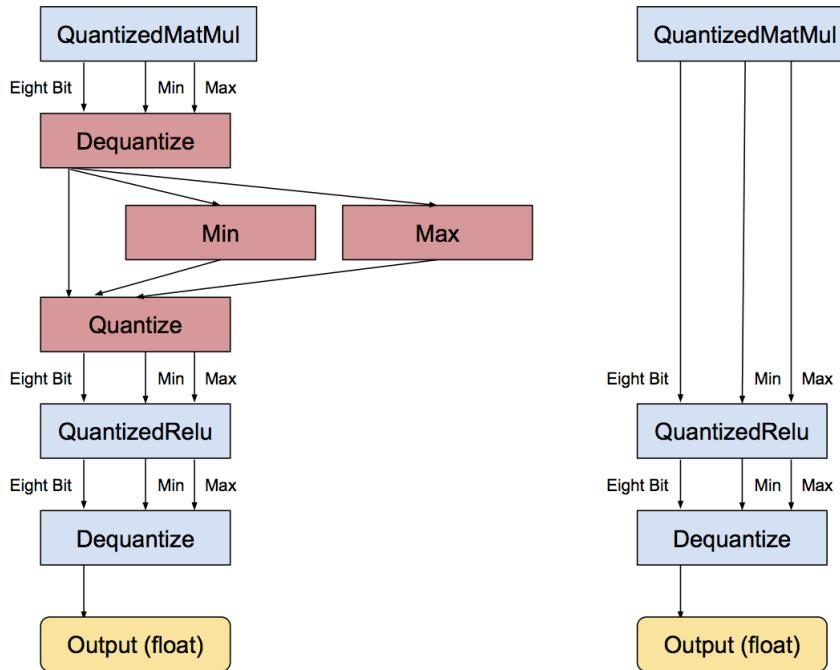


然后相应的转换子图，仍然是浮点输入输出，内部转换完成后以 8 位计算：



最小，最大操作实际上查看输入浮点 tensor 的值，输入他们到量化操作转换 tensor 为 8 位。

当单个操作被转换后，下一步是移除必要的转换到浮点。如果有一个连续的序列操作，将有一些链接 Dequantize/Quantize 操作



应用到大规模模型上时所有的操作已经相应的量化，图上所有的计算用 8 为计算不转换为浮点数。

5.6.6 量化 Tensor 将呈现什么

我们通过转化浮点数组为 8 位表达式作为压缩问题。我们知道权重和激活 Tensor 在训练神经网络模型时值的分布在一个小的范围（你也许有一个 -15 到 +15 的权重，-500 到 1000 激活）。在实验中我们了解到神经网络通常在处理噪声时非常健壮，由量化产生噪声类似的误差将不会伤害精度。我们像卷则一个表达式这是容易执行计算，特别是大的矩阵惩罚影城需要运行一个模型的块。

这导致我们选择一个表达式有两个浮点数存储最小值和最大值代表最低和最高量化精度，每个在量化数组中的入口代表一个浮点值范围，现行飞蛾分布在最小值和最大值之间。例如我们有最小值 =-10.0 和最大值 30.0f，和 8 位数据，下面是量化表达式：

量化值	浮点数
0	-10
255	30.0
128	10.

这种表达式的好处是可以代表任一幅度的范围，我们不必退成，它可以代表有符号和无

符号的值，现行扩展使得直接相乘。对转换浮点数前后的一个清晰明确的量化格式定义好处，或者基于调试目的的查看 tensor 在 Tensorflow 上一个是线细节时希望提高将来最小值和最大值需要传递分割开得 Tensor 保持量化值，因此图个变得一点稠密。

最小和最大值范围可以提前计算，权重参数是常数在载入时就知道，因此他们的范围可以作为常数被存储。我们经常知道输入范围例如 (RGB 的值在 0-255)，一些激活函数也知道范围。这可以避免必须分析操作的输出决定范围，我们需要从 8 位输出像卷积或者矩阵乘法这样的做数学操作形成 32Bit 累加结果。

5.6.7 下一步

我们发现通过 8 为算法而不是不浮点数可以在移动短和嵌入式设备上得到机器好的性能。你可以看到这个框架我们优化矩阵乘在[gemmlowp](#)，我们仍然需要应用所有的我们需要学习的 TensorFlow 操作去在移动短得到最大型能，但是我们很兴奋正在为此努力。马上，量化实现是一个合理的快和精确度实现我们希望将能在更多的设备上广泛的支持 8 位模型。我们也希望站着将鼓励社区探索更低精度的神经网络的可能性。

Chapter 6

常用的 python 模块

6.1 Argparse

argparse 模块是一个用户友好的命令行接口，当用户每有给定可用的参数时，argparser 能自动生成帮助和使用信息。

```
1 import argparse
2 parser = argparse.ArgumentParser(description='Process some integers.')
3 parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer
4                     for the accumulator')
5 parser.add_argument('--sum', dest='accumulate', action='store_const', const=sum,
6                     default=max, help='sum the integers(
7                     default: find the max)')
8 args = parser.parse_args()
9 print(args.accumulate(args.integers))
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ vim code/demo1.py
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo1.py
usage: demo1.py [-h] [--sum] N [N ...]
demo1.py: error: the following arguments are required: N
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo1.py 1 2 3 4
4
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo1.py --sum 1 2 3 4
10
```

代码能根据传入的参数选择相应的函数计算。

- 创建一个 parser
- 增加 arguments
- 解析参数

6.1.1 ArgumentParser 对象

```
class argparse.ArgumentParser(prog=None, usage=None, description=None, epilog=None,
parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-', fromfile_prefix_chars=None,
argument_default=None, conflict_handler='error', add_help=True, allow_abbrev=True)
```

- prog: 程序的名字 (默认为 sys.argv[0])
- usage: 描述程序用法的字符串。 (默认通过 arguments 增加到 parser)
- description: argument 帮助前的文本展示。 (默认为:None)
- epilog: argument 帮助之后的文本展示。 (默认为:None)
- parents: 应该被包含的列表对象。
- formatter_class: 自定义输出帮助的类。
- prefix_chars: 参数前面的字符。 (默认为'-')
- fromfile_prefix_chars: 应该被读的文件的字符串。
- argument_default: 参数的全局值。 (default:None)
- conflict_handler: 解决冲突选项的策略。 (通常不是必需的)
- add_help: 增加-h/-help 选项到 parser。 (默认为 True)
- allow_abbrev: 如果缩略不冲突, 可以允许长的选项被缩略。 (默认为 True)

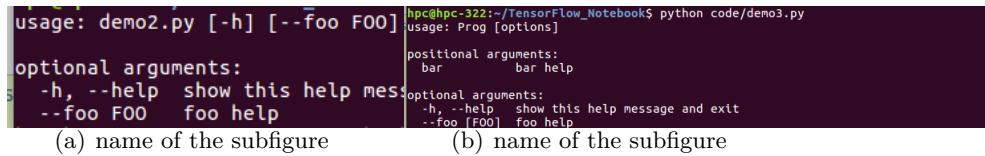
6.1.2 prog

默认情况下 ArgumentParser 对象用 sys.argv[0] 决定如何显示程序的名字。

```
1 #filename : arg1.py
2 import argparse
3 parser = argparse.ArgumentParser()
4 parser.add_argument("echo")
5 args = parser.parse_args()
6 print(args.echo)
```

默认情况下 ArgumentParser 从包含用法信息的参数计算 usage message。

```
1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument('--foo', help='foo help')
4 args = parser.parse_args()
```



大多数的 ArgumentParser 构造体用 `description=` 关键字，这个参数给出一个简单的程序说明其如何工作的。在帮助信息中表述在命令行和帮助信息之间。

```
1 import argparse
2 parser = argparse.ArgumentParser(description='A foo that bars')
3 parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo4.py
usage: demo4.py [-h]

A foo that bars

optional arguments:
  -h, --help  show this help message and exit
```

一些程序喜欢在参数表述后添加一些额外的信息说明，这些说明可以通过 ArgumentParser 中的 `epilog=` 参数指定。

```
1 import argparse
2 parser = argparse.ArgumentParser(description='A foo that bars',
3 epilog="And that's how you'd foo a bar")
4 parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo5.py
usage: demo5.py [-h]

A foo that bars
|
optional arguments:
  -h, --help  show this help message and exit
|
And that's how you'd foo a bar
```

有时候一些 parser 共享一些参数，相比于重复定义这些参数，一个单个的 parser 通过传递 `parents` 给 ArgumentParser。`parents=` 参数得到一个 ArgumentParser 对象的列表对象，从中收集所有的位置和选项行为

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

大多数的 parent parser 指定 `add_help=False`，因此 ArgumentParser 将看到两个帮助选项（一个在 parent 一个在 child）同时报错。你必须在通过 `parsers=` 传递前必须完全初始化

parser, 如果你在 child parser 改变 parent parsers, 改变将不被反映到 child.formatter_class ArgumentParser.durian 允许指定可用的格式化类自定义格式, 当前有 4 个类:

- argparse.RawDescriptionHelpFormatter
- argparse.RawTextHelpFormatter
- argparse.ArgumentDefaultHelpFormatter
- argparse.MetavarTypeHelpFormatter

RawDescriptionHelpFormatter 和 RawTextHelpFormatter 在如何显示说明上给与更多控制, 默认 ArgumentParser 对 description 和 epilog 在命令终端一行显示。

```

1 import argparse
2 parser = argparse.ArgumentParser(prog='PROG', description='''
3     description was indented wierd
4     but that is okey ''',
5     epilog=''''
6     likewise for this epilog whose whitespace will be
7     cleaned up and whose words will be wrapped
8     across a couple lines ''')
9 parser.print_help()

```

```

hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo/.py
usage: PROG [-h]

this description was indented wierd but that is okey
optional arguments:
  -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines

```

传递 RawDescriptionHelpFormatter 作为 formatter_class= 让 description 和 epilog 正确显示。RawTextHelpFormatter 主要维持素有的帮助文本, 值描述的信息。

ArgumentDefaultHelpFormatter: 自动增加关于值的默认信息。

```

1 import argparse
2 parser = argparse.ArgumentParser(prog='Prog',
3     formatter_class = argparse.ArgumentDefaultsHelpFormatter)
4 parser.add_argument('foo', type=int, default=42, help='FOO')
5 parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
6 parser.print_help()

```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo9.py
usage: Prog [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
  bar          BAR! (default: [1, 2, 3])

optional arguments:
  -h, --help    show this help message and exit
  --foo FOO    FOO (default: 42)
```

MatavarTypeHelpFormatter 用 type 显示参数显示值的名字。

```
1 import argparse
2 parser = argparse.ArgumentParser(prog='Prog',
3                                 formatter_class=argparse.ArgumentDefaultsHelpFormatter)
4 parser.add_argument('--foo', type=int, default=42, help='FOO')
5 parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
6 parser.print_help()
```

```
hpc@hpc-322:~/TensorFlow_Notebook$ python code/demo10.py
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help    show this help message and exit
  --foo int
```

prefix_chars, 大多数命令行参数选项用-, 比如-f/-foo。parsers 需要支持不同的或者说另外的前缀, 像 +f 或者/fo 可以设置 prefix_chars= 参数指定。prefix_chars 默认默认为-, 用非-字符能禁用-f/-foo 这种类型的选项。

```
1 import argparse
2 parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+-')
3 parser.add_argument('+f')
4 parser.add_argument('++bar')
5 parser.parse_args('+f X ++bar Y'.split())
```

fromfile_prefix_chars, 有时我们处理一个长的参数列表, 将参数保存在文件中比直接在命令行中更容易理解, 如果 fromfile_prefix_chars= 参数给 ArgumentParse 结构体, 指定的参数将被作为文件, 被下面的参数取代。例如

```
1 import argparse
2 with open('args.txt', 'w') as fp:
3     fp.write('-f\nbar')
4 parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
5 parser.add_argument('-f')
6 parser.parse_args(['-f', 'foo', '@args.txt'])
```

默认从一个文件读取参数, 上面的表达式 ['-f', 'foo', '@args.txt'] 等于表达式 ['-f', 'foo', '-f', 'bar'], fromfile_prefix_chars 参数默认为 None, 意味着参数不被当作文件。argument_default

通常通过传递 add_argument 或者通过调用 set_defaults() 方法指定名字和值对, 然而有时候通过给参数指定一个简单的 parser-wide 是有用的, 这可以通过传递 argument_default=关键字到 ArgumentParser, 例如调用其全局抑制属性在 parse_args() 调用, 我们用 argument_default=SUPPRESS:

```
1 import argparse
2 parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
3 parser.add_argument('--foo')
4 parser.add_argument('bar', nargs='?')
5 parser.parse_args(['--foo', '1', 'BAR'])
6 print(parser.parse_args([]))
```

allow_abbrev

通常我们传递一个参数 liebhiao 给 ArgumentParser 的方法 parse_args(), 如果选项参数太长的话。特征展示可能通过设置 allow_abbrev 设置为 False 被禁用。

```
1 import argparse
2 parser = argparse.ArgumentParser(prog='Prog', allow_abbrev=False)
3 parser.add_argument('--foobar', action='store_true')
4 parser.add_argument('--fooley', action='store_true')
5 parser.parse_args(['--foon'])
```

```
hpc@hpc-322:~/TensorFlow_Notebook/code$ python demo14.py
usage: Prog [-h] [--foobar] [--fooley]
Prog: error: unrecognized arguments: --foon
```

conflict_handler

ArgumentParser 对象不允许相同的选项字符串有两个行为, 默认情况下当已经一偶选项字符串使用时尝试穿件一个新的参数 ArgumentParser 对象将报出异常。

```
In [1]: import argparse
In [2]: parser = argparse.ArgumentParser(prog='PROG')
In [3]: parser.add_argument('-f', '--foo', help='old foo help')
Out[3]: _StoreAction(option_strings=['-f', '--foo'], dest='foo', nargs=None, const=None, default=None, type=None, choices=None, help='old foo help', metavar=None)
In [4]: parser.add_argument('--foo', help='new foo help')
      File "<ipython-input-4-b0dbd0131b6e>", line 1
          parser.add_argument('--foo', help='new foo help')
                                         ^
SyntaxError: invalid syntax
```

有时候覆

盖掉就得参数时有用的, 为了得到参数的行为值'resolvce' 可能被应用在 conflict_handler=参数。

```
1 import argparse
2 parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
3 parser.add_argument('-f', '--foo', help='old foo help')
4 parser.add_argument('--foo', help='new foo help')
```

```
5 parser.print_help()
```

```
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
  -h, --help  show this help message and exit
  -f FOO      old foo help
  --foo FOO   new foo help
```

如果所有的选项字符串被覆盖，ArgumentParser 对象仅仅移除一个行为，因此上面的例子中，就得行为-f/-foo 行为保留-f 行为，因为仅仅-foo 选项字符串被覆盖。add_help

默认情况下 ArgumentParserdurian 增加帮助信息到显示的消息中，例如：

```
1 import argparse
2 parser = argparse.ArgumentParser(description='Process some integers.')
3 parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer
                     for the accumulator')
4 parser.add_argument('--sum', dest='accumulate', action='store_const', const=sum,
                     default=max, help='sum the integers(
                     default:find the max)')
5 args = parser.parse_args()
6 print(args.accumulate(args.integers))
```

```
usage: demo1.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N          an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers(default:find the max)
```

```
1 import argparse
2 parser = argparse.ArgumentParser(description='Process some integers.', add_help=
                                 False)
3 parser.add_argument('integers', metavar='N', type=int, nargs='+', help='an integer
                     for the accumulator')
4 parser.add_argument('--sum', dest='accumulate', action='store_const', const=sum,
                     default=max, help='sum the integers(
                     default:find the max)')
5 args = parser.parse_args()
6 print(args.accumulate(args.integers))
```

```
usage: demo1.py [--sum] N [N ...]
demo1.py: error: the following arguments are required: N
```

6.1.3 add_argument() 方法

ArgumentParser.add_argument(name or flags..., action][, nargs][, const][, default][, type][, choices][, required][, help][, metavar][, dest]) 定一个一个命令行参数应该被如何解

析，每一个参数自己有自己的详细描述，如下：

- name or flags: 名字或者选项字符串，foo 或者 (-f,--foo)。
- action: 参数出现在命令行后采取的基本的行为。
- nargs: 命令行参数应该被使用的参数的数量。
- const:action 和 nargs 选项要求的常数值。
- default: 缺乏参数的默认值。
- type: 传递参数读取的数据类型。
- choices: 参数的允许值的容器。
- required: 是否命令行选项被忽略。
- help: 简易的参数说明。
- metavar: 在 usage 消息的名字。
- dest: 增加到 parse_args() 返回对象的属性的名字。

name 或者 flags

当 parse_args() 被调用的时候。选项参数通过-前缀识别。

```

1 import argparse
2 parser = argparse.ArgumentParser(prog='PROG')
3 parser.add_argument('-f', '--foo')
4 parser.add_argument('bar')
5 print(parser.parse_args(['BAR']))
6 print(parser.parse_args(['BAR', '--foo', 'FOO']))

```

```

hpc@hpc-322:~/TensorFlow_Notebook/code$ python demo16.py
Namespace(bar='BAR', foo=None)
Namespace(bar='BAR', foo='FOO')

```

action

- 'store': 仅仅保存参数的值，例如

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo')
3 parser.parse_args('--foo 1'.split())

```

输出 Namespace(foo='1')

- 'store_true': 存储 const 参数指定的值，'store_const' 行为通常用于指定一些 flag。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', action='store_const', const=42)
3 parser.add_argument('--foo')

```

输出:Namespace(foo=42)

- 'store_true' 和 'store_false' 指定 'store_const'。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', action='store_true')
3 parser.add_argument('--bar', action='store_false')
4 parser.add_argument('--baz', action='store_false')
5 parser.parse_args('--foo --bar'.split())

```

输出:Namespace(foo=True, bar=False, baz=True)

- 'append': 一个存储列表，添加每个参数值到列表中，允许选项被多次指定时很有用。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--str', dest='types', action='append_const', const=str)
3 parser.add_argument('--int', dest='types', action='append_const', const=int)
4 parser.parse_args('--str --int'.split())

```

输出:Namespace(type=[<class 'str'>, <class 'int'>])

- 'count': 关键参数出现的次数。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--verbose', '-v', action='count')
3 parser.parse_args(['-vvv'])

```

输出:Namespace(verbose=3)

- help: 打印当前 parser 所有选项的帮助信息，默认帮助行为被添加到 parser。
- version: add_argument 调用指定 version= 关键字

```

1 import argparse
2 parser = argparse.ArgumentParser(prog='PROG')
3 parser.add_argument('--version', action='version', version='(%prog) 2.0')
4 parser.parse_args(['--version'])

```

输出 PROG 2.0。

- 你可以通过传递行为子类或者其它对象的接口传递给 action，推荐的方法是扩展 Action，覆盖掉 `__call__` 方法和 `__init__`。

```

1 class FooAction(argparse.Action):
2     def __init__(self, option_strings, dest, nargs=None, **kwargs):
3         if nargs is not None:
4             raise ValueError("nargs not allowed")
5     def __call__(self, parser, namespace, values, option_string=None):
6         print('%r %r %r' % (namespace, values, option_string))
7         setattr(namespace, self.dest, values)
8     parser = argparse.ArgumentParser()
9     parser.add_argument('--foo', action=FooAction)
10    parser.add_argumentParser('bar', action=FooAction)
11    args = parser.parse_args('1 -- foo 2'.split())

```

输出：

Namespace(bar=None,foo=None) '1' None

Namespace(bar=1,foo=None) '2' '--foo'

nargs

- N: 一个整数，命令行下的参数被放到一起成为一个列表：

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', nargs=2)
3 parser.add_argument('bar', nargs=1)
4 parser.parse_args('c --foo a b'.split())

```

输出:Namespace(bar=['c'],foo=['a','b'])

- ?：根据不同情况生成不同的值，如果没有参数指定它的值来自默认生成如果有一个带有-前缀的参数值将被 const 参数生成，如果指定了值将生成指定值。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', nargs='?', const='c', default='d')
3 parser.add_argument('bar', nargs='?', default='d')
4 parser.parse_args(['XX', '--foo', 'YY'])
5 parser.parse_args(['XX', '--foo'])
6 parser.parse_args([])

```

分别输出：

Namespace(bar='XX',foo='YY')

Namespace(bar='XX',foo='x')

Namespace(bar='d',foo='d')

用 nargs='?' 更常用的用法时允许选项输入输出文件:

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('infile',nargs='?',type=argparse.FileType('r'),default
                     =sys.stdin)
3 parser.add_argument('outfile',nargs='?',type=argparse.FileType('w'),
                     default=sys.stdout)
4 parser.parse_args(['input.txt','output.txt'])

```

输出:Namespace(infile=<_io.TextIOWrapper name='input.txt',encoding='UTF-8'>,
outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>) parser.parse_args([])
输出: Namespace(infile=<io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)

- *: 所有的命令行参数将被放到一个列表中。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo',nargs='*')
3 parser.add_argument('--bar',nargs='*')
4 parser.add_argument('--barz',nargs='*')
5 parser.parse_args('a b --foo x y --bar 1 2'.split())

```

输出:Namespace(bar=['1','2'],baz=['a','b'],foo=['x','y'])

- +: 所有的命令行参数将被添加到一个列表中, 至少需要一个参数否则将报错。

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('foo',nargs='+')
3 parser.parse_args(['a','b'])
4 parser.parse_args([])

```

输出:Namespace(foo=['a',nargs='+'])

usage: PROG [-h] foo [foo ...]

PROG: error: too few arguments

- argparse.REMAINDER: 所有已经存在的参数被添加到一个列表。

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('--foo')
3 parser.add_argument('command')
4 parser.add_argument('args',nargs=argparse.REMAINDER)
5 print(parser.parse_args('--foo B cmd --arg1 xx zz'.split()))

```

输出:Namespace(args=['-arg1','XX','ZZ'],command='cmd',foo='B') 如果 nargs 参数没有提供, argument 由 action 决定, 通常这意味着一个的命令行参数被使用一个项目被产生。

const

const 参数被用在保存没有被命令行读入的常数来常数值, 两个常见的用法如下:

- 当 add_argument() 调用的时候设置了 action='store_const' 或者是 action='append_const' 通过增加 const 值到一个 parse_args() 返回的对象的属性。
- 当 add_argument() 通过选项字符串 (像-f 或者-foo) 和 nargs='?' , 这将穿件一个由 0 行或者一行参数跟着的选项, 当解析命令行时, 如果选项字符串遇到没有命令行参数的时候, 值 const 将被用来替代。'store_const' 和'append_const' 行为, const 关键字参数必须给定, 对于其它行为, 默认为 None。

default

所有的参数和一些位置的参数在命令行下可能被忽略, add_argument() 参数 default 的值默认为 None, 指定当没有参数时什么值被使用。没有指定选项字符串, default 的值将取代参数。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', default=42)
3 parser.parse_args(['--foo', '2'])
4 parser.parse_args([])
```

输出: Namespace(foo='2')

Namespace(foo=42)

如果默认值是一个字符串, parser 解析值就好象命令行参数一样, 类似的, parser 应用任何 type 转换参数, 如果在设置属性值前 Namespace 返回值, 否则 parser 用下面的值。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--length', default=42, type=int)
3 parser.add_argument('--width', default=10.5, type=float)
4 parser.parse_args()
```

输出: Namespace(length=10, width=10.5)

对于参数为'?' 或者'*', 命令行没有值的时候 default 值将被使用

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('foo', nargs='?', default=42)
3 parser.parse_args(['a'])
4 parser.parse_args([])
```

分别输出:

```
Namespace(foo='a')
```

```
Namespace(foo=42)
```

如果 default=argparse.SUPPRESS 如果没有命令行参数将导致没有属性被添加。

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', default=argparse.SUPPRESS)
3 parser.parse_args([])
4 parser.parse_args(['--foo', '1'])
```

分别输出:

```
Namespace()
```

```
Namespace(foo='1')
```

type

默认 ArgumentParser 对象读命令行参数为字符串，然而，经常命令行应该以另一种数据类型解析，像 float, int, add_argument() 的 type 关键字允许需要的类型检查和转换被执行，常用的内部数据类型和参数可以被作为 type 的值直接使用。

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('foo', type=int)
3 parser.add_argument('bar', type=open)
4 parser.parse_args('2 temp.txt'.split())
```

输出:Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8',foo=2) 为了能轻松的使用多种文件类型,argparse 模块提供了工厂 FileType,利用 mode=,bufsize=,encoding= 和 error= 参数，例如 FileType('w') 可以被用来创建一个可写的文件。

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('bar', type=argparse.FileType('w'))
3 parser.parse_args(['output'])
```

输出:Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8';>) type 能够调用一个字符串参数返回转换过值的参数

```
1 import math
2 import argparse
3 def perfect_square(string):
4     value = int(string)
5     sqrt = math.sqrt(value)
6     if sqrt != int(sqrt):
7         msg = '%r is not a perfect square' % string
8         raise argparse.ArgumentTypeError(msg)
9     return value
```

```

10 parser = argparse.ArgumentParser(prog='PROG')
11 parser.add_argument('foo', type=perfect_square)
12 print(parser.parse_args(['9']))
13 print(parser.parse_args(['7']))

```

输出: Namespace(foo=9)

usage: PROG [-h] foo

PROG: error: argument foo: '7' is not a perfect square

choise

choise 参数在检查值的范围时很方便。

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('foo', type=int, choices=range(5, 10))
3 parser.parse_args(['7'])
4 parser.parse_args(['11'])

```

分别输出:Namespace(foo=7)

usage: PROG [-h] 5,6,7,8,9

PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)

choise

一些命令行参数从一些限定值的中选定,可以通过传递 choice 关键字参数给 add_argument(),当命令行解析的时候,值将被检查如果不在可接受值范围内将显示错误消息。

```

1 parser = argparse.ArgumentParser(prog='game.py')
2 parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
3 parser.parse_args(['rock'])
4 parser.parse_args(['file'])

```

分别输出:

Namespace(move='rock')

usage: game.py [-h] rock,paper,scissors

game.py: error: argument move: invalid choice: 'fire' (choose from 'rock', 'paper', 'scissors')
choice 选项检查在转化数据类型后进行。

```

1 parser = argparse.ArgumentParser(prog='doors.py')
2 parser.add_argument('door', type=int, choices=range(1, 4))
3 print(parser.parse_args(['3']))
4 print(parser.parse_args(['4']))

```

分别输出:

Namespace(door=3)

usage: doors.py [-h] 1,2,3

doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)

任何支持 in 操作的对象都能被传递给 choise 作为值，因此 dict, set 对象都是常用的支待的对象。required

通常 argparse 模块假设 flag 像可以被省略的-f 和–bar,, 为了一个选项必需要设置 required=True。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', required=True)
3 parser.parse_args(['--foo', 'BAR'])
4 parser.parse_args([])

```

分别输出：

Namespace(foo='BAR')

usage: argparse.py [-h] [-foo FOO]

argparse.py: error: option -foo is required

正如上例，如果 parse_args() 的 required 被标记，如果不给值将报错。help

help 值包含一些简单的参数说明，当用户要求帮助的时候（通常用-h 或者–help）, help 描述信息将被展示

```

1 parser = argparse.ArgumentParser(prog='frobble')
2 parser.add_argument('--foo', action='store_true', help='foo the bars before
                     frobbing')
3 parser.add_argument('bar', nargs='+', help='foo the bars before frobbed')
4 parser.parse_args(['-h'])

```

输出：

usage: frobble [-h] [-foo] bar [bar ...]

positional arguments:

bar one of the bars to be frobbled

optional arguments:

-h, –help show this help message and exit

–foo foo the bars before frobbing

help 字符串能包含多种格式像程序名字或者默认参数, 可用的指定包含程序的名字,%(prog)s 和多数 add_argument() 关键字，像%(default)s,%(type)s 等等。

```

1 parser = argparse.ArgumentParser(prog='frobble')
2 parser.add_argument('bar', nargs='?', type=int, default=42, help='the bar to %(prog)
                     s(default:%(default)s)')
3 parser.print_help()

```

输出:

```
usage: frobble [-h] [bar]
```

optional arguments:

```
bar the bar to frobble (default: 42)
```

optional arguments:

```
-h, --help show this help message and exit
```

帮助字符串支持% 格式, 如果你想一个% 出现在帮助字符串中, 你需要使用%% argparse 对于指定的选项通过设置 argparse.SUPPRESS 设置支持静默帮助。

```
1 parser = argparse.ArgumentParser(prog='frobble')
2 parser.add_argument('--foo', help=argparse.SUPPRESS)
3 parser.print_help()
```

输出:

```
usage: frobble [-h]
```

optional arguments:

```
-h, --help show this help message and exit
```

metavar

当 ArgumentParser 生成帮助消息的时候需要一些方法设计查询每个参数, 默认, ArgumentParser 对象用 dest 值作为每个对象的名字, 默认对于 action 位置的参数, dest 值被直接使用, 对于一些选项行为, dest 值时大写的。因此单个位置参数 dest='bar' 将被认做 bar, --foo 应该被跟着一个命令作为 FOO

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo')
3 parser.add_argument('bar')
4 parser.parse_args('X --foo Y'.split())
5 print .print_help()
```

分别输出:

```
Namespace(bar='X', foo='Y')
```

```
usage: [-h] [-foo FOO] bar
```

optional arguments:

```
bar
```

optional arguments:

-h, --help show this help message and exit
 -foo FOO

一个可用的名字被 metavar 指定:

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', metavar='YYY')
3 parser.add_argument('bar', metavar='XXX')
4 parser.parse_args('X -- foo Y'.split())
5 parser.print_help()
```

Namespace(abr='X',foo='Y')
 usage: [-h] [-foo YYY] XXX

positional arguments:

XXX

optional arguments:

-h, --help show this help message and exit
 -foo YYY

注意 metavar 仅仅改变显示的名字, parse_args() 属性的名字仍然由 dest 值决定。不同的 nargs 也许导致 metavar 被多次使用, 提供一个元组给 metavar 指定一个不同的显示。

```

1 parser = argparse.ArgumentParser(prog='prog')
2 parser.add_argument('-x', nargs=2)
3 parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
4 parser.print_help()
```

输出:

usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:

-h, --help show this help message and exit
 -x X X
 --foo bar baz
 dest

大多数 ArgumentParser 行为增加一些值作为 parser_args() 返回值的属性。属性的名字由 dest 决定

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('bar')
3 parser.parse_args(['xxx'])

```

输出:Namespace(bar='xxx')

对于选项参数, dest 的值从选项字符串推断出, ArgumentParser 通过得到长的选项字符串删除初始化-字符串生成 dest 的值, 如果 meiyoiu 长的选项字符串提供, dest 将通过初始化字符-从第一个短的字符串选项得到。任何内部-字符将被转换为 _ 字符确保字符串是一个可用的属性名字。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('-f', '--foo-bar', '--foo')
3 parser.add_argument('-x', '-y')
4 parser.parse_args ['-f 1 -x 2'.split()]
5 parser.parse_args('--foo 1 -y 2'.split())

```

分别输出:

Namespace(foo_bar=1,x='2')

Namespace(foo_bar='1',x='2')

dest 允许自定义属性的名字:

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', dest='bar')
3 parser.parse_args('--foo XXX'.split())

```

输出:Namespace(bar='XXX')

Action class

Action classes 实现的 Action API,一个命令行返回的可调的 API。任何这个 API 对象都可以被 zuoweiaction 参数传递给 add_argument()。class argparse.Action(option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None) Action 实力应该是可调用的, 因此子类必须被 __call__ 方法覆盖, 应该接受四个参数:

- parser: 包含这个 action 的 ArgumentParser。
- namespace:parser_args() 返回的 Namespace 对象, 大多数行为通过 setattr() 增加一个属性到对象。
- varlue: 结合命令行参数和任何转化应用, 类型转换被 type 关键字指定。
- option_string: 宣告像字符串被用于激活这个 action, option_string 时一个选项, 将

缺席如果这个 action 和 positional 参数结合。`__call__` 方法也许执行任意行为，但是典型的设置基于 dest 和 value 的 namespace 属性。

`parse_args()` 方法:

`ArgumentParser.parse_args(args=None, namespace=None)` 转换参数字符串为对象指定他们作为 namespace 的属性。之前调用 `add_argument()` 决定 决定创建什么对象如何复制， 默认 argument 字符串来自 `sys.argv`, 一个新的空的 Namespace 对象被创建。Option value syntax

`parse_args` 方法支持多种方法指定选项的值，在最简单的情况下，这个选项和它的值被传递作为两个分开的参数:

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('-x')
3 parser.add_argument('--foo')
4 parser.parse_argument('-x', 'X')
5 parser.parse_args('--foo', 'FOO')
```

分别输出:

`Namespace(foo=None,x='X')`

`Namespace(foo='FOO',x=None)`

对于短的选项，这个选项值可以被链接，多个短选项可以被-前缀连接在一起，只要最新的选项（非空）要求值:

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('-x', action='store_true')
3 parser.add_argument('-y', action='store_true')
4 parser.add_argument('-z')
5 parser.parse_args(['-xyzZ'])
```

输出:`Namespace(x=True,y=True,z='Z')` 不可用的参数

当解析命令行时 `parse_args()` 检查多种错误，包括不明确的选项，不可用的类型， 错误的参数为值等等，当出现一个错误，它推出同时打印错误和用法信息。

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('--foo', type=int)
3 parser.add_argument('bar', nargs='?')
4
5 # invalid type
6 parser.parse_args(['--foo', 'spam'])
7 usage: PROG [-h] [--foo FOO] [bar]
8 PROG: error: argument --foo: invalid int value: 'spam'
9
10 # invalid option
```

```

11 parser.parse_args(['--bar'])
12 usage: PROG [-h] [--foo FOO] [bar]
13 PROG: error: no such option: --bar
14
15 # wrong number of arguments
16 parser.parse_args(['spam', 'badger'])
17 usage: PROG [-h] [--foo FOO] [bar]
18 PROG: error: extra arguments found: badger

```

参数包含

当用户犯错时 `parse_args()` 方法尝试给出错误, 但是一些情况下固有的二义, 例如, 命令行参数-1 可能同时指定一个选项或者尝试提供一个指定位置参数, `parse_args()` 方法导致, 指定位置的参数仅仅用-开始如果他们看起来像负数在 `parser` 没有选像解析看起来像负数:

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('-x')
3 parser.add_argument('foo', nargs='?')
4
5 # no negative number options, so -1 is a positional argument
6 parser.parse_args(['-x', '-1'])
7 Namespace(foo=None, x='-1')
8
9 # no negative number options, so -1 and -5 are positional arguments
10 parser.parse_args(['-x', '-1', '-5'])
11 Namespace(foo='-5', x='-1')
12
13 parser = argparse.ArgumentParser(prog='PROG')
14 parser.add_argument('-1', dest='one')
15 parser.add_argument('foo', nargs='?')
16
17 # negative number options present, so -1 is an option
18 parser.parse_args(['-1', 'X'])
19 Namespace(foo=None, one='X')
20
21 # negative number options present, so -2 is an option
22 parser.parse_args(['-2'])
23 usage: PROG [-h] [-1 ONE] [foo]
24 PROG: error: no such option: -2
25
26 # negative number options present, so both -1s are options
27 parser.parse_args(['-1', '-1'])
28 usage: PROG [-h] [-1 ONE] [foo]

```

```
29 PROG: error: argument -1: expected one argument
```

如果你有一个必须以-开始的参数而且不是负数，你可以插入'-'告诉 parse_args() 之后的一切：

```
1 parser.parse_args(['--', '-f'])
```

输出:Namespace(foo='f',one=None) 参数缩略 如果缩略没有歧义 parser_args() 方法默认允许长选项被简写为前缀。

```
1 parser = argparse.ArgumentParser(prog='PROG')
2 parser.add_argument('-bacon')
3 parser.add_argument('-badger')
4 parser.parse_args('-bac MMM'.split())
5 Namespace(bacon='MMM', badger=None)
6 parser.parse_args('-bad WOOD'.split())
7 Namespace(bacon=None, badger='WOOD')
8 parser.parse_args('-ba BA'.split())
9 usage: PROG [-h] [-bacon BACON] [-badger BADGER]
10 PROG: error: ambiguous option: -ba could match -badger, -bacon
```

可能产生多个选项时错误产生，可以通过设置 allow_abbrev 设置为 False 禁用。Beyond sys.argv

ArgumentParser 通常比 sys.argv 有用，可以穿地一个字符串列表到 parser_args() 完成，这在测试交互式提示符很有用。

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('integers', metavar='int', type=int, choices=range(100),
3 nargs='+', help='an integer in range 0..9')
4 parser.add_argument('--sum', dest='accumulate', action='store_const',
5 const=sum, default=max, help='sum the integers (default: find the max)')
6 parser.parse_args(['1','2','3','4'])
7 parser.parse_args(['1','2','3','4'], '--sum')
```

输出结果分别为：

Namespace(accumulate=<built-in function max>,integers=[1,2,3,4])

Namespace(accumulate=<built-in function sum>,integers=[1,2,3,4])

Namespace 对象

class argparse.Namespace，简单的 parse_args() 创建一个对象，保存属性返回它。这个类很简单，仅仅是一个可读表达的对象子类，如果你希望有字典类似的属性，你可以用标准的 python idiom()：

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo')
```

```

3 args = parser.parse_args(['--foo', 'BAR'])
4 var(args)

```

输出'foo':'BAR'

当 ArgumentParser 指定属性到已经存在的对象时它是很有用的，相比于新的 Namespaced 对象，它可以指定 namespace= 关键参数获得。

```

1 class C:
2     pass
3 C = C()
4 parser = argparse.ArgumentParser()
5 parser.add_argument('--foo')
6 parser.parse_args(args=['--foo', 'BAR'], namespace=C)
7 c.foo

```

输出:'BAR'

子命令: ArgumentParser.add_subparsers([title][, description][, prog][, parser_class][, action][, option_string][, dest][, help][, metavar]) 一些程序分割他们的功能为一个子命令，例如，svn 程序可以有子命令 svn checkout,svn commit,svn update。当程序有一些要求不同类型命令行参数的不同的功能的时候分割功能的方法是一个好的想法，ArgumentParser 支持支持 add_subparsers 一个子命令，add_subparsers() 方法调用通常没有参数返回一个特殊的行为对象，这个对象是一个方法，add_parser() 得到一个命令名字和任何 ArgumentParser 能够草体参数，返回一个可以被修改的 ArgumentParser 对象。

- title: 帮助输出 sub-parser 组的标题，如果说明提供了的话默认”subcommands”，否则用参数作为标题。
- description: 在输出帮助中描述 sub-parser 组， 默认是 None。
- prog:sub-command 的帮助信息， 默认程序的名字和位置上的参数在 subparser 参数前。
- parser_class: 用于创建一个 sub-parser 实例的类， 默认时当 parser。
- action: 在命令行中参数出现厚的基础类型的行为。
- dest:sub-command 下属性的名字将被存储， 默认没有值被存储。
- help: 在帮助输出的 sub-parser, 默认为 None。
- metavar: 在 help 中可用的子命令默认是 None 代表子命令 cmd1,cmd2,...

用法:

```

1 # create the top-level parser
2 parser = argparse.ArgumentParser(prog='PROG')
3 parser.add_argument('--foo', action='store_true', help='foo help')
4 subparsers = parser.add_subparsers(help='sub-command help')
5
6 # create the parser for the "a" command
7 parser_a = subparsers.add_parser('a', help='a help')
8 parser_a.add_argument('bar', type=int, help='bar help')
9
10 # create the parser for the "b" command
11 parser_b = subparsers.add_parser('b', help='b help')
12 parser_b.add_argument('--baz', choices='XYZ', help='baz help')
13
14 # parse some argument lists
15 parser.parse_args(['a', '12'])
16 Namespace(bar=12, foo=False)
17 parser.parse_args(['--foo', 'b', '--baz', 'Z'])
18 Namespace(baz='Z', foo=True)

```

注意 parser_args() 返回的 durian 将包含住 parser 和 subparser 命令行选中的参数，因此在上面的例子中，当一个命令被指定，仅仅 foo 和 bar 被呈现，当 b 被指定，仅仅 foo 和 baz 属性被呈现，类似的，subparser 要求帮助信息，仅仅这个 parser 的帮助信息被打印，帮助信息不包含父或者兄弟 parser 信息。（一个 subparser 命令的帮助消息，然而，可以被 help= 参数增加到上面）

```

1 parser.parse_args(['--help'])
2 usage: PROG [-h] [--foo] {a,b} ...
3
4 positional arguments:
5   {a,b}    sub-command help
6     a      a help
7     b      b help
8
9 optional arguments:
10  -h, --help  show this help message and exit
11  --foo    foo help
12
13 parser.parse_args(['a', '--help'])
14 usage: PROG a [-h] bar
15
16 positional arguments:
17   bar      bar help
18

```

```

19 optional arguments:
20   -h, --help    show this help message and exit
21
22 parser.parse_args(['b', '--help'])
23 usage: PROG b [-h] [--baz {X,Y,Z}]
24
25 optional arguments:
26   -h, --help      show this help message and exit
27   --baz {X,Y,Z}  baz help

```

add_subparsers() 方法也支持 title 和 description 关键参数, 当两者都呈现的时候在帮助输出 subparser 的命令将出现在自己的组。

```

1 >>> parser = argparse.ArgumentParser()
2 >>> subparsers = parser.add_subparsers(title='subcommands',
3 ...                                     description='valid subcommands',
4 ...                                     help='additional help')
5 >>> subparsers.add_parser('foo')
6 >>> subparsers.add_parser('bar')
7 >>> parser.parse_args(['-h'])
8 usage: [-h] {foo,bar} ...
9
10 optional arguments:
11   -h, --help    show this help message and exit
12
13 subcommands:
14   valid subcommands
15
16   {foo,bar}    additional help

```

更进一步, add_parser 支持一个 aliases 参数, 允许多字符串访问同一个 subparser, 像 svm, 别名 co 作为 checkout 的简写。

```

1 >>> parser = argparse.ArgumentParser()
2 >>> subparsers = parser.add_subparsers()
3 >>> checkout = subparsers.add_parser('checkout', aliases=['co'])
4 >>> checkout.add_argument('foo')
5 >>> parser.parse_args(['co', 'bar'])
6 Namespace(foo='bar')

```

一个类似的高效处理 sub-commands 结合 add_subparsers() 方法调用 set_default() 以至于每个 subparser 知道那个 python 函数应该被执行。

```

1 >>> # sub-command functions
2 >>> def foo(args):

```

```

3 ...     print(args.x * args.y)
4 ...
5 >>> def bar(args):
6 ...     print('((%s))' % args.z)
7 ...
8 >>> # create the top-level parser
9 >>> parser = argparse.ArgumentParser()
10 >>> subparsers = parser.add_subparsers()
11 >>>
12 >>> # create the parser for the "foo" command
13 >>> parser_foo = subparsers.add_parser('foo')
14 >>> parser_foo.add_argument('-x', type=int, default=1)
15 >>> parser_foo.add_argument('y', type=float)
16 >>> parser_foo.set_defaults(func=foo)
17 >>>
18 >>> # create the parser for the "bar" command
19 >>> parser_bar = subparsers.add_parser('bar')
20 >>> parser_bar.add_argument('z')
21 >>> parser_bar.set_defaults(func=bar)
22 >>>
23 >>> # parse the args and call whatever function was selected
24 >>> args = parser.parse_args('foo 1 -x 2'.split())
25 >>> args.func(args)
26 2.0
27 >>>
28 >>> # parse the args and call whatever function was selected
29 >>> args = parser.parse_args('bar XYZYX'.split())
30 >>> args.func(args)
31 ((XYZYX))

```

你可以用 `parse_args()` 在参数解析完成后通过调用合适的函数做这个工作，结合函数和 `action` 像这个像这样典型的轻松的方法处理不同的行为，然而，如果它需要检查 `subparser` 的名字，`dest` 关键值通过 `add_subparsers()` 调用将发挥作用。

```

1 >>> parser = argparse.ArgumentParser()
2 >>> subparsers = parser.add_subparsers(dest='subparser_name')
3 >>> subparser1 = subparsers.add_parser('1')
4 >>> subparser1.add_argument('-x')
5 >>> subparser2 = subparsers.add_parser('2')
6 >>> subparser2.add_argument('y')
7 >>> parser.parse_args(['2', 'frobble'])
8 Namespace(subparser_name='2', y='frobble')

```

FileType 对象：

class argparse.FileType(mode='r', bufsize=-1, encoding=None, errors=None) FileType 工厂创建一个能被传递给 ArgumentParser.add_argument() 的对象。参数有 FileType 对象将用要求的模式打开命令行参数作为文件，换从大小，编码，错误处理。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--raw', type=argparse.FileType('wb', 0))
3 parser.add_argument('out', mtype=argparse.FileType('w', encoding='UTF-8'))
4 parser.parse_args(['--raw', 'raw.dat', 'file.txt'])

```

输出: Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8', raw=<_ioFileIO name='raw.dat' mode='wb'>)

FileType 对象明白伪参数同时自动转换 sys.stdin 为可读的 FileType 对象，sys.stdout 可写的 FileType 对象。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('infile', type=argparse.FileType('r'))
3 parser.parse_args(['-'])

```

输出 Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8') Argument group

ArgumentParser.add_argument_group(title=None, description=None)

默认情况下，ArgumentParser groups，当显示帮助信息的时候命令行参数进入对应位置的参数和选项参数。当有一个比默认更好的概念上的参数组，合适的组能被 add_argument_group() 创建：

```

1 parser = argparse.ArgumentParser(prog='PROG', add_help=False)
2 group = parser.add_argument_group('group')
3 group.add_argument('--foo', help='foo help')
4 group.add_argument('bar', help='bar help')
5 parser.print_help()

```

输出: usage: PROG [-foo FOO] bar

group:

bar bar help

-foo FOO foo help

add_argument_group() 方法返回一个有 add_argument() 方法的参数组对象。当一个参数增加到组中，parser 就当它为正常参数，但是在帮助信息中分组显示。add_argument_group() 方法接受 title 和 description 参数自定义显示：

```

1 parser = argparse.ArgumentParser(prog='PROG', add_help=False)
2 group1 = parser.add_argument_group('group1', 'group1 description')

```

```

3 group1.add_argument('foo', help='foo help')
4 group2 = parser.add_mutually_exclusive_group(required=True)
5 group2.add_argument('--bar', help='bar help')
6 parser.print_help()

```

usage: PROG [-bar BAR] foo

group1:
group1 description

foo foo help

group2:
group2 description

--bar BAR bar help

注意任何不再你的用户定义组中的参数将以对应位置参数和选项参数结束。Mutual exclusion

`ArgumentParser.add_mutually_exclusive_group(required=False)`

创建一个转悠的组，`argparse` 将确保唯一的参数在彼此的组被呈现在命令行。

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 group = parser.add_mutually_exclusive_group()
3 group.add_argument('--foo', action='store_true')
4 group.add_argument('--bar', action='store_false')
5 parser.parse_args(['--foo'])
6 parser.parse_args(['--bar'])
7 parser.parse_args(['--foo', '--bar'])

```

分别输出：

`Namespace(bar=True,foo=True)`

`Namespace(bar=False,foo=False)`

`sage: PROG [-h] [-foo | --bar]`

`PROG: error: argument --bar: not allowed with argument -foo`

`add_mutually_exclusive_group()` 方法接受一个 `required` 参数，预示着最新的参数被要求。

```

1 parser = argparse.ArgumentParser(prog='PROG')
2 group = parser.add_mutually_exclusive_group(required=True)
3 group.add_argument('--foo', action='store_true')
4 group.add_argument('--bar', action='store_true')

```

```
5 parser.parse_args([])
```

输出:

```
usage: PROG [-h] (-foo | -bar)
```

```
PROG: error: one of the arguments -foo -bar is required
```

注意当前的 mutually exclusive 参数组不支持 title 和 description 参数。Parser defaults

```
ArgumentParser.set_defaults(**kwargs)
```

大多数时候,parse_args() 返回的属性对象将被命令行参数和参数行为完全决定。set_default() 允许一些额外的属性决定没有命令行增加时的行为:

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('foo', type=int)
3 parser.set_default(bar=42, baz='badger')
4 parser.parse_args(['736'])
```

输出:Namespace(bar=42,baz='badger',fpp=736) 注意 parser 级默认覆盖参数级。

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', default='bar')
3 parser.set_defaults(foo='spam')
4 parser.parse_args([])
```

输出: Namespace(foo='spam') Parser 级别在多个 parser 时特别有用。ArgumentParser.get_default(dest) 得到 namespace 属性的默认值, 正如设置 add_argument() 或者 set_defaults()

```
1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', default='badger')
3 parser.get_default('foo')
```

输出:'baadger' Printing help

在一些典型的应用中 parse_args() 将考虑打印用法和错误信息的格式, 然而一些格式方法是可用的: ArgumentParser.print_usage(file=None): 打印 ArgumentParser 应该在命令行调用的简单描述, 如果 file 是 None, sys.stdout 被假定。ArgumentParser.print_help(file=None): 打印程序的用法信息和 ArgumentParser 参数注册信息, 如果 file 为 None, sys.stdout 被假定。ArgumentParser.format_usage(): 返回在命令行中 ArgumentParser 参数应该被如何调用的简要说明字符串。ArgumentParser.format_help(): 返回一个包含程序用法和 ArgumentParser 参数注册信息的帮助字符串。Partial parsing

```
ArgumentParser.parse_known_args(args=None, namespace=None)
```

有时候一些脚本也许仅仅解析一些命令行参数, 传递参数到另一个脚本或者程序, 在这种情形下, parser._known_args() 方法很有用, 它像 parser_args() 除了当有额外的参数呈现的时候不生成错误, 相反, 它返回一个包含 populated namespace 和保留参数字符串的列表的两个元素的元组。

```

1 parser = argparse.ArgumentParser()
2 parser.add_argument('--foo', action='store_true')
3 parser.add_argument('bar')
4 parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])

```

输出:(Namespace(bar='BAR', foo=True), ['--badger', 'spam']) Customizing file parsing

ArgumentParser.convert_arg_line_to_args(arg_line)

从文件中读入的参数一行读一个, convert_arg_line_to_args() 能被覆盖。这个方法从参数文件得到一个简单的 arg_line 字符串, 返回一个参数列表, 每读取一行方法被调用一次。一个有用的覆盖每这个方法是当空格分开的 word 为参数, 下面的例子展示:

```

1 class NyArgumentParser(argparse.ArgumentParser):
2     def convert_tag_line_to_args(self, arg_line):
3         return arg_line.split()

```

Exiting method

ArgumentParser.exit(status=0, message=None): 这个方法终止程序, 以指定的状态推出, 如果参数被给, 打印消息。ArgumentParser.error(message): 这个方法打印包含消息用法信息到标准错误终止程序以状态代码 2。Upgrading optparse code

最初 argparse 模块尝试用 optparse 维持兼容性, 然而 optparse 很难扩展, 特别是改变要求支持新的 nargs= 指定更好的用法消息。当大多数 optparse 已经被复制粘贴过或者 monkey-patched, 它不再尝试维持向后兼容。, argparse 模块在一些方法改进了标准库 optparse:

- 处理位置参数。
- 支持子命令。
- 允许 + 和/前缀。
- 处理 0 或者更多 1 或者更多风格的参数。
- 处理更多的用法消息。
- 提供简单的接口自定义 type 和 action。

optparse 到 argparse 的并行升级

```

1 \item 用 ArgumentParser.add_argument() 调用取代 optparse.OptionParser.add_option()
      调用。
2 \item 用 args=parser.parser_args() 取代 (options, args)=parser.parse_args() 增加
      ArgumentParser.add_argument() 调用给指定
      位置的参数, 记住显现的前向, 现在在
      argparse 上下文称为 args。

```

```

3 \item 用 type 和 action 取代 callback 行为和 callback\_* 关键参数。
4 \item 取代 type 关键字的字符串名字和相关的对象类型（如 int, float, complex 等等）
5 \item 用 Namespace 和 optparse.OptionError, optparse.OptionValueError 取代 optparse.
      Value。
6 \item 用标准那得 Python 语法取代 \%default 或者 \%( default )s 和 \%( prog )s。
7 \item 通过调用 parser.add\_argument( '--version', action='version', version='<the
      version>' ) 取代 OptionParser 结构体 version
      。

```

setting 输入:

```

hpc@hpc322:~/文档/Tensorflow$ python code/arg1.py
usage: arg1.py [-h] echo
arg1.py: error: the following arguments are required: echo
hpc@hpc322:~/文档/Tensorflow$ python code/arg1.py -h
usage: arg1.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
hpc@hpc322:~/文档/Tensorflow$ 

```

add_argument 方法指定程序需要接受的命令参数，本例中为 echo，此程序运行必须指定一个参数，方法 parse_args() 通过分析指定的参数返回数据 echo。

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("echo", help="show the help information", type=int)
4 args = parser.parse_args()
5 print(args.echo**2)

```

指定参数类型为 int，默认为 string。

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("--verbosity", help="increase output verbosity")
4 args = parser.parse_args()
5 if args.verbosity:
6     print("Verbosity turned on")

```

```

Verbosity turned on
hpc@hpc322:~/文档/Tensorflow$ python code/arg3.py --verbosity a
Verbosity turned on

```

这里指定了--verbosity 程序就显示一些信息，如果不指定程序也不会出错，对应的变量就被设置为 None。

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("--verbosity", help="increase output verbosity", action="store_true")

```

```

4 args = parser.parse_args()
5 if args.verbose:
6     print("Verbosity turned on")

```

指定一个新的关键词 action, 赋值为 store_true。如果指定了可选参数, args.verbose 就赋值为 True, 否则就为 False。

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("-v", "--verbose", help="Increase output verbosity", action="store_true")
4 args = parser.parse_args()
5 if args.verbose:
6     print("verbosity turned on")

```

```

hpc@hpc322:~/文档/Tensorflow$ python code/arg4.py --help
usage: arg4.py [-h] [-v]

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Increase output verbosity

```

```

1 #args5.py
2 import argparse
3 parser = argparse.ArgumentParser()
4 parser.add_argument("square", type=int, help="display help information")
5 parser.add_argument("-v", "--verbose", action="store_true", help="increase output verbosity")
6 args = parser.parse_args()
7 answer = args.square**2
8 if args.verbose:
9     print("The square of {} equals {}".format(args.square, answer))
10 else:
11     print(answer)

```

输入参数--verbose 和整数 (4) 顺序不影响结果。python args5.py -verbose 4 和 python args5.py 4 -verbose

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("square", type=int, help="display a square of a given number")
4 parser.add_argument("-v", "--verbose", type=int, help="increase output verbosity")
5 args = parser.parse_args()
6 answer = args.square**2
7 if args.verbose == 2:

```

```

8     print("The square of {} equals {}".format(args.square, answer))
9 elif args.verbosity == 1:
10    print("{}^2=={}".format(args.square, answer))
11 else:
12    print(answer)

```

python args6.py 4 -v 0,1,2 通过指定不同的参数 v 为 0,1,2 得到不同的结果。

```

1 #arg7.py
2 import argparse
3 parser = argparse.ArgumentParser()
4 parser.add_argument("square", type=int, help="display the square of a given number")
5 parser.add_argument("-v", "--verbosity", action="count", help="increase output verbosity")
6 args = parser.parse_args()
7 answer = args.square**2
8 if args.verbosity == 2:
9     print("The square of {} equals {}".format(args.square, answer))
10 elif args.verbosity == 1:
11     print("{}^2 == {}".format(args.square, answer))
12 else:
13     print(answer)

```

这里添加参数 action="count", 统计可选参数出现的次数。python arg7.py 4 -v(出现一次), 对应结果为 $x^2 == 16$

python arg7.py 4 -vv(出现两次), 对应出现 The square of 4 equals 16

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("square", type=int, help="display a square of a given number")
4 parser.add_argument("-v", "--verbosity", action="count", default=0, help="increase output verbosity")
5 args = parser.parse_args()
6 answer = args.square**2
7 if args.verbosity>=2:
8     print("The square of {} equals {}".format(args.square, answer))
9 elif args.verbosity>=1:
10    print("{}^2 == {}".format(args.square, answer))
11 else:
12    print(answer)

```

加速让 default 参数。这只默认为值 0, 当参数 v 不指定时参数就被置为 None, None 不能

和整型比较。

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 parser.add_argument("x", type=int, help="The base")
4 parser.add_argument("y", type=int, help="The exponent")
5 parser.add_argument("-v", "--verbosity", action="count", default=0)
6 args = parser.parse_args()
7 answer = args.x**args.y
8 if args.verbosity >= 2:
9     print("{} to the power {} equals {}".format(args.x, args.y, answer))
10 elif args.verbosity >= 1:
11     print("{}^{} == {}".format(args.x, args.y, answer))
12 else:
13     print(answer)

```

为了让后面的参数不冲突，我们需要使用另一个方法：

```

#args10.py
1 import argparse
2 parser = argparse.ArgumentParser()
3 group = parser.add_mutually_exclusive_group()
4 parser.add_argument("-v", "--verbose", action="store_true")
5 group.add_argument("-q", "--quit", action="store_true")
6 parser.add_argument("x", type=int, help="The base")
7 parser.add_argument("y", type=int, help="The exponent")
8 args = parser.parse_args()
9 answer = args.x**args.y
10 if args.quit:
11     print(answer)
12 elif args.verbose:
13     print("{} to the power {} equals {}".format(args.x, args.y, answer))
14 else:
15     print("{}^{} == {}".format(args.x, args.y, answer))

```

可以输入 `python arg10.py 3 4 -vq` 得到计算结果。

```

1 import argparse
2 parser = argparse.ArgumentParser()
3 group = parser.add_mutually_exclusive_group()
4 group.add_argument("-v", "--verbose", action="store_true")
5 group.add_argument("-q", "--quit", action="store_true")
6 parser.add_argument("x", type=int, help="The base")
7 parser.add_argument("y", type=int, help="The exponent")
8 args = parser.parse_args()
9 answer = args.x**args.y

```

```
10 if args.quit:  
11     print(answer)  
12 elif args.verbose:  
13     print("{} to the power {} equals {}".format(args.x, args.y, answer))  
14 else:  
15     print("{}^{} == {}".format(args.x, args.y, answer))
```

这里参数 v 和 q 不能同时使用。

6.2 path

6.2.1 函数说明

- `os.path.abspath(path)`: 返回 path 的绝对路径, 在多数平台下, 相当于调用函数 `normpath(join(os.getcwd(),path))`
- `os.path.basename(path)`: 返回 path 的路径 base name, 第二个元素通过传递 path 给 `split()`, 注意这个结果不同于 unix 的 basename 程序, 这里 basename,'foo/bar' 然会 bar, 而 basename() 函数返回空字符串 ("")。
- `os.path.commonpath(paths)`: 返回 paths 队列中最长的 sub-path, 日国路径中包含绝对路径和相对路径的话将报 ValueError 或者如果 paths 是空, 不想 commonprefix(), 这个函数返回一个错的路径。
- `os.path.dirname(path)`: 返回目录的名字, 就是 path 用 split 分割厚的第一个元素。
- `os.path.exists(path)`: 如果春在路径 path 或者一个打开的文件描述返回 True。对于破掉的符号链接返回 False, 在一些平台, 如果权限不允许执行 `os.stat()` 即使存在物理路径这个函数也返回 False。
- `os.path.lexists(path)`: 如果路径存在返回 True, 对 broken 符号链接返回 True, 等效与 exists()。
- `os.path.expanduser(path)`: 在 Unix 和 Windows 上用 ~ 或者 user 取代用户路径的值。在 unix 上一个 被环境变量 HOME 替代 (如果设置了 HOME 环境变量的话), 否则当前用户的 home 目录通过内建模块 pwd 查找, 一个初始化 user 是寻找在 password 目录里面的目录。
- `os.path.expandvars(path)`: 返回环境变量的值, 子字符串形式时 `namename` 被环境变量名取代, 变形的变量名字和参考不存在的变量将不改变。
- `os.path.getatime(path)`: 返回上次访问路径的时间, 返回一个从 epoch 起经历的秒数, 如果文件不存在或不可访问则报 OSError。
- `os.path.getmtime(path)`: 返回最新修改路径的时间, 返回值时一个 epoch 其开始的秒数, 文件不存在或者不可范围跟时报 OSError。
- `os.path.getctime`: 返回系统的 ctime, 在 Unix 上时最新的 metadata 改变的时间, 在 windows 上时 path 创建的时间, 返回一个从 epoch 起经历的秒数, 如果文件不存在或不可访问则报 OSError。

- `os.path.getsize(path)`: 返回字节表示的路径的大小, 如果不存在文件或者文件不可范
围跟将报出 `OSError`。
- `os.path.isabs(path)`: 如果路径是绝对路径返回 `True`。
- `os.path.isfile(path)`: 如果路径是文件将返回 `True`。
- `os.path.isdir(path)`: 如果存在路径返回 `True`。
- `os.path.islink(path)`: 如果路径查询一个目录入口时符号链接返回 `True`, 如果 Python
运行时符号链接不支持将返回 `False`。
- `os.path.ismount(path)`: 如果 `path` 是一个挂载点, 返回 `True`。
- `os.path.join(path,*paths)`: 加入一个或者更多的组建, 返回值是连接路径和任何成员
的路径。
- `os.path.normcase(path)`: 在 Unix, MAX OS 上返回路径不变, 在一些敏感的文件系
统上将转换路径为小写, 在 windows 上将转化斜线为反斜线, 如果 `path` 不是 `str` 或者
`bytes` 将报 `TypeError`。
- `os.path.normpath(path)`: 删减冗余得分和服, 因此 `A//B,A/B,A./B,A/foo../B` 将变
为 `A/B`. 字符串操作也许改变包含符号链接的意义, 在 windows 上它转化斜线为反斜
线。
- `os.path.realpath(path)`: 返回指定文件名的确定路径, 消除路径中出现的任何符号链
接。
- `os.realpath(path,start=os.curdir)`: 从当前路径或者 `start` 路径返回相对的文件路径,
这是一个路径计算: 文件系统不妨问确定的存在的或者自然的路径或者 `start`。
- `os.path.samefile(path1,path2)`: 如果 `pathname` 值访问相同的文件或者目录则返回 `True`,
这有 device 名字和 i-node 数量决定, 如果 `os.stat()` 调用 `pathname` 失败将报出异常。
- `os.path.sameopenfile(fp1,fp2)`: 如果 `fp1` 和 `fp2` 指定的时相同的文件将返回 `True`。
- `os.path.samestat(stat1,stat2)`: 如果元组 `stat1` 和 `state2` 查询的时相同的文件, 返回
`True`, 这个结构可需已经被 `os.fstate()`, `os.lstat()` 或者 `os.stat()` 返回, 番薯通过 `same-
file()` 和 `sameopenfile()` 实现基本的比较。
- `os.path.split(path)`: 分割路径为 `(head,tail)`。 `tail` 不包含斜线, 如果以斜线将诶为, `tail`
将为空, 如果没有斜线, 头将为空, 如果 `path` 时空, 头尾都为空。后面的斜线从 `head`

删除出位它是 root(一个或者更多的斜线), 在所有的情况下 join(head,tail) 返回一个路径到相同位置作为路径。

- os.path.splitdrive(path): 返回 pathname 到 (drive,tail), 这里 drive 可以使挂载点或者空字符串。在系统上没有用驱动器指定, 驱动器将为空字符串, 在所有的倾向下, drive+tail 将时相同的路径。在 Windows 上, 分割 pathname 成 drive/UNC 共享点和相对路径, 如果路径包含驱动器驱动器将包含冒号 (splitdrive("c:/dir")) 返回 ("c:", "/dir"), 如果路径包含驱动 UNC 路径, 驱动器将包含主机名和 share, 但是不包含四个分隔符 splitdrive("//host/computer/dir")return("//host/computer","/dir")
- os.path.split(path): 分割路径名为 (root,ext) 像 root+ext == path, ext 时空或者以一个周期开头, 导致 basename 被忽略, splitext('.cshrc') 返回 ('.cshrc', '')
- os.path.supports_unicode_filenames(): 如果文件名时 unicode 编码的则为 True。

6.2.2 例子

1. 获取文件名, 目录, 扩展, 新文路径。

```

1 import os
2 file_path = '~/iris_test.csv'
3 filename = os.path.basename(file_path)
4 new_dir = os.path.join('home', 'hpc', filename)
5 file_dir = os.path.dirname(file_path)
6 dir1 = '~/'
7 fulldir = os.path.expanduser(dir1)
8 sp = os.path.split(new_dir)
9 print('new_dir:', sp[0], 'ext:', sp[1])

```

2. 查看文件同时打开文件

```

1 import os
2 path = '/etc'
3 filename = 'passwd'
4 if os.path.isdir(path):
5     full_path = os.path.join(path, filename)
6     if os.path.isfile(full_path):
7         with open(full_path, 'r') as f:
8             line = f.readlines()
9             for _ in range(len(line)):
10                 print(line)

```

3. 获取文件大小和修改时间

```

1 os.path.getsize('/etc/passwd')
2 os.path.gettime('/etc/passwd')
3 import time
4 time.ctime(os.path.gettime('/etc/passwd'))

```

4. 获取当前目录里面的指定文件的文件名

```

1 dir_name = '/home/hpc/TensorFlow_Notebook/code'
2 pyfile = [name for name in os.listdir(dir_name) if name.endswith('.py')]
3 #or use glob and fnmatch
4 import glob
5 pyfiles = glob.glob(dir_name+'/*.py')
6 from fnmatch import fnmatch
7 pyfiles = [name for name in os.listdir(dir_name) if fnmatch(name, '*.py')]

```

4. 获取指定目录的文件的相关信息

```

1 import os
2 import os.path
3 import glob
4 import time
5 path_name = '/home/hpc/TensorFlow_Notebook/code'
6 pyfiles = glob.glob(path_name+'/*.py')
7 name_sz_data = [(name, os.path.getsize(name), os.path.getmtime(name)) for name in
                  pyfiles]
8 file_metadata = [(name, os.stat(name)) for name in pyfiles]
9 for name, meta in file_metadata:
10     print(name, '\t|', meta.st_size, '\t|', time.ctime(meta.st_mtime))

```

6.2.3 常见问题

1. 当你的程序获得目录中的一个文件列表，但是当试着打印文件名的时候文件崩溃，出现 UnicodeEncodeError 异常和一条奇怪的消息—surrogates not allow。

打印位置文件名时使用下面的方法可以避免下面的错误：

```

1 def bad_filename(filename):
2     return repr(filename)[1:-1]
3 try:
4     print(filename)
5 except UnicodeEncodeError:
6     print(bad_filename(filename))

```

默认情况下，Python 假定所有文件名都已经根据 sys.getfilesystemencoding() 的值编码过了。但是，有一些文件系统并没有强制要求这样做，因此允许创建文件名没有正确编码的文

件。这种情况不太常见，但是总会有些用户冒险这样做或者是无意之中这样做了（可能是在一个有缺陷的代码中给 open() 函数传递了一个不合规范的文件名）。当执行类似 os.listdir() 这样的函数时，这些不合规范的文件名就会让 Python 陷入困境。一方面，它不能仅仅只是丢弃这些不合格的名字。而另一方面，它又不能将这些文件名转换为正确的文本字符串。Python 对这个问题的解决方案是从文件名中获取未解码的字节值比如 \xhh 并将它映射成 Unicode 字符 \udchh 表示的所谓的“代理编码”。当你有一个不合格的文件名在目录列表中的是后，python 会将其转化为 unicode 如果你有代码需要操作文件名或者将文件名传递给 open() 这样的函数，一切都能正常工作。只有当你想要输出文件名时才会碰到些麻烦（比如打印输出到屏幕或日志文件等）。特别的，当你想打印上面的文件名列表时，你的程序就会崩溃，崩溃的原因就是字符\udce4 是一个非法的 Unicode 字符。它其实是一个被称为代理字符对的双字符组合的后半部分，因此他是一个非法的 Unicode，所以唯一能称该输出的方法就是遇到不合法文件名时采取相应的补救措施。可以将上述代码修改为：

```

1 for name in files:
2     try:
3         print(name)
4     except UnicodeEncodeError:
5         print(bad_filename(name))

```

或者：

```

1 def bad_filename(filename):
2     temp = filename.encode(sys.getfilesystemencoding(), errors='surrogateescape')
3     return temp.decode('latin-1')

```

2. 不关闭一个以打开的文件前提下增加或改变它的 Unicode 编码。

```

1 >>> f = open('sample.txt', 'w')
2 >>> f
3 <_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
4 >>> f.buffer
5 <_io.BufferedReader name='sample.txt'>
6 >>> f.buffer.raw
7 <_io.FileIO name='sample.txt' mode='wb'>
8 >>>

```

在这个例子中，io.TextIOWrapper 是一个编码和解码 Unicode 的文本处理层，io.BufferedReader 是一个处理二进制数据的带缓冲的 I/O 层，io.FileIO 是一个表示操作系统底层文件描述符的原始文件。增加或改变文本编码会涉及增加或改变最上面的 io.TextIOWrapper 层。

detach() 会断开文件最顶层并返回第二层，之后顶层就没什么用了，例如

```

1 >>> f = open('text.txt', 'w')
2 >>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')

```

```

3 >>> b = f.detach()
4 >>> f.write('hello')
5 ValueError
6 <ipython-input-21-0ec9cf64e174> in <module>()
7     f.write('hello')
8
9 ValueError: underlying buffer has been detached

```

一旦断开最顶层后，你就可以给返回结果添加一个新的最顶层，比如：

```

1 >>> f = io.TextIOWrapper(b, encoding='latin-1')
2 <_io.TextIOWrapper name='text.txt' encoding='latin-1'>

```

在文本模式打开的文件中写入原始的字节数据 (将数据直接写入缓冲区)

```

1 In [1]: import sys
2
3 In [2]: sys.stdout.write(b'Hello\n')
4 -----
5 TypeError
6 <ipython-input-2-51d3384e9645> in <module>()
7     sys.stdout.write(b'Hello\n')
8
9 TypeError: write() argument must be str, not bytes
10
11 In [3]: sys.stdout.buffer.write(b'Hello\n')
12 Hello

```

类似的，能够读取文本的 buffer 属性来读取二进制数据。I/O 系统以层级结构的形式构建而成。文本文件是通过在一个拥有缓冲的二进制模式文件上增加一个 Unicode 编码/解码层来创建。buffer 属性指向对应的底层文件。如果你直接访问它的话就会绕过文本编码/解码层。

本小节例子展示的 sys.stdout 可能看起来有点特殊。默认情况下，sys.stdout 总是以文本模式打开的。但是如果你在写一个需要打印二进制数据到标准输出的脚本的话，你可以使用上面演示的技术来绕过文本编码层。3. 你有一个对应于操作系统上一个已经打开的 I/O 通道 (比如文件，管道，套芥子等) 的整形文件描述符，你想将它包装成一个更高层的 Python 文件对象。

一个文件描述和一个打开的普通文件不一样。文件描述仅仅是一个操作系统指定的整数，用来指代某系统的 I/O 通道。如果你碰巧有这么一个文件描述符你可以通过 shiyingopen() 函数来将其包装为一个 Python 的文件对象。你仅仅需要使用这个整数值的文件描述符作为第一个参数来替代文件名即可：

```

1 In [4]: import os

```

```

2 In [5]: fd = os.open('text.txt', os.O_WRONLY|os.O_CREAT)
3 In [6]: f = open(fd, 'wt')
4 In [7]: f.write('hello world\n')
5 In [8]: f.close()

```

当高层文件对象被关闭或者破坏的时候，底层文件描述符也会被关闭。如果这个并不是你想要的结果，你可以给 open() 函数传递一个可选的 closefd=False。比如：

```
1 f = open(fd, 'wt', closefd=False)
```

在 Unix 系统中，这种包装文件描述符的技术可以很方便的将一个类文件接口作用于一个以不同方式打开的 I/O 通道上，如管道、套接字等。举例来讲，下面是一个操作管道的例子：

```

1 from socket import socket, AF_INET, SOCK_STREAM
2
3 def echo_client(client_sock, addr):
4     print('Got connection from', addr)
5
6     # Make text-mode file wrappers for socket reading/writing
7     client_in = open(client_sock.fileno(), 'rt', encoding='latin-1',
8                      closefd=False)
9
10    client_out = open(client_sock.fileno(), 'wt', encoding='latin-1',
11                      closefd=False)
12
13    # Echo lines back to the client using file I/O
14    for line in client_in:
15        client_out.write(line)
16        client_out.flush()
17
18    client_sock.close()
19
20 def echo_server(address):
21     sock = socket(AF_INET, SOCK_STREAM)
22     sock.bind(address)
23     sock.listen(1)
24     while True:
25         client, addr = sock.accept()
26         echo_client(client, addr)

```

需要重点强调的一点是，上面的例子仅仅是为了演示内置的 open() 函数的一个特性，并且也只适用于基于 Unix 的系统。如果你想将一个类文件接口作用在一个套接字并希望你的代码可以跨平台，请使用套接字对象的 makefile() 方法。但是如果要考虑可移植性的话，那上面的解决方案会比使用 makefile() 性能更好一点。

你也可以使用这种技术来构造一个别名，允许以不同于第一次打开文件的方式使用它。例如，下面演示如何创建一个文件对象，它允许你输出二进制数据到标准输出（通常以文本模式打开）：

```

1 import sys
2 # Create a binary-mode file for stdout
3 bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
4 bstdout.write(b'Hello World\n')
5 bstdout.flush()

```

尽管可以将一个已存在的文件描述符包装成一个正常的文件对象，但是要注意的是并不是所有的文件模式都被支持，并且某些类型的文件描述符可能会有副作用（特别是涉及到错误处理、文件结尾条件等等的时候）。在不同的操作系统上这种行为也是不一样，特别的，上面的例子都不能在非 Unix 系统上运行。

5. 创建临时文件和文件夹，在程序执行完后自动销毁。

```

1 from tempfile import TemporaryFile
2 with TemporaryFile('w+t') as f:
3     # Read/write to the file
4     f.write('Hello world \n')
5     f.write('testing\n')
6     # Seek back to beginning and read the data
7     f.seek(0)
8     data = f.read()
9 # Temporary file is destroyed

```

或者，如果你喜欢，你还可以像这样使用临时文件：

```

1 f = TemporaryFile('w+t')
2 # Use the temporary file
3 ...
4 f.close()
5 # File is destroyed

```

`TemporaryFile()` 的第一个参数是文件模式，通常来将文本模式使用 `w+t`, 二进制模式使用 `w+b`。这个模式同时支持读和写操作，在这里很有用，因为当你关闭文件去修改模式的时候，文件实际上已经不存在了。`TemporaryFile()` 另外还支持内置的 `open()` 函数一样的参数。比如：

```

1 with TemporaryFile('w+t', encoding='utf-8', errors='ignore') as f:
2     ...

```

在大多数系统上，同感 `TemporaryFile()` 创建的文件都是匿名的，甚至连目录都没有。如果你想打破这个限制，可以使用 `NamedTemporaryFile()` 来代替。比如：

```

1 In [16]: with NamedTemporaryFile('w+t') as f:
2     ...:     print('filename is:', f.name)
3 from tempfile import NamedTemporaryFile
4 with NamedTemporaryFile('w+t') as f:
5     print('filename is:', f.name)
6 filename is: /tmp/tmp4dwoxytf

```

这里被打开的文件的 `f.name` 属性包含了临时文件的文件名。当你需要将文件传递给其它代码来打开这个文件的 `scipio`, 这个就很有用了, 和 `TemporaryFile()` 一样, 结果文件关闭时会被自动删除调。如果你不想这么做呢, 可以传递一个关键字参数 `delte=False` 即可。比如:

```

1 with NamedTemporaryFile('w+t', delete=False) as f:
2     print('filename is:', f.name)
3 ...

```

为了创建一个临时目录, 可以使以哦嗯 `tempfile.TemporaryDirectory()`。比如:

```

1 from tempfile import TemporaryDirectory
2
3 with TemporaryDirectory() as dirname:
4     print('dirname is:', dirname)
5     # Use the directory
6 ...
7 # Directory and all contents destroyed

```

`TemporaryFile()`、`NamedTemporaryFile()` 和 `TemporaryDirectory()` 函数应该是处理临时文件目录的最简单的方式了, 因为它们会自动处理所有的创建和清理步骤。在一个更低的级别, 你可以使用 `mkstemp()` 和 `mkdtemp()` 来创建临时文件和目录。比如:

```

1 In [19]: tempfile.mkstemp()
2 Out[19]: (13, '/tmp/tmp6heplg63')
3
4 In [20]: tempfile.mkdtemp()
5 Out[20]: '/tmp/tmpcd70_9po'

```

但是, 这些函数并不会做进一步的管理了。例如, 函数 `mkstemp()` 仅仅就返回一个原始的 OS 文件描述符, 你需要自己将它转换为一个真正的文件对象。同样你还需要自己清理这些文件。

通常来讲, 临时文件在系统默认的位置被创建, 比如 `/var/tmp` 或类似的地方。为了获取真实的位置, 可以使用 `tempfile.gettempdir()` 函数。比如:

```

1 In [20]: tempfile.mkdtemp()
2 Out[20]: '/tmp/tmpcd70_9po'
3

```

```
4 In [21]: tempfile.gettempdir()
5 Out[21]: '/tmp'
```

所有和临时文件相关的函数都允许你通过使用关键值参数 prefix,suffix 和 dir 来自定义目录以及命名规则，比如：

```
1 In [24]: from tempfile import NamedTemporaryFile
2 In [25]: f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/tmp')
3 In [26]: f.name
4 '/tmp/mytempw2pxl2v5.txt'
```

最后还有一点，尽可能以最安全的方式使用 tempfile 模块来创建临时文件。包括仅给当前用户授权访问以及在文件创建过程中采取措施避免竞态条件。

你需要将一个 Python 对象序列化为一个字节流，一边将它保存到一个文件，存储到数据库或者通过网络传输它。

用 pickle 模块将一个对象保存在一个文件中

```
1 import pickle
2 data = [1,2,3,4]
3 f = open('sample', 'wb')
4 pickle.dump(data, f)
```

将一个对象保存在一个文件中

```
1 import pickle
2 data = range(10)
3 f = open('temp', 'wb')
4 pickle.dump(data)
```

如果想将一个对象转化为字符串，可以使用 pickle.dumps()：

```
1 s = pickle.dumps(data)
```

为了从字节流中恢复一个对象，使用 pickle.load() 或者 pickle.loads() 函数。比如：

```
1 # Restore from a file
2 f = open('somefile', 'rb')
3 data = pickle.load(f)
4
5 # Restore from a string
6 data = pickle.loads(s)
```

对于大多数应用程序来讲，dump() 和 load() 函数的使用就是你有效使用 pickle 模块所需的全部了。它可适用于绝大部分 Python 数据类型和用户自定义类的对象实例。如果你碰到某个库可能让你在数据库中保存/恢复 Python 对象或者是通过网络传输对象的话，那么很有可能这个库的底层就使用了 pickle 模块。

pickle 是一种 Python 特有的自描述的数据编码。通过自描述，被序列化后的数据包含每个对象开始和结束以及它的类型信息。因此，你无需担心对象记录的定义，它总是能工作。举个例子，如果要处理多个对象，你可以这样做：

```

1 >>> import pickle
2 >>> f = open('somedata', 'wb')
3 >>> pickle.dump([1, 2, 3, 4], f)
4 >>> pickle.dump('hello', f)
5 >>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
6 >>> f.close()
7 >>> f = open('somedata', 'rb')
8 >>> pickle.load(f)
9 [1, 2, 3, 4]
10 >>> pickle.load(f)
11 'hello'
12 >>> pickle.load(f)
13 {'Apple', 'Pear', 'Banana'}
14 >>>

```

还能序列化成函数，类，接口，但是结果数据仅仅将他们的名称编码成对应的代码对象。例如

```

1 >>> import math
2 >>> import pickle.
3 >>> pickle.dumps(math.cos)
4 b'\x80\x03cmath\ncos\nq\x00.'
5 >>>

```

当数据反序列化回来的时候，会先假定所有的源数据时可用的。模块、类和函数会自动按需导入进来。对于 Python 数据被不同机器上的解析器所共享的应用程序而言，数据的保存可能会有问题，因为所有的机器都必须访问同一个源代码。有些类型的对象是不能被序列化的。这些通常是那些依赖外部系统状态的对象，比如打开的文件，网络连接，线程，进程，栈帧等等。用户自定义类可以通过提供 `__getstate__()` 和 `__setstate__()` 方法来绕过这些限制。如果定义了这两个方法，`pickle.dump()` 就会调用 `__getstate__()` 获取序列化的对象。类似的，`__setstate__()` 在反序列化时被调用。为了演示这个工作原理，下面是一个在内部定义了一个线程但仍然可以序列化和反序列化的类：

```

1 # countdown.py
2 import time
3 import threading
4
5 class Countdown:
6     def __init__(self, n):

```

```

7     self.n = n
8     self.thr = threading.Thread(target=self.run)
9     self.thr.daemon = True
10    self.thr.start()
11
12    def run(self):
13        while self.n > 0:
14            print('T-minus', self.n)
15            self.n -= 1
16            time.sleep(5)
17
18    def __getstate__(self):
19        return self.n
20
21    def __setstate__(self, n):
22        self.__init__(n)

```

运行下面结构化代码

```

1 >>> import countdown
2 >>> c = countdown.Countdown(30)
3 >>> T-minus 30
4 T-minus 29
5 T-minus 28
6 ...
7
8 >>> # After a few moments
9 >>> f = open('cstate.p', 'wb')
10 >>> import pickle
11 >>> pickle.dump(c, f)
12 >>> f.close()

```

然后退出 Python 解析器并重启后再试验下:

```

1 >>> f = open('cstate.p', 'rb')
2 >>> pickle.load(f)
3 countdown.Countdown object at 0x10069e2d0>
4 T-minus 19
5 T-minus 18
6 ...

```

你可以看到线程又奇迹般的重生了，从你第一次序列化它的地方又恢复过来。

pickle 对于大型的数据结构比如使用 array 或 numpy 模块创建的二进制数组效率并不是一个高效的编码方式。如果你需要移动大量的数组数据，你最好是先在一个文件中将其保存为数组数据块或使用更高级的标准编码方式如 HDF5 (需要第三方库的支持)。

由于 pickle 是 Python 特有的并且附着在源码上，所有如果需要长期存储数据的时候不应该选用它。例如，如果源码变动了，你所有的存储数据可能会被破坏并且变得不可读取。坦白来讲，对于在数据库和存档文件中存储数据时，你最好使用更加标准的数据编码格式如 XML, CSV 或 JSON。这些编码格式更标准，可以被不同的语言支持，并且也能很好的适应源码变更。

6.3 正则表达式介绍

操作符	说明	实例
[]	字符集合, 对单个字符给出取值范围	[abc] 表示 a,b,c,[a-z] 表示 a 到 z 的单个字符
.	任何单个字符	
[^]	非字符集, 对单个字符给出排除范围	[^ abc] 表示 非 a 或者 b 或者 c 的单个字符
*	前一个字符 0 次或者无限次扩展	abc* 表示 ab,abc,abcc 等
+	前一个字符 1 次或无限次扩展	abc+ 表示 abc,abcc,abccc 等
?	前一个字符 0 次或者一次扩展	abc? 表示 ac,abc
	左右表达式任一个	abc def 表示 abc 或者 def
{m}	扩展前一个字符 m 次	ab{2}c 表示 abc,abbc
{m,n}	扩展前一个字符 m 到 n 次, 包含 n	ab{1,2}c 表示 abc,abbc
^	匹配字符串开头	^abc 表示 abc 且在一个字符串开头
\$	匹配字符串结尾	abc\$ 表示 abc 且在一个字符串的结尾
()	分组标记, 内部只能使用 操作符	(abc) 表示 abc, (abc def) 表示 abc 或者 def
(...)	这是一个扩展的符号, 第一个字符在'?' 后面决定了深层的语法。扩展通常没有创建一个新的 group,(?P<name>...) 时该规则惟一的特例)	
(?aiLmsux)	来自集合'a','i','L','m','s','u','x' 的一个或者多个字母, group 匹配空字符串字符给整个正则表达式设置相关的 flags: re.A,re.I,re.L,re.M,re.S,re.X。如果你洗完桑包含 flags 作为正则表达式的一部分而不是传递一个 flag 参数到 re.compile() 函数这就是很有用的, Flasg 应该首先用在表达式字符串。	

操作符 (?:...)	说明	实例
\d	数字等价与 [0-9]	
\D	非数字等价与 [0-9]	
\number	匹配相同 number 的组。组以 1 开始, 例如 (.+)\1 匹配'the the'or'55 55', 但是'the the'(中间需要有空格), 这种特殊的序列仅仅被用来匹配 1 到 99 组。如果第一个数字为 0 或者是 3 为八进制的, 他将被解释为一个 group match, 在字符类 '[' and ']' 中, 所有的数被当作字符。	
A 匹配	匹配字符串的开始	
\b	匹配空字符串, 但是仅仅是单词前面或者后面的空字符串, 单词被定义为一个 unicode 字母数字序列或下划线特征, 因此单词为被空格或者为字母数字预示, 非强跳得字符串, 注意, \b 被定义为 a\w 和 a\W 之间, 或者在\w 和单词开始之间, 这意味着 r'\bfoo\b' 匹配'foo','foo.','','(foo)','','bar foo baz' 而不是'foobar' 或者'foo3'	
\B	匹配空字符串, 但是仅仅当它不在单词的开头或者结尾时, 这意味着 r'py\B' 匹配'python','py3','py2', 而不是'py','py.' 或者是'py!'\B 和\b 相反, 因此单词时 unicode 字母数字或者下划线, 尽管这能被 ASCII flag 改变	
\S	匹配不是任何不是空格的 unicode 字符, 和\s 相反, 如果 ASCII flag 被用这因为等于 [^\t\n\r\f\v](但是 flag 影响整个正则表达式, 因此在这种情况下 [^\t\n\r\f\v])	
z	匹配字符串的尾部	
(?imsx-imsx:...)	在字符字母集合'i','m','s','x' 中, '-' 跟着的来自同样字母集合的一个或者更多字母), 对于部分表达式字母集合或者移去相关的 flags:re.I,re.M,re.S,re.X。	
<?P=name>	:对于 group 的一个反向引用, 它匹配之前 name 命名的 group 无论什么文本。	

(?+...)	一个注释，括号里面的内容被简单的忽视	
(?==...)	如果... 匹配下一步，不小于任何字符串。例如 Isaac (?=Asimov) 将匹配'Isacc' 如果它被'Asimov' 跟着的话。	
(?!....)	如果... 不匹配下一个，例如 Isaac (?!Asimov) 将匹配'Isaac'，仅仅是它没有'Asimov' 跟着。	
\w	单词字符，等价与 [A-Za-z0-9_]	

正则表达式的语法实例

P(Y YT YTH YTHO)?N	'PN', 'PYN', 'PYTN', 'PYTHN', 'PYTHON'
PYTHON+	'PYTHON', 'PYTHONN', 'PYTHONNN', ...
PY[TH]ON	'PYTON', 'PYHON'
PY[TH]?ON	'PYON', 'PYaON', 'PYbON', 'PYcON', ...
PY{:3}N	'PN', 'PYN', 'PYYN', 'PYYYN', ...

常用的正则表达式:

^[A-Za-Z]+\$	26 个字母组成的字符串
^[A-Za-z0-9]+\$	由 26 个字母和数字组成的字符串
^-?\d+\$	整数形式的字符串
^[0-9]*[1-9][0-9]* \$	正整数形式的字符串
[1-9]\d{5}	中国境内邮政编码，6 位
[\u4e00-\u9fa5]	匹配中文字符
\d{3}-\d{8} \d{4}-\d{7}	国内电话号码，010-68913536

匹配 IP 地址的正则表达式: \d+\.\d+\.\d+ 或者 \{1,3\}. 精确写法:

0-99:[1-9]? \d

100-199:1\d{2}

200-249:2[0-4]? \d

250-255:25[0-5]

IP 地址的正则表达式: (([1-9]? \d|1\d{2}|2[0-4]\d|25[0-5]).){3}(([1-9]? \d|1\d{2}|2[0-4]\d|25[0-5])

6.4 RE 库的主要功能函数

re.search()	在一个字符串搜索匹配正则表达式的一个位置。
re.match()	从一个字符的开始为值起匹配正则表达式，返回 match 对象。
re.fullmatch()	如果整个字符串匹配正则表达式然会相应的 match 对象，不匹配返回 None，注意这不同于 0 长度匹配
re.findall()	搜索字符串，以列表类型返回全部匹配的字串
re.split()	将一个字符串按照正则表达式匹配结果进行分割，返回列表类型
re.finditer()	搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素时 match 对象
re.sub()	在字符串中替换所有匹配正则表达式的子串，返回替换后的字符串。
re.subn()	执行替换操作凡是返回一个 (new_string,number_of_subs_made) 元组
re.escape(pattern)	转义素有的字符除了 ASCII 字母，数字和下划线，如果你想匹配一个也许有正则表达式在里面的任一字符串这是很有用的。
re.purge()	清除正则表达式缓存

re.search(pattern,string,flags=0): 在一个字符串中搜索匹配正则表达式的一个位置返回 match 对象。

- pattern: 正则表达式的字符串或原声字符串表示。
- string: 待匹配字符串。
- flags: 正则表达式使用时的控制标记。

6.4.1 re 表达式中的 flags	使\w \W\b\B\d\D \s\S 值执行 ASCII 匹配而不是 Unicode 匹配，仅仅对于 Unicode 样式有意义对 Byte 样式忽略。
re.A	
re.DEBUG	显示编译表达式的调试信息
re.I	忽略正则表达式的大小写，[A-Z] 能够匹配小写。
re.L	使得\w \W\b\B\d\D \s\S 依赖于当前现场，当现场机制不可信时不鼓励使用，在不管什么时候它处理一个 cultrue，你应该用 Unicode 匹配，这个 flag 仅仅可以被用在 bytes 样式中。
re.M	正则表达式中的操^ 作能够将给定字符串的每一行当作匹配开始
re.S	正则表达式中的. 操作能够匹配所有的字符，默认匹配除换行外的所有字符
re.VERBOSE(re.X)	这个 flag 通过允许你分割逻辑部分和增加注释允许你写的正则表达式更好，空 pattern 中的空格被忽略特别是当一个字符类或者当有为转义的反斜线时，当一行包含不饿时字符类得 # 和非转义斜线时，所有的左边以 # 开头的字符将被忽略
re.error(msg, pattern=None, pos=None)	<ul style="list-style-type: none"> - msg: 非正式格式的错误消息 - pattern: 正则表达式 - pos: 在 pattern 编译失败的索引（也许是 None） - lineno: 对应位置的行（也许是 None） - colno: 对应位置的列（也许是 None）

```

1 import re
2 match = re.match(r'1\d{5}', 'BIT 100081')
3 if match:
4     match.group(0)

```

re.match(pattern,string,flags=0): 从一个字符串的开始位置起匹配正则表达式，返回 match 对象。

```

1 import re
2 match = re.match(r'1\d{5}', '100081 BIT')
3 if match:
4     print(match.group(0))

```

re.findall(pattern,string,flags=0): 搜索字符串，以列表类型返回能匹配的子串。

```

1 import re
2 ls = re.findall(r'1\d{5}', 'BIT 100081 TSU100084')

```

re.split(pattern,string,maxsplit = 0,flags=0): 将字符串按照正则表达式匹配结果进行分割，返回列表类型。

maxsplit: 最大分割数，剩余部分作为最后一个元素输出。

```

1 import re
2 re.split(r'1\d{5}', 'BIT100081 TSU100084')
3 re.split(r'1\d{5}', 'BIT100081 TSU100084', maxsplit=1)

```

re.finditer(pattern,string,flags=0): 搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素时 matchdurian。

```

1 import re
2 for m in re.finditer(r'1\d{5}', 'BIT100081 TSU100084'):
3     if m:
4         print(m.group(0))

```

re.sub(pattern,repl,string,count=0,flags=0) 在一个字符串中替换所有匹配正则表达式的子串返回替代后的字符串。

- repl: 替换匹配字符串的字符串
- string: 待匹配字符串
- count: 匹配的最大替换次数

```

1 import re
2 re.sub(r'1\d{5}', '110', 'BIT100081 TSU100084')

```

Re 库的另一种等价用法:

```

1 rst = re.search(r'1\d{5}', 'BIT 100081')

```

等价于

```

1 pat = re.compile(r'1\d{5}')
2 pat.search('BIT 100081')

```

regex.search	在字符串中搜索匹配正则表达式的第一位置，返回 match 对象
regex.match()	在字符串的开始为值起配置正则表达式，返回 match 对象
regex.findall()	所有字符串，以列表类型返回全部能匹配的子串
regex.split()	将字符串按照正则表达式匹配结果进行分割，返回列表类型。
regex.finditer()	搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素是 match 对象
reg.sub()	在一个字符串中替换所有匹配正则表达式的子串，返回替换后的字符串

Match 对象：一次匹配的结果，包含匹配的很多信息。

```

1 match = re.search(r'1\d{5}', 'BIT 100081')
2 if match:
3     print(match.group(0))
4 type(match)

```

match 对象的属性和方法

.string	待匹配的文本
.re	匹配时使用的 patter 对象 (正则表达式)
.pos	正则表达式搜索文本的开始位置
.endpos	正则表达式搜索文本的结束位置
.group(0)	获得匹配后的字符串
.start()	匹配字符串在原始字符串的开始位置
.end()	匹配字符串的结尾位置
.span()	返回 (.start(),.end())
.expand()	用 sub() 方法返回一个通过在 temple 字符串替代\的像\n 被转换成合适的字符串，数值反向索引 (\1,\2) 和 (\g<1>,\g<name>) 被相应组里面的内容取代\字符串
.__getitem__(g)	允许你轻松的访问一个 match 组
.groupdict(default=None)	返回一个包含所有子组的匹配对象, key 是子组的名字, 被用在 groups 的默认参数 默认参数不参加匹配, 默认值时 None。
.lastindex	最新匹配的组的整数索引, 或者如果没有组被匹配就为 None。例如表达式 (a)b,((a)(b)) 和 ((ab)) 将有 lastindex == 1 如果应用的字符串'ab', 然而表达式 (a)(b) 将有 lastindex == 2, 如果与应用在同一个字符串。
.lastgroup	最新匹配名字, 如果 group 没有一个名字或者没有 group 就匹配为 None。
.re	正则表达式的 match() 或者 search() 方法生成的 match 实例

Re 库默认采用贪婪匹配, 即输出匹配最长的字子串

```

1 match = re.search(r 'PY.*N', 'PYANBNCNDN')
2 match.group(0)

```

通常搜索的时候 PYAN 就能匹配出结果但是根据贪婪匹配, 匹配待匹配字符串中最长的字符串。输出最短子串 PYAN。

```

1 match = re.search(r 'PY.*?N', 'PYANBNCNDN')

```

最小匹配操作符

操作符	说明
*?	前一个字符 0 次或者无限次扩展，最小匹配
+?	前一个字符 1 次或者浮现次扩展，最小匹配
??	前一个字符 0 次或者 1 次扩展，最小匹配
{m,n}?	扩展前一个字符串 m 到 n 次(含 n)，最小匹配

```

1 import re
2 m = re.match(r'(\w+ \w+)', 'Isaac Newton, physicist')
3 m.group(0)
4 m.group(1)
5 m.group(2)
6 m.group(1,2)

```

输出：

```

'Isaac Newton'
'Isaac'
'Newton'
('Isaac','Newton')

```

```

1 m = re.match(r'(\d+).(\d+)', '3.1415')
2 m.groups()

```

输出：

```
('3','1415')
```

```

1 m = re.match(r'(?P<first_name>\w+) (?P<last_name>\w+)', 'Malcolm Reynolds')
2 m.groupdict()

```

输出：'first_name': 'Malcolm', 'last_name': 'Reynolds'

6.5 常用的 sys 函数

- sys.abiflags: 在 POSIX 体同上 Python 用标准的 configure 脚本编译, 包含 PEP3149 指定的 ABI flags。
- sys.argv: 传递给 Python 的命令行参数, argv[0] 是脚本的名字, 在解释器中如果命令行用-c 选项, argv[0] 被设置为'-c'。如果没有脚本名字被传递给 python 解释器, argv[0] 是空字符串。
- sys.base_exec_prefix: Python 启动时设置, 在 site.py 之前运行前设置为 exec_prefix。如果不运行一个虚拟环境, 值保持不变, 如果 site.py 找到的虚拟环境被用了, prefix 和 exec_prefix 的值将被改变到指向虚拟环境, 由于 base_prefix 和 base_exec_prefix 将任何指向 python 安装的 base 环境 (虚拟换将被创建)。
- sys.base_prefix: 在 site.py 运行前 python 启动中值和 prefix 相同。如果不运行在虚拟环境中, 值将保持不变 rugosasite.py 找到一个虚拟环境被用, prefix 和 exec_prefix() 值将被改变到指向虚拟环境, 由于 base_prefix 和 base_exec_prefix 将保留指向 python 安装的 base 环境 (虚拟换将被创建)。
- byteorder: 本地变量的指示器, 这将在 big-endian 平台有一个值'big','title' 在 little-endian 平台。
- sys.vuiltin_module_name: 被编译进 Python 解释器的模块的字符串元组。(信息在其他方法下不可用-modules.keys() 仅仅显示导入的模块)。
- sys.call_tracing(func,args): 调用 func(*args), 当 trace 使能时。trace 状态被后来保存和恢复。从 checkpoint 文件 debug 去玄幻调试其它代码。
- sys.copyright: 包含 python 解释器版权信息的字符串。
- sys._clear_type_cache(): 清除内部变量的缓存, 类型缓存用来加速属性和方法的查找这个函数用来降低泄漏 debug 的非比要得查找。
- sys._current_fnames(): 返回映射每个线程的标识符到函数调用时的线程栈的栈顶。注意 traceback 模块中的函数能编译调用被给定一个帧的栈。在调试线程锁时很有用: 这个函数线程锁死操作, 这样线程的调用被冻结和特们的死锁一样长。帧返回一个非死锁的线程也许忍受没有关系到当前这次调用的代码激活的线程检查帧。, 这个函数仅仅被用在内部或者特殊的目的。
- sys._debugmallocstats(): 打印 cpython 内存分贝其的低级的信息到标准的错误输出。如果 python 配置了-with-pydebug, 它也只行一些开销巨大的内部组成检查。

- sys.dllhandle: 指定处理 python dll 的整数, 在 Windows 上可用。
- sys.displayhook(value): 如果值为 None, 函数打印 rep(value) 到 sys.stdout, 报春之在 builtins.__。如果 repr(value) 时不可编码的 sys.stdout.encoding 和 sys.stdout.error 句柄, 解码 sys.stdout.encoding 和 backslashreplace 错误句柄。sys.displayhook 被调用在计算输入交互式 python 会话表达式的结果, 显示值能通过指定参数被自定义。

```

1 def displayhook( value ):
2     if value is None:
3         return
4     builtins.___ = None
5     text = repr( value )
6     try:
7         sys.stdout.writer( text )
8     except UnicodeEncodeError:
9         bytes = text.encode( sys.stdout.encoding, 'backslashreplace' )
10        if hasattr( sys.stdout, 'buffer' ):
11            sys.stdout.buffer.writer( bytes )
12        else:
13            text = bytes.decode( sys.stdout.encoding, 'strict' )
14            sys.stdout.buffer.writer( text )
15        sys.stdout.write( "\n" )
16        builtins.___ = value

```

- sys.dont_write_bytecode: 如果为真, python 不尝试写.pyc 文件到源模块, 值依赖-B 命令行选项和 PYTHONDONTWRITEBYTECODE 环境变量通过设置 True 或者 False 确定, 但是你可以在你自己控制二进制文件生成。
- sys.excepthook(type,value,traceback): 这个函数打印出一个给定的 traceback 和 sys.stderr 异常。当出现异常时, 解释器设置三个参数异常类, 异常实例和 traceback 对象调用 sys.execpthook。在交互式会话中这发生在控制被返回到终端前, 在 Python 程序中仅当程序退出时被调用, 在处理类似顶级异常可以通过指定另一个三个参数函数它哦 sys.exeothook。
- sys.__displayhook__
- sys.__excepthook__: 这个对象在程序的开头包含 displayhook 和 excepthook 的初始值, 他们被保存以至于他们被异常取代时 displayhook 和 excepthook 可以被恢复。
- sys.exc_info: 这个函数给出关于当前被处理的异常的信息的元组。信息返回被指定到当前线程和当前栈帧, 如果当前栈帧没有处理异常, 信息被调用的栈帧得到, 或者它的调用器得到, 因此直到在处理异常时栈帧被发现, 这里处理一个异常被定义为处理

一个异常发生。对于任何栈帧，仅仅当前异常信息被处理。如果在栈帧中没有异常被处理，返回包含三个 None 的元组。否则返回值为 (type,value,traceback)，他们分别为 d 得到的被处理异常的类型，异常实例，和 traceback 对象（压缩调用栈）。

- sys.exec_prefix: 一个字符串给 site-specific 目录前缀到 python 文件安装平台之前，默认是’/usr/local’，这可以通过设置 configure 脚本-exec-prefix 参数被设置编译时间，特别是所有的配置文件(像 pyconfig.h 头文件)被安装于 exec_prefix/lib/pythonX.Y/config 和共享库模块被按转子于 exec_prefix/lib/pythonX.Y/lib-dynload，这里 X,Y 代表跑一趟好哦那得版本。
- sys.executable: 给 python 解释器一个绝对路径字符串，如果 python 不能获得真是的执行路径，sys.executable 将为 None。
- sys.exit([arg]): 从 python 推出，SystemExit 异常时生成，选项参数可以被给定为整数(默认为 0)或者其它对象类型。如果时一个整数，0 被认为成功终止，任何非零数值被认为异常终止。多数系统要求值在 0-127 之间，否则将产生不确定结果，一些系统约定指定推出代码，但是通常不完善，Unix 程序生成用 2 代表命令行语法错误 1 代表其它错误，如果另一类型的对象被传递，None 相当于传递 0，其他队像被打印到 stderr 和推出代码为 1，类似的 sys.exit("some error message") 是当程序出错一个快速退出程序的方法。因此 exit() 当从主进程退出进程时产生异常。异常不被拦截。
- sys.flags 结构序列 flags 暴露命令行状态

attribute	flag
debug	-d
inspect	-i
interactive	-i
optimize	-O or -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
verbose	-v
bytes_warning	-b
quit	-q
hash_randomization	-R

- sys.float_info: 一个结构序列保持 float 类型的信息，它包含精确度和内部表达式低级信息，值符合在头文件 float.h 中定义的浮点常数。

attribute	float.h macro	explanation
epsilon	DBL_EPSILON	1 和大于 1 的最新值之间的差作为浮点数
dig	DBL_DIG	浮点数能带秒的最大精度
mant_dig	DBL_MANT_DIG	浮点精度。base-radix 浮点数的精度
max	DBL_MAX	有限浮点数的最大值
max_exp	DBL_MAX_EXP	radix ^(e-1) 代表的最大整数 e 代表无穷浮点数
max_10_exp	DBL_MAX_10_EXP	最大 $e10^{**2}$ 代表的最大浮点
radix	FLT_RADIX	指数表达式的基数
rounds	FLT_ROUNDS	整数常数代表 round 模式，这反映了系统在解释器启动时 FL

属性 sys.float_info.dig 需要更进一步扩展，如果 s 时任何字符串表达一个十进制数，然后转换 s 为浮点数将恢复一个字符串表达式。

```

1 import sys
2 sys.float_info.dig
3 15
4 s = '3.14159265358979'      # decimal string with 15 significant digits
5 format(float(s), '.15g')    # convert to float and back -> same value
6 '3.14159265358979'

```

但是对于字符串 sys/float_info.dig 指定精度，这不总是 true。

```

1 s = '9876543211234567'      # 16 significant digits is too many!
2 format(float(s), '.16g')    # conversion changes value
3 '9876543211234568'

```

- sys.float_repr_style: 指示 repr() 函数如何处理入店时的字符串。日国字符串有一个值'short' 然后对于一个有限的浮点数 x, repr(x) 产生一个短字符串 float(repr(x)) == x。否则 float_repr_style 有值'legacy' 和 repr(x) 行为正如子啊 python3.1 中的一样。
- sys.getalloctedblock(): 返回解释器当前分配的内存块数量，这个函数在更重和调式内存泄漏时很有用，因为解释器内部换传，结果可能因为调用而不同，你也许可以调用 _clear_type_cache() 和 gc.collect() 得到雨鞋结果。如果 python 编译实现不能合理的计算这些信息，getalloctedbloacks() 允许返回 0。
- sys.getdefaultencoding(): 返回 Unicode 实现的字符串的默认编码的名字。
- sys.getdlopenflags(): 同 dlopen() 返回当前 flag 的值。flag 值的符号名字能被在 os 模块中找到

- `sys.getfilesystemencoding()`: 返回用于转换 Unicode 文件名和 bytes 文件名的编码名字, 为了最好的兼容性, str 应该在所有情况下被用在 filename, 尽管文件名作为 bytes 被支持, 函数接受 fanti 文件名应该支持 str 或者 bytes 内部转换系统偏好的表达。编码总是兼容 ASCII os.fsencode() 和 os.fsdecode() 应该被用于保证正确的编码和错误的模型使用。
 - 在 MAC OS 上编码为 utf-8
 - Unix 编码时 locale 编码
 - 在 windows 上编码也许是'utf-8' 或者是'mbcs', 依赖于用户配置。
- `sys.getfilesystemencodererrors()`: 返回转换 unicode 文件名和 bytes 文件名错误模式的名字, 编码名字有 `getfilesystemencoding()` 指定的便阿妈名字。os.fsencode() 和 os.fsdecode() 用来确保争取的编码和错误模式使用。
- `sys.getrefcount(object)`: 返回 object 对象的引用返回的储量通常高于你认为的呀, 因为它包含临时引用作为 `getrefcount()` 的参数。
- `sys.getsizeof(object,[,default])`: 返回对象的比特大小, 对象可以使任何类型的对象, 所有内建的多项将返回争取的结果, 但是这没有保持新的第三方扩展作为实现。仅仅内存消耗直接属性到对象, 对象访问时没有内存消耗。如果内定默认将返回不提供均值到这个值, 否则, `TypeError` 将产生。`getsizeof()` 调用对象的 `__sizeof__` 方法, 如果对象通过垃圾回收器管理增加一个额外的垃圾回收器。
- `sys.getrecursionlimit()`: 返回循环限制的当前值, 最大的 python 解释器栈深度。这限制阻止由无限循环从 c 栈移除和 python 崩溃, 它可以被 `setrecursionlimit()`。
- `sys.getsizeof(object,[,default])`: 返回对象的比特大小, 对象可以使任何类型, 所有内建的兑奖将被正确返回, 但是不是必须保持 true 给第三方扩展当它的实现被指定, 仅仅对象的直接内存消耗属性, 不是独享引用的内存消耗。如果对象没有给定获取大小, 默认将被返回, 否则 `TypeError` 将被报出。
- `sys.getwitchinterval()` 返回解释器的线程交换区间。
- `sys._getframe([depth])`: 从调用的栈返回一个帧对象, 如果宣讲整数 depth 被给定, 返回栈顶下的帧对象调用。如果 depper 比调用的栈深, `ValueError` 被报出。默认深度为 0, 返回调用栈顶的帧。
- `sys.getprofile()`: 获取 `setprofile()` 设置的 profile 函数。
- `sys.gettrace()`: 得到 `settrace()` 的 trace 函数。

- sys.getwindowsversion(): 返回一个描述当前 windows 版本的描述的名字元组。命名元素时 major,minor,build,platform,service_pack_minor,service_pack_major,suit_mask,product_type 和 platform_version.service_pack 包含一个字符串, platform_version 一个三元组和所有其它值。这个组建可以同感 name 访问, 因此 sys.getwindowsversion()[0] 被等同
 (VER_NT_WORKSTATION) 系统是工作站
 platform 将被 2(VER_PLATFORM_WIN32_NI) (VER_NT_DOMAIN_CONTROLLER) 是系统是域控制器
 (VER_NT_SERVER) 系统是服务器
 函数包装 WIN32 GetVersionEx() 函数, windows 可用
- sys.get_asyncgen_hooks(): 返回一个类似名称元组的 asyncgen_hooks 对象, 这里 firstier 和 expected 均可设为 None 或者获取异步生成器作为参数的函数, 通过时间循环调度异步生成器终止。
- sys.get_coroutine_wrapper(): 返回 None 或者一个由 set_coroutine_wrapper 包装器.

- | | |
|--|--|
| <ul style="list-style-type: none"> • sys.has_info: 数值 hash 实现参数给一个结构序列。 | width hash 值的位宽
modulus 用于数值 hash 方案的主要模块 P
inf 返回正无穷大的 hash 值
nan 返回非数的 hash 值
imag 返回复数的虚部
algorithm str,bytes 和 memoryview 的 hash 算法的实现
hash_bits hash 算法的内部输出大小
seed_bits hash 算法的种子值 |
|--|--|
- sys.hexversion: 单个证书的编码版本。这被保证增加, 包括合适的 support non-product release 版本, 例如。测试 Python 解释器时最新的版本 1.5.2 用

```

1 if sys.hexversion >= 0x010502F0:
2     # use some advanced feature
3     ...
4 else:
5     # use an alternative implementation or warn the user
6     ...

```

- sys.
- sys.
- sys.

- sys.
- sys.

6.6 collections

collections 是 Python 内建的一个集合模块，提供了许多有用的集合。

6.6.1 namedtuple

namedtuple 是一个函数，用来创建一个自定义的 tuple 对象，并且规定了 tuple 元素的个数，并且可以用属性而不是索引访问 tuple 的某个元素。定义一个 Point 类型。

```

1 from collections import namedtuple
2 Point = namedtuple('Point', ['x', 'y'])
3 p = Point(1, 2)
4 p.x
5 p.y

```

上面创建的 Point 是 tuple 的一种子类：

```

1 >>> isinstance(p, Point)
2 True
3 >>> isinstance(p, tuple)
4 True

```

用坐标和半径表示一个元，可以用 namedtuple 定义：

```

1 Circle = namedtuple('Circle', ['x', 'y', 'r'])

```

6.6.2 deque

deque 是为了高效实现插入和删除操作的双向列表，适合与队列和栈：

```

1 In [1]: from collections import deque
2 In [2]: q = deque(['a', 'b', 'c', 'd'])
3 In [3]: q.append('f')
4 In [4]: q
5 Out[4]: deque(['a', 'b', 'c', 'd', 'f'])
6 In [5]: q.appendleft('1')
7 In [6]: q
8 Out[6]: deque(['1', 'a', 'b', 'c', 'd', 'f'])
9 In [7]: q.rotate()
10 In [8]: q
11 Out[8]: deque(['f', '1', 'a', 'b', 'c', 'd'])

```

6.6.3 defaultdict

使用 dict 时，如果 key 不存在就会抛出 KeyError。如果不希望 key 不存在时，返回会一个默认值，就可以采用 defaultdict。

```

1 In [9]: from collections import defaultdict
2 In [10]: aa = defaultdict(lambda: 'N/A')
3 In [11]: aa['key1'] = 'abc'
4 In [12]: aa['key1']
5 Out[12]: 'abc'
6 In [13]: aa['key2']
7 Out[13]: 'N/A'
```

除了 key 不存在时返回默认值外，defaultdict 的其他行为和 dict 完全一样。

6.6.4 OrderedDict

使用 dict 时，Key 是无序的。对 dict 做迭代时，我们无法确定 Key 的顺序。要保持 key 的顺序可以使用 OrderedDict：

```

1 In [1]: from collections import OrderedDict
2 In [2]: a = dict([('a',1),('E',2),('B',3),('Q',5)])
3 In [3]: a
4 Out[3]: {'B': 3, 'E': 2, 'Q': 5, 'a': 1}
5 In [4]: b = OrderedDict([('a',1),('E',2),('B',2),('Q',5)])
6 In [5]: b
7 Out[5]: OrderedDict([('a', 1), ('E', 2), ('B', 2), ('Q', 5)])
```

OrderedDict 的 Key 会按照插入的顺序排列，不是 key 本身的排序。

```

1 In [7]: od['z'] = 1
2 In [8]: od['y'] = 2
3 In [9]: od['x'] = 3
4 In [10]: od
5 Out[10]: OrderedDict([('z', 1), ('y', 2), ('x', 3)])
```

OrderedDict 可以实现 FIFO 的 dict，当容量超出限制时，先删除最早添加的 Key。

```

1 from collections import OrderedDict
2 class LastUpdatedOrderedDict(OrderedDict):
3     def __init__(self, capacity):
4         super(LastUpdatedOrderedDict, self).__init__()
5         self._capacity = capacity
6     def __setitem__(self, key, value):
7         containsKey = 1 if key in self else 0
8         if len(self) - containsKey >= self._capacity:
```

```
9     last = self.popitem(last=False)
10    print('remove:', last)
11    if containsKey:
12        del self[key]
13        print('set:', (key, value))
14    else:
15        print('add:', (key, value))
16    OrderedDict.__setitem__(self, key, value)
```

6.6.5 Counter

Counter 是一个简单的计数器，例如统计字符出现的个数。

```
1 In [11]: from collections import Counter
2 In [12]: c = Counter()
3 In [13]: for ch in 'programming':
4 ...:     c[ch]=c[ch]+1
5 ...
6 In [14]: c
7 Out[14]: Counter({'a': 1, 'g': 2, 'i': 1, 'm': 2, 'n': 1, 'o': 1, 'p': 1, 'r': 2})
```

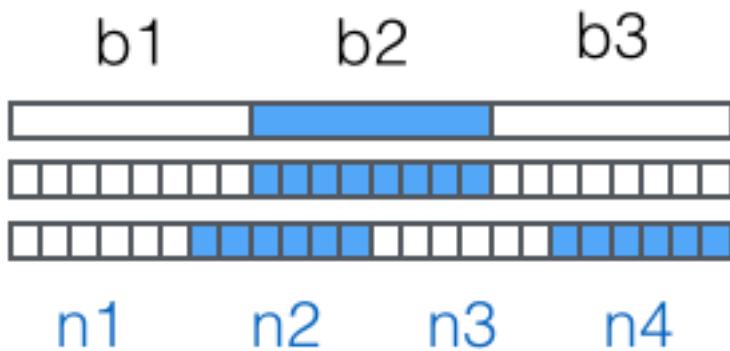
6.7 base64

base64 是一种用 64 个字符来表示任意二进制数据的方法。

用记事本打开 exe,jpg, pdf 这些文件时，我们都会看到一大堆乱码。因为二进制文件包含很多无法显示和打印的字符，所以如果要让记事本这样额的文本处理软件处理二进制数据，就需要一个二进制到字符串的转换方法。Base64 是一种最常见的二进制编码方法。

Base64 的原理很简单，首先，准备一个包含 64 个字符串的数组: ['A','B','C',...,'a','b','c',...,'0','1',...,'+', '/']

然后对二进制数据进行处理，每 3 个字节一组，一共是 $3 \times 8 = 24bit$, 划为 4 组，每组正好 6 个 bit。



这样我们得到 4 个数字所谓索引，然后查表，获得相应的 4 个字符，这就是编码后的字符串。所以 Base64 编码会把 3 字节的二进制数据编码为 4 字节的文本数据，长度正佳 33%。好处是编码后的文本数据可以在邮件正文，网页等直接显示。

如果需要编码的二进制数据不是 3 的倍数，最后会剩下 1 个或 2 个字节怎么办?Base64 用

x00 字节在末尾补足后，再在编码的末尾加上一个或者 2 个 = 号，表死补了多少字节，解码的时候会自动去掉。

python 内置的 base64 可以直接进行 base64 编码解码:

```

1 In [15]: import base64
2 In [17]: base64.b64encode(b'binary\x00string')
3 Out[17]: b'YmluYXJ5AHN0cmLuZw=='
4 In [18]: ben = base64.b64encode(b'binary\x00string')
5 In [19]: base64.b64decode(ben)
6 Out[19]: b'binary\x00string'
7 In [20]: ben
8 Out[20]: b'YmluYXJ5AHN0cmLuZw=='

```

由于标准的 base64 编码后可能出现 + 和 /。在 URL 中就不能直接作为参数，所以又有之中“url safe”的 base64 编码，其实就是吧字符 + 和 / 分别变成了 - 和 _。

```
1 In [23]: base64.b64encode(b'i\xb7\x1d\xfb\xef\xff')
2 Out[23]: b'abcd++//'
3 In [25]: bus = base64.urlsafe_b64encode(b'i\xb7\x1d\xfb\xef\xff')
4 In [26]: base64.urlsafe_b64decode(bus)
5 Out[26]: b'i\xb7\x1d\xfb\xef\xff'
6 In [27]: bus
7 Out[27]: b'abcd--__'
```

Base64 是一同通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64 适用于小段内容的编码，比如数字证书签名，Cookie 的内容等。

由于 = 字符也可能出现在 Base64 编码中，但 = 用在 URL，Cookie 里面会造成歧义，所以很多 Base64 编码后会把 = 去掉：

```
1 # 标准Base64:
2 'abcd' -> 'YWJjZA=='
3 # 自动去掉=:
4 'abcd' -> 'YWJjZA'
```

去掉 = 后怎么解码呢？因为 Base64 是把 3 个字节变成 4 个字节，所以，Base64 编码的长度永远是 4 的倍数，因此，需要加上 = 吧 Base64 字符串长度变为 4 的倍数就可以正常解码了。

6.8 struct

6.9 hashlib

Python 中的 hashlib 提供了常见的摘要算法，如 MD5 和 SHA1 等等。

什么是摘要算法？摘要算法又称哈希算法，散列算法。他通过一个函数，吧任意长度的数据转换为一个长度固定的数据串（通常 16 进制的字符串表示）。

举个例子，你写了一篇文章，内容是一个字符串 'how to use python hashlib - by Michael'，并附上这篇文章的摘要是 '2d73d4f15c0db7f5ecb321b6a65e5d6d'。如果有人篡改了你的文章，并发表为 'how to use python hashlib - by Bob'，你可以一下子指出 Bob 篡改了你的文章，因为根据 'how to use python hashlib - by Bob' 计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数 `f()` 对任意长度的数据 `data` 计算出固定长度的摘要 `digest`，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算 `f(data)` 很容易，但通过 `digest` 反推 `data` 却非常困难。而且，对原始数据做一个 bit 的修改，都会导致计算出的摘要完全不同。

以常见的 MD5 算法为例，计算出一个字符串的 MD5 值：

```

1 In [29]: import hashlib
2 In [30]: md5 = hashlib.md5()
3 In [31]: md5.update('how to use md5 in python hashlib?'.encode('utf-8'))
4 In [32]: print(md5.hexdigest())
5 d26a53750bc40b38b65a520292f69306

```

如果数据量很大，可以分快多次调用 `update()`，最后计算的结果是一样的：

```

1 In [44]: import hashlib
2 In [46]: md5 = hashlib.md5()
3 In [47]: md5.update('how to use md5 in '.encode('utf-8'))
4 In [48]: md5.update('python hashlib?'.encode('utf-8'))
5 In [49]: print(md5.hexdigest())
6 d26a53750bc40b38b65a520292f69306

```

MD5 是最常见的摘要算法，速度很快，生成的结果是固定的 128bit 字节，通常一个 32 位或者 16 进制字符串表示/另一种常见的摘要算法是 SHA1，调用 SHA1 和调用 MD5 完全类似：

```

1 In [53]: sha1 = hashlib.sha1()
2 In [55]: sha1.update('how to use sha1 in '.encode('utf-8'))
3 In [57]: sha1.update('python hashlib?'.encode('utf-8'))
4 In [58]: print(sha1.hexdigest())
5 2c76b57293ce30acef38d98f6046927161b46a44

```

SHA1 的结果是 160bit 字节，通常用一个 40 位的 16 进制表示。比 SHA1 更安全的算法是 SHA256 和 SHA512，不过更安全的算法不仅越慢，而且摘要长度更长。有没有可能两个不同的数据通过某个摘要算法的到了相同的摘要？有可能，因为任何摘要算法都是吧无限多的数据集合映射到一个有限的集合中。这种情况成为碰撞，比如 Bob 试图根据你的摘要算打反推出一篇文章 ‘how to learning hashlib in python - by Bod’，并且这篇文章的摘要恰好和你的文章完全一致，这种情况并非不可能出现，但是非常困难。

6.10 itertools

Python 的内建模块 `itertools` 提供了非常有用的用于操作迭代对象的函数。首先我们看 `itertools` 提供的无限迭代器。

```
1 import itertools
2 naturals = itertools.count(1)
3 for i in naturals:
4     print(i)
```

因为 `count()` 会创建一个无限的迭代器。所以上述代码会一直打印。

6.10.1 cycle

`cycle()` 会把传入的一个序列无限的重复下去:

```
1 import itertools
2 cs = itertools.cycle('ABC')
3 for i in cs:
4     print(i)
```

无限重复打印'A','B','C'。

`repeat()` 负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复的次数:

```
1 In [63]: ns = itertools.repeat('Ab', 3)
2 In [64]: for i in ns:
3 ...:     print(i)
4 ...:
5 Ab
6 Ab
7 Ab
```

无限序列只有在 `for` 迭代时才会无限的迭代下去，如果只是创建一个迭代对象，他不会事先把无限个元素生成出来，是时尚也不可能在内存中创建无限多个元素/无限序列虽然可以无限迭代下去，但是通常我们会通过 `takewhile()` 等函数根据条件判断截取一个有限的序列:

```
1 n [65]: naturals = itertools.count(2)
2 In [66]: ns = itertools.takewhile(lambda x: x < 10, naturals)
3 In [68]: ns
4 Out[68]: <itertools.takewhile at 0x7f306c12fdc8>
5 In [69]: list(ns)
6 Out[69]: [2, 3, 4, 5, 6, 7, 8, 9]
```

6.10.2 chain()

chain() 可以把一组迭代对象串联起来，形成一个更大的迭代器：

```

1 In [71]: for c in itertools.chain('ABC', 'XYZ'):
2     ...:     print(c)
3     ...:
4     A
5     B
6     C
7     X
8     Y
9     Z
10
11 \end{pyhton}
12 \subsection{groupby()}
13 groupby() 把迭代器中相邻的重复元素挑出来放在一起：
14 \begin{python}
15 [74]: for key, group in itertools.groupby('AAABBCCCAA'):
16     ...:     print(key, list(group))
17     ...:
18 A ['A', 'A', 'A']
19 B ['B', 'B']
20 C ['C', 'C', 'C']
21 A ['A', 'A']

```

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组，而函数返回值作为组的 key。如果我们要忽略大小写分组，就可以让元素'A' 和'a' 都返回相同的 key：

```

1 In [76]: for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.upper()):
2     ...:
3         print(key, list(group))
4
5 A ['A', 'a', 'a']
6 B ['B', 'B', 'b']
7 C ['c', 'C']
8 A ['A', 'A', 'a']

```

6.11 contextlib

在 Python 中，读写文件时必须在使用完成后正确关闭它们。正确的关闭文件资源的方法是使用 try...finally

```

1  try:
2      f = open('/path/to/file', 'r')
3      f.read()
4  finally:
5      if f:
6          f.close()

```

写 try...finally 非常繁琐。Python 的 with 语句允许我们非常方便的使用资源，而不必担心没有关闭，所以上面的代码可以简化为：

```

1  with open('path/to/file', 'r') as f:
2      f.read()

```

并不是只有 open() 函数返回的 fp 对象才能使用 with 语句。实际上，任何对象，只要实现了正确的上下文管理就能用于 with 语句。

实现上下文管理通过 `__enter__` 和 `__exit__` 这两个方法实现。例如，下面的 class 实现了这种方法：

```

1  class Query(object):
2      def __init__(self, name):
3          self.name = name
4      def __enter__(self):
5          print('begin')
6          return self
7      def __exit__(self, exc_type, exc_value, traceback):
8          if exc_type:
9              print('Error')
10         else:
11             print('End')
12     def query(self):
13         print('Query info about %s ...' % self.name)
14 with Query('Bob') as q:
15     q.query()

```

编写 `__enter__` 和 `__exit__` 仍然很繁琐，因此 Python 的标准库 contextlib 提供了更简单的写法，上面的代码可改写为如下：

```

1
2  from contextlib import contextmanager

```

```

3 class Query(object):
4     def __init__(self, name):
5         self.name = name
6     def query(self):
7         print('Query info about %s ... %s' % self.name)
8     @contextmanager
9     def create_query(name):
10        print('Begin')
11        q = Query(name)
12        yield q
13        print('End')
14
15 with create_query('Bob') as q:
16     q.query()

```

@contextmanager 这个 decorator 接受一个 generator，用 yield 语句把 with...as var 把变量输出出去，然后，with 语句就能正常工作的。很多时候，我们希望在某段代码执行前后自动执行特定代码，可以用 @contextmanager 实现。例如：

```

1 @contextmanager
2 def tag(name):
3     print("<%s>" % name)
4     yield
5     print("</%s>" % name)
6 with tag("h1"):
7     print("hello")
8     print("world")

```

上述代码的执行结果为：

```

1 <h1>
2 hello
3 world
4 </h1>

```

代码的执行顺序为：

1. with 语句首先执行 yield 之前的语句，因此打印出 <h1>;
2. yield 调用会执行 with 语句内所有语句，因此打印出 hello 和 word。
3. 最后执行 yield 之后的语句，打印出 </h1>

因此 contextmanager 让我们通过编写 generator 简化上下文管理。closing 如果一个对象没有实现上下文，我们就不能把它用于 with 语句。这时候可以调用 closeing() 吧该对象变为上下文对象。例如，用 with 语句使用 urlopen():

```
1 from contextlib import closing
2 from urllib.request import urlopen
3 with closing(urlopen('https://www.python.org')) as page:
4     for line in page:
5         print(line)
```

closing 也是一个经过 contextmanager 装饰的 generator，这个 generator 编写起来其实十分简单：

```
1 @contextmanager
2 def closing(thing):
3     try:
4         yield thing
5     finally:
6         thing.close()
```

6.12 XML

6.13 HTMLParser

6.14 ZipFile

6.15 url

6.15.1 urllib.request

6.16 requests

6.16.1 发送请求

```

1 import requests
2 r = requests.get('https://github.com/timeline.json')
3 r = requests

```

6.16.2 requests 库的 7 个主要方法

requests.request()	够找一个请求，支持一下各方法的基础方法
requests.get()	获取 HTML 网页的主要方法，对应于 HTTP 的 GET
requests.head()	获取 HTML 网页头信息的方法，对应于 HTTP 的 HEAD
requests.post()	向 HTML 网页提交 POST 请求的方法，对应于 HTTP 的 POST
requests.put()	像 HTML 网页提交 PUT 请求的方法，对应于 HTTP 的 PUT
requests.patch()	像 HTML 网页提交局部修改请求，对应于 HTMP 的 PATCH
requests.delete()	像 HTML 网页提交删除请求，对应于 HTTP 的 DELETE

requests.get(url,params=None,**kwargs)

- url: 想要获取的网页的 url 链接。
- params:url 中额外的参数，字典或字节流格式，可选
- **kwargs:12 个控制访问的参数。

6.16.3 request 对象的属性

属性	说明
r.status_code	HTTP 请求的返回状态，200 表示连接成功，404 表示失败
r.text	HTTP 响应内容的字符串形式，即，url 对应的页面内容
r.encoding	从 HTTP header 中猜测响应的内容编码方式
r.apparent_encoding	从内容分析出的响应内容编码方式（备选编码方式）
r.content	HTTP 响应内容的二进制形式

6.16.4 理解 encoding 和 apparent_encoding

r.encoding: 从 HTTp header 中猜测的响应内容编码方式,如果 header 中不存在 charset, 则认为编码为 ISO-8859-1 r.text 根据 r.encoding 显示网页内容

r.apparent_encoding: 根据网页内容分析出的编码方式, 可以看做是 r.encoding 的备选。

6.16.5 理解 Requests 库的异常

异常	说明
requests.ConnectionError	网络链接错误异常, 如 DNS 查询时白, 拒绝链接
requests.HEEPError	HTTP 错误异常
requests.URLRequired	URL 缺失异常
requests.TooManyRedirects	超过最大重定向次数, 产生重定向异常
requests.ConnectTimeout	连接远程服务器超时异常
requests.Timeout	请求 URL 超时, 产生超时异常
requests.raise_for_status()	如果不是 200, 产生异常, requests.HTTPError

r.raise_for_status() 方法在内部判断 r.status_code 是否等于 200, 不需要额外加 if 语句, 该语句便于利用 try-except 进行异常处理。

6.16.6 HTTP 协议

HTTP:Hypertext Transfer Protocol 超文本传输协议。
HTTP 是一句”请求与响应”模式的, 武装到的应用层协议, HTTP 协议采用 URL 定位网络资源的标志, URL 格式如下:

```
http://host[:port][path]
host: 合法的 Internet 主机域名和 IP 地址。
port: 端口号, 缺省端口为 80
path: 请求资源的路径
```

HTTP 协议对资源的操作

方法	说明
GET	请求获取 url 位置的资源
HEAD	请求获取 URL 位置资源的响应消息报告，即获得该资源的头部信息
POST	请求像 URL 位置的资源后附加新的数据
PUT	请求 URL 位置存储一个资源，覆盖原 URL 位置的资源
PATCH	请求局部更新 URL 位置的资源，即改变该处资源的部分内容
DELETE	请求删除 URL 位置存储的资源

head 方法的使用

```

1 In [3]: r = requests.head('http://www.baidu.com')
2 In [4]: r
3 Out[4]: <Response [200]>
4 In [5]: r.headers
5 Out[5]: {'Server': 'bfe/1.0.8.18', 'Date': 'Tue, 08 Aug 2017 11:46:32 GMT', 'Content-Type': 'text/html', 'Last-Modified': 'Mon, 13 Jun 2016 02:50:04 GMT', 'Connection': 'Keep-Alive', 'Cache-Control': 'private, no-cache, no-store, proxy-revalidate, no-transform', 'Pragma': 'no-cache', 'Content-Encoding': 'gzip'}
```

post 方法的使用 (像 URLPOST 一个表单，自动编码为 form)

```

1 In [10]: payload = {'key1': 'value1', 'key2': 'value2'}
2 In [11]: r = requests.post('http://httpbin.org/post', data=payload)
3 In [12]: print(r.text)
4 {
5     "form": {"key2": "value2",
6             "key1": "value1"
7     }
8 }
```

put 方法

```

1 In [18]: r = requests.put('http://httpbin.org/put', data = payload)
2
3 In [19]: print(r.text)
4 {
5     "args": [],
6     "data": "",
7     "files": [],
```

```

8   "form": {
9     "key1": "value1",
10    "key2": "value2"
11  },
12  "headers": {
13    "Accept": "*/*",
14    "Accept-Encoding": "gzip, deflate",
15    "Connection": "close",
16    "Content-Length": "23",
17    "Content-Type": "application/x-www-form-urlencoded",
18    "Host": "httpbin.org",
19    "User-Agent": "python-requests/2.18.3"
20  },
21  "json": null,
22  "origin": "210.47.0.232",
23  "url": "http://httpbin.org/put"
24 }

```

requests.request(method,url,**kwargs)

**kwargs 控制访问参数

params	字典或字节序列，作为参数增加到 url 中
data	字典，字节序列或文件对象，作为 Request 的内容
json	JSON 格式的数据，作为 Request 的内容
header	字典,HTTP 定制头
cookies	字典或 CookieJar，Request 中的 cokkie
auth	元组，支持 HTTP 认证功能
file	字典类型，传输文件
timeout	设定草食时间，s 为单位。
proxies	字典类型，设定访问代理服务器，可以增加登录认证
allow_redirects	:True/False, 默认为 True, 重定向开关
stream	True/False, 默认为 True, 获取内容立即下载开关
verify	True/False, 默认为 True, 认证 SSL 整数开关
cert	本地 SSL 整数路径

Chapter 7

Bazel

7.1 Bazel start

7.1.1 用工作空间

所有的共建操作在你的文件系统上包含有所有构建的软件的源代码目录，正如 build 输出的符号连接的目录（例如 `bazel-bin` 和 `bazel-out`），构建发生在 `workspace`。`workspace` 的位置目录不重要，但是必须包含一个叫最哦 `WORKSPACE` 的顶层目录。一个可用的 `workspace`。`WORKSPACE` 文件能被用来查询构建输出需要的外部依赖。一个 `workspace` 可以在多个项目中共享。

```
1 touch WORKSPACE
```


Chapter 8

Tensorflow 技巧

8.1 文件读取

Chapter 9

Tensorflow API

9.1 tf.app.flags

9.1.1 DEFINE_boolean

DEFINE_boolean(flag_name,default_value,docstring): 定义一个'boolean'类型的flag。

- flag_name: flag 的名字，是一个字符串。
- default_value: flag 应被看作一个 boolean 的默认值。
- docstring: 用 flag 的一个帮助信息。

9.1.2 DEFINE_boolean

: 定义一个'boolean'类型的flag。

- flag_name: flag 的名字，是一个字符串。
- flag_default_value: 默认的 boolean 类型的值。
- docstring: 用 flag 的一个有用的帮助信息。

9.1.3 DEFINE_float

: 定义一个浮点数类型的 flag。

- flag_name: 作为 flag 的名字，应该是字符串。
- default_value: flag 的默认值，应该是浮点数。
- docstring: 用 flag 的一个有用的帮助信息。

9.1.4 DEFINE_integer

: 定义一个整数的 flag。

- flag_name:flag 的名字，应该是字符串。
- default_value:flag 的默认值，应该是一个整数。
- docstring: 用 flag 的一个有用的帮助信息。

9.1.5 DEFINE_string

: 定义一个字符串的 flag。

- flag 的名字，应该是字符串。
- default_value:flag 的默认值，应该是字符串。
- docstring: 用 flag 的一个有用的帮助信息。

9.1.6 tf.gather

```

1 gather(
2     params,
3     indices,
4     validate_indices=None,
5     name=None,
6     axis=0
7 )

```

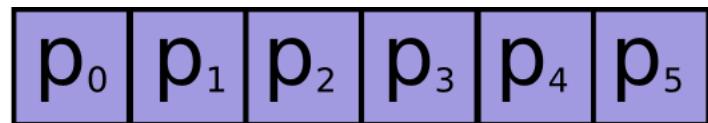
根据 indices 从 params 和 axis 轴获取 slices。indices 必须是任何维度 (通常是 0 维或者一维) 的一个整数 tensor 生成输出 tensor 的形状 params.shape[:axis]+indices.shape+params.shape[axis+1:]，这里

```

1 # Scalar indices (output is rank(params) - 1).
2 output[a_0, ..., a_n, b_0, ..., b_n] =
3 params[a_0, ..., a_n, indices, b_0, ..., b_n]
4 # Vector indices (output is rank(params)).
5 output[a_0, ..., a_n, i, b_0, ..., b_n] =
6 params[a_0, ..., a_n, indices[i], b_0, ..., b_n]
7 # Higher rank indices (output is rank(params) + rank(indices) - 1).
8 output[a_0, ..., a_n, i, ..., j, b_0, ..., b_n] =
9 params[a_0, ..., a_n, indices[i, ..., j], b_0, ..., b_n]

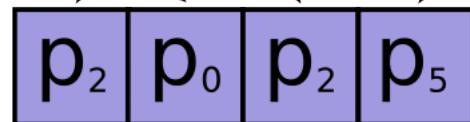
```

params



indices

[2, 0, 2, 5]



参数:

- params: 一个 Tensor, 从它那里收集值, rank 必须是 axis+1
- indices: 一个 tensor。必须是 int32,int64. 索引 tensor, 必须是在 [0,params.shape[axis])
- axis: 一个 Tensor, 必须是下面的数据类型:int32,int64.params 的轴, 从 params 中获得收集索引, 默认是第一维, 支持负索引。
- name: 操作的名字
- 返回: 一个 tensor, 和 params 有相同的数据类型, 通过给定 indeces 从 params 收集, 形状为 [params.shape[:axis]+indices.shape+params.shape[axis+1:]

例子:

```

1 import tensorflow as tf
2 a = tf.constant([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
3 b = tf.constant([1,0,2])
4 g = tf.gather(a,b)
5 with tf.Session() as sess:
6     print(sess.run(g))#获取第一轴[1,0,2]行的数据

```

输出:

```

1 [[ 5   6   7   8]
2  [ 1   2   3   4]
3  [ 9  10  11  12]]

```

```

1 placeholder(
2     dtype ,
3     shape=None ,
4     name=None
5 )

```

为 placeholder 输入需要被插入的 tensor。如果这个 tensor 被计算将产生错误。他的值必须通过 feed_dict 选项在 Session.run(), Tensor.eval() 或者是 Operation.eval() 输入。

```

1 x = tf.placeholder(tf.float32 , shape=(1024 , 1024))
2 y = tf.matmul(x , x)
3 with tf.Session() as sess:
4     print(sess.run(y)) # ERROR: will fail because x was not fed .
5     rand_array = np.random.rand(1024 , 1024)
6     print(sess.run(y , feed_dict={x: rand_array})) # Will succeed .

```

参数:

- dtype: 被输入的 tensor 元素的数据类型。
- shape: 被 fed 的 tensor 的形状, 如果形状没有被指定, 你可以输入任何形状的 tensor。
- : 操作的名字
- Return 一个可能被用作输入数据处理的 Tensor, 值不能被直接计算。

9.1.8 tf.squeeze

tf.squeeze(input,axis=None,name=None,squeeze_dims=None) 说明: 从指定的 Tensor 中移除 1 维度。

- input:tensor, 输入 Tensor。
- axis: 列表, 指定需要移除的位置的列表, 默认为空列表 [], 索引从 0 开始 squeeze 不为 1 的索引会报错。
- name: 操作的名字
- squeeze_dims: 否决当前轴的参数。
- 返回一个 Tensor, 形状和 input 相同, 包含和 input 相同的数据, 但是不包含有 1 的元素。
- 异常: squeeze_dims 和 axis 同时指定时会有 ValueError。

```
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t)) ==> [2, 3]
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t, [2, 4])) ==> [1, 2, 3, 1]
```

9.1.9 tf.metrics

accuracy(labels,predictions,weights=none,metrics_collections,updates_collections=none,name=None)

- labels:tensor, 和 predictions 的形状相同, 代表真实值。
- predictions:tensor, 代表预测值。
- weights:tensor, rank 可以为 0 或者 labels 的 rank, 必须能和 label 广播 (所有的维度必须是 1, 或者和 labels 维度相同)
- metrics_collection:accuracy 应该被增加的一个 collection 列表选项。
- update_collections:update_op 应该添加的选项列表。
- name:variable_scope 名字选项。
- accuracy: 返回值 tensor, 代表精度, 总共预测对的和总数的商。
- update_op: 返回值适当增加 total 和 count 变量和 accuracy 匹配。
- valueerror: 异常如果 predictions 和 labels 有不同的形状, 或者 weight 不是 none 它的形状不合 prediction 匹配, 或者 metrics_collections 会哦这 updates_collections 不是一个 list 或者 tuple。

9.1.10 tf.stack

stack(values,axis=0,name='stack'): stack 一个 n 维 tensor 为 n+1 维 tensor。给定一个长度为 N 的形状为 (A,B,C) 的 tensor, 如果 axis==0 输出 tensor 的形状为 (N,A,B,C), 如果 axis==1, 输出 tensor 的形状为 (A,N,B,C) # 'x' is [1,4]

```
# 'y' is [3,6]
# 'z' is [3,6]
stack([x,y,z]) ==> [[1,4],[2,5],[3,6]]
stack(x,y,z,axis=1) ==> [[1,2,3],[4,5,6]]
tf.stack([x,y,z]) = np.asarray([x,y,z])
输出参数:
```

- 一个 Tensor 列表。
- 整数， 默认为 0, 支持负坐标。
- 操作的名字。

§ 一个 stack 的 Tensor。

§ ValueError: 如果 axis 超过 $[-(R+1), R+1]$

Example

```

1 import tensorflow as tf
2 x = tf.constant([1,4])
3 y = tf.constant([2,5])
4 z = tf.constant([3,6])
5 r1 = tf.stack([x,y,z])
6 r2 = tf.stack([x,y,z],axis=1)
7 with tf.Session() as sess:
8     print(sess.run(r1).shape)
9     print(sess.run(r2).shape)

```

9.1.11 tf.reshape

`tf.reshape(tensor, shape, name=None)`

- Tensor: 一个 Tensor。
- shape: 一个列表, 数值类型为 int32 或者时 int64
- name: 操作的名字。

S 指定形状的 Tensor。

```

1 import tensorflow as tf
2 a = tf.linspace(0.,9.,10)
3 b = tf.reshape(a,[2,5])
4 with tf.Session() as sess:
5     a = sess.run(a)
6     b = sess.run(b)
7     print(a.shape)
8     print(b.shape)

```

9.1.12 tf.random_crop

```
1 random_crop(  
2     value ,  
3     size ,  
4     seed=None ,  
5     name=None  
6 )
```

从 value tensor 随机剪裁一块指定 size 的 tensor, 要求 value.shape 大于参数 size 的值。如果一维不剪切需要传递完整的 size, 例如 RGB 图像可以用 size = [crop_height,crop_width,3] 剪切。

输入参数:

- value: 需要被剪切的 tensor
- size: 和 value 有相同 rank 的一维 tensor。
- seed: Python 整数, 用于创建一个随机种子。
- name: 操作的名字。

Retuen 一个剪切的和 value 有相同的 rank, 形状为 size 的 tensor。

9.1.13 tf.random_gamma

```
1 random_gamma(  
2     shape ,  
3     alpha ,  
4     beta=None ,  
5     dtype=tf.float32 ,  
6     seed=None ,  
7     name=None  
8 )
```

从指定 [Gamma 分布](#) 中获得 shape 样本, alpha 是形状参数, beta 是反向放大参数。

输入参数:

- shape: 一个一维整数 Tensor 或者 Python 数组, 指定 alpha, beta 参数的输出采样数据。
- alpha: 一个 tensor 或者 python 值或者 N 维 dtype 数据类型。alpha 提供形状参数, 必须和 beta 广播。

- beta: 一个 tensor 或 python 值或者 dtype 类型的 n 维数组。默认为 1. beta 提供 gamma 分布的反向放大参数。必须和 alpha 广播。
- dtype:alpha,beta,output 的输出:float16,float32 或者 float64。
- seed: 一个 Python 整数，用于创建随机种子。
- name: 操作的选项。

Returns : samples: 数据类型为 dtype 的一个形状为 tf.concat(shape,tf.shape(alpha+beta)) 的 tensor。

例子:

```

1 samples = tf.random_gamma([10],[0.5,1.5])#形状为10, alpha为0.5, 1.5, 采样输出形状
   为[10,2]
2 samples = tf.random_gamma([7, 5], [0.5, 1.5]) # 采样输出形状为[7, 5, 2], where
   each slice [:, :, 0] and [:, :, 1] # 代
   表从两个分布采样的$7\items5$。
3 samples = tf.random_gamma([30], [[1.],[3.],[5.]], beta=[[3., 4.]]) #采样输出形状
   [30, 3, 2], 每$2\times2$（广播运算）30个
   采样点

```

9.1.14 tf.random_normal

```

1 random_normal(
2     shape,
3     mean=0.0,
4     stddev=1.0,
5     dtype=tf.float32,
6     seed=None,
7     name=None
8 )

```

输出正态分布随机值。

输入参数:

- shape: 一个一维整数 tensor 或者 Python 数组。表示输出 tensor 的形状。
- mean: 一个 0 维 tensor 和 dtype 的 Python 值，正态分布的均值。
- stddev: 一个 0 维 tensor 和 dtype 的 Python 值，正态分布的标准差。
- dtype: 输出数据类型。

- seed: 一个整数，用于创建一个随机数种子。
- name: 操作的名字。

Returns 制订形状的正态分布值。

9.1.15 tf.random_normal_initializer

继承于[Initializer](#) 别名:

- Class `tf.contrib.keras.initializers.RandomNormal`
- Class `tf.random_normal_initializer`

输入参数:

- mean: 一个 python 标量或者标量 tensor。生成随机值的均值。
- stddev: 一个 python 标量或者标量 tensor。标生成随机值的标准差。
- seed:python 整数，用于创建随机种子。
- dtypes: 数据类型，仅仅支持浮点数类型。

方法:

- `__init__`

```
1 __init__(  
2     mean=0.0,  
3     stddev=1.0,  
4     seed=None,  
5     dtype=tf.float32  
6 )
```

- `__call__`

```
1 __call__(  
2     shape,  
3     dtype=None,  
4     partition_info=None  
5 )
```

- `from_config`: 从配置字典实例化 initializer。

```
1 from_config(   
2     cls,   
3     config  
4 )
```

例如：

```
1 initializer = RandomUniform(-1, 1)
2 config = initializer.get_config()
3 initializer = RandomUniform.from_config(config)
```

- `get_config()`

9.1.16 tf.random_possion

```
1 random_poisson(   
2     lam ,   
3     shape ,   
4     dtype=tf.float32 ,   
5     seed=None ,   
6     name=None   
7 )
```

从 Poisson 分布中获取指定形状的样本，`lam` 是描述分布的参数。

输入参数:

- lam: 一个 Tensor 或者 Python 值或者 dtype 指定的 N 维数组。lam 是 Poisson 分布的参数。
 - shape: 一个一维整数 tensor 或者 python 数组，输出每个 rate 指定的参数的形状。
 - dtype: lam 和输出的数据类型:float16,float32,float64.
 - seed: Python 整数，用于创建一个分布的随机种子。
 - name: 操作的名字。

Returns : 形状为 `tf.concat(shape, tf.shape(lam))` 的 tensor (值的数据类型如 `dtype` 指定)。

例子：

```
1 samples = tf.random_poisson([0.5, 1.5], [10]) # 采样输出形状为[10, 2], 切片输出  
2 [ :, 0] and [ :, 1] 代表不同分布的数据
```

```

3 samples = tf.random_poisson([12.2, 3.3], [7, 5]) # 采样输出形状[7, 5, 2], chide
切片[:, :, 0] 和[:, :, 1] 代表 $7\times 5$ 从两个分布的采样输出。

```

9.1.17 random_shuffle

```

1 random_shuffle(
2     value,
3     seed=None,
4     name=None
5 )

```

沿着 tensor 的一维随机打乱数据。像 value[j] 映射一个 tensor 到一个数出 output[i]。例如

```

1 [[1, 2],           [[5, 6],
2  [3, 4], ==>   [1, 2],
3  [5, 6]]           [3, 4]]

```

输入参数:

- value: 一个应该被打乱的值。
- seed: 一个 python 正数用于创建随机数种子。
- name: 操作的名字。

Returns 一个指定类型的值和形状，沿着某一维读被打乱的 tensor。

9.1.18 tf.random_uniform

```

1 random_uniform(
2     shape,
3     minval=0,
4     maxval=None,
5     dtype=tf.float32,
6     seed=None,
7     name=None
8 )

```

输出均匀分布的随机值，生成的值的范围在 [minval,maxval)，下界 minval 包含在范围内，上界不再。对于浮点数默认范围是 [0,1)。至少 maxval 必须被明确指定。在整数情况下，会有一些轻微的偏移，除非 maxval-minval 是 2 的次幂。偏移是 maxval-minval 比较小的值相对于输出范围 ($2^{32}, 2^{64}$) 很小。

输入参数:

- shape: 一维整数 tensor 或者 Python 数组，和数出 tensor 的形状相同。
- minval:0 维 tensor 或者 dtype 指定的 python 值，生成随机数的下界，默认为 0,。
- maxval:0 维 tensor 或者 dtype 指定的 python 值。生成随机数的上界，dtype 是浮点数时，默认为 1。
- dtype: 输出的类型:'float16,float32,float64,int32,int64'。
- seed: 一个 Python 整数。用于创建分布的随机数种子。
- 操作的名字。

Return 返回指定形状的均匀分布值。

Raise ValueError: 如果 dtype 是整数并且 maxval 没有被指定。

9.1.19 tf.random_uniform_initializer

一个继承自 Initializer 的类。

别名:

- tf.contrib.keras.initializers.RandomUniform
- tf.random_uniform_initializer

生成均匀分布 tensor 的 initializer。

输入参数:

- minval:0 维 tensor 或者 dtype 指定的 python 值，生成随机数的下界，默认为 0,。
- maxval:0 维 tensor 或者 dtype 指定的 python 值。生成随机数的上界，dtype 是浮点数时，默认为 1。
- seed: 一个 Python 整数。用于创建分布的随机数种子。
- dtype: 输出的类型:'float16,float32,float64,int32,int64'。

方法:

```

1     minval=0,
2     maxval=None,
3     seed=None,
4     dtype=tf.float32
5 )
```

- `__call__`

```

1 __call__(
2     shape ,
3     dtype=None ,
4     partition_info=None
5 )

```

- `from_config`

```

1 from_config
2
3 from_config(
4     cls ,
5     config
6 )

```

- `get_config`

9.1.20 `tf.one_hot`

```

1 ne_hot(
2     indices ,
3     depth ,
4     on_value=None ,
5     off_value=None ,
6     axis=None ,
7     dtype=None ,
8     name=None
9 )

```

返回一个 One-hot tensor。indices 表示的下表的值为 on_value, 所有其它值为 off_value。

如果 on_value 不提供, 默认为 dtype 下的 1。off_value 不提供, 默认值为 0, 如果输入 Incices rank 是 N, 输出 rank 将是 N+1。新的 axis 将在 axis 创建。, 如果 indices 是一个标量, 输出形状将是一个 depth 长度的向量, 如果 indices 是亿个长度为 features 向量输出形状将为:

```

1 feature x depth ifn axis == -1
2 depth x features if axis == 0

```

如果 indices 是一个矩阵, 形状为 [batch,features], 输出形状将为:

```

1 batch x features x depth if axis == -1
2 batch x depth x features if axis == 1

```

```
3 depth x batch x features if axis == 0
```

如果 dtype 没有被提供，他将假设 on_value 或 off_value 的数据类型，如果一个或者倍多的值呗传递，，如果没有 on_value,off_value, 或者 dtype 被提供， dtype 将默认 tf.float32。例子：

假设：

```
1 indices = [0, 2, -1, 1]
2   depth = 3
3   on_value = 5.0
4   off_value = 0.0
5   axis = -1
```

输出形状为 $[4 \times 3]$ 输出：

```
1 output =
2   [5.0 0.0 0.0] // one_hot(0)
3   [0.0 0.0 5.0] // one_hot(2)
4   [0.0 0.0 0.0] // one_hot(-1)
5   [0.0 5.0 0.0] // one_hot(1)
```

假设：

```
1 indices = [[0, 2], [1, -1]]
2   depth = 3
3   on_value = 1.0
4   off_value = 0.0
5   axis = -1
```

输出

```
1 output =
2   [
3     [1.0, 0.0, 0.0] // one_hot(0)
4     [0.0, 0.0, 1.0] // one_hot(2)
5   [
6     [0.0, 1.0, 0.0] // one_hot(1)
7     [0.0, 0.0, 0.0] // one_hot(-1)
8   ]
```

用默认 on_value 和 off_value:

```
1 indices = [0, 1, 2]
2   depth = 3
```

输出

```

1   output =
2     [[1., 0., 0.],
3      [0., 1., 0.],
4      [0., 0., 1.]]

```

参数:

- indices: 一个索引的 Tensor。
- depth: 一个定义 one hot 维度的深度的标量。
- on_value: 制定 Indices[j]=i(填入 i), 默认为 1。
- off_value: 一个标量当 Indices[j]!=i(默认为 0)。
- axis: 填入的轴, 默认为-1。
- dtype: 输出 tensor 的数据类型。

Returns :one-hot tensor。

- Raise
- TypeError: 如果 dtype 和 on_value,off_value 数据类型不一样。
 - TypeError: 如果 on_value 和 off_value 不合另一个匹配。

9.1.21 tf.unstack

```

1 unstack(
2   value,
3   num=None,
4   axis=0,
5   name='unstack',
6 )

```

unstack rank-R 为 rank-(R-1) 的 tensor。通过 chipping 沿着 axis 从 value unstack num tensor。如果 num 不被指定 (默认) 它从 value 的形状推算。如果 value.shape[axis] 不知道, ValueError 异常。例如一个 tensor 的形状为 (A,B<C<D); 如果 axis ==0 输出第 i 维的切片 value[i,:,:,:], output 的输出每个 tensor 形状为 (B,C,D). (注意沿着维度 unstack) 如果 axis == 1 然后输出第 i 个 tensor 在切片的值 [:,i,:,:] 和每个 output 的每个输出将有形状 (A,C,D)。tf.unstack(x,n)=list(x)

参数:

- value:rank R>0 的 Tensor 能被 unstack。

- num 一个整数。axis 的长度，如果为 None 将自动推断。
- axis: 一个整数。ustack 沿着的轴。默认是第一维，支持负的索引。
- name: 操作的名字。

Returns 从 value unstack 的 Tensor 列表。

Raises :

- ValueError: 如果 num 没有被指定且不能推断出。
- ValueError: 如果 axis 超过 [-R,R)。

9.2 tf.Variable

9.2.1 Variable 类

一个变量通过在图上运行 run() 方法维持变量状态, 你可以构造一个 Variable 类的实例添加到图上。Variable() 够着要求一个初始值, 这个值可以是一个任何类型和形状的 Tensor。初始值定义了变量的形状了类型。在构造后变量的形状和类型就被固定, 值可以通过 assign 方法更改。如果你之后像改变变量的形状可以用 assign 操作设置 validate_shape=False。和任何 Tensor 一样 Variable() 创建的变量可能被用于图中的操作的输入。另外所有的操作重载了 Tensor 类为变量, 因此你可以通过在变量上做算法添加节点到图上。

```

1 import tensorflow as tf
2
3 # Create a variable.
4 w = tf.Variable(<initial-value>, name=<optional-name>)
5
6 # Use the variable in the graph like any Tensor.
7 y = tf.matmul(w, ...another variable or tensor...)
8
9 # The overloaded operators are available too.
10 z = tf.sigmoid(w + y)
11
12 # Assign a new value to the variable with assign() or a related method.
13 w.assign(w + 1.0)
14 w.assign_add(1.0)
```

当你启动图运行操作前变量需要被明确的初始化。你可以通过初始化器操作, 从保存的文件恢复或者仅仅运行一个 assign 操作给变量指定值。事实上, 变量初始化操作仅仅是一个赋值给变量初始值的 assgin 操作。

```

1 # Launch the graph in a session.
2 with tf.Session() as sess:
3     # Run the variable initializer.
4     sess.run(w.initializer)
5     # ...you now can run ops that use the value of 'w'...
```

通常初始化是添加 global_variables_initializer() 操作到图上初始化所有的变量。你然后启动图后运行 Op

```

1 # Add an Op to initialize global variables.
2 init_op = tf.global_variables_initializer()
3
4 # Launch the graph in a session.
```

```

5 with tf.Session() as sess:
6     # Run the Op that initializes global variables.
7     sess.run(init_op)
8     # ... you can now run any Op that uses variable values...

```

如果你创建一个变量的时候它的初始值依赖于另一个变量,用其它的变量的 initialized_value()。确保变量按照正确的顺序初始化。当它们被创建的时候所有的变量被自动地收集在图中。默认,构造体添加变量到图上手机 GraphKeys.GLOBAL_VARIABLES。一个方便的函数是 global_variables() 返回集合里面的内容。

当创建一个机器学习模型的时候区别保持可训练模型参数和其它的变量像用于记录训练步数 global step 变量是很方便的。为了使这个更简单,变量构造体支持 trainable=<bool> 参数,如果为 True,新的变量也被添加到图集合 GraphKeys.TRAINABLE_VARIABLES。方便的函数 trainable_variables() 返回这个集合的内容, Optimizer 类用这个集合作为优化的默认的列表变量。

属性	功能
device	变量所在的设备
dtype	变量的数据类型
graph	变量所在的图
initial_value	用作变量初始值的 tensor
initializer	对于变量的初始化操作
name	变量的名字
op	这个变量的 Operation
shape	这个变量的 tensorShape

9.2.2 方法

__init__

```

1 __init__(
2     initial_value=None,
3     trainable=True,
4     collections=None,
5     validate_shape=True,
6     caching_device=None,
7     name=None,
8     variable_def=None,
9     dtype=None,
10    expected_shape=None,

```

```
11     import_scope=None  
12 )
```

创建一个值为 initial_value 的新的变量，新的变量被添加到 collections 的列表中，默认是 GraphKeys.GLOBAL_VARIABLES。如果 trainable 是 True 变量被添加到集合 GraphKeys.TRAINABLE_VARIABLES。这个构造体创建一个 variable 操作和 assign 操作设置它的初始值。参数：

- initial_value: 一个 Tensor，或者 Python 对象转化为一个 Tensor，是 Variable 的初始值。初始值必须有一个指定的形状厨卫 validate_shape 被设置为 False。没有参数可能被调用，当被调用的时候返回初始值、在这种情况下，dtype 必须被指定（注意 init_ops.py 的初始化函数必须在使用前被限制形状）
- trainable: 如果为真（默认），添加变量到 GraphKeys.TRAINABLE_VARIABLES 集合。这个集合 Optimizer 类用作默认的变量列表集合。
- collections: 集合 keys 的列表，新的变量被添加到这些集合。默认是 GraphKeys.GLOBAL_VARIABLES。
- validate_shape: 如果为 False 允许变量初始化时不指定形状。如果为 True， 默认。initial_value 必须知道。
- caching_device: 描述变量应该被缓存的设备的字符串。默认是 Variable 的设备。如果不是 None，cache 在另一个设备上。通常操作保存变量时缓存在设备上，通过 Switch 复制在不同的条件状态下转换。
- name: 变量的名字，默认为'Variable' 自动设置为独一无二。
- variable_def: VariableDef protocol buffer. 如果不为 None，用它的值重新创建变量对象，访问在图上变量的节点，节点必须存在。图不美改变，variable_def 和其它的参数被相互排斥。
- dtype: 如果设置，initial_value 将被转化为给定的类型。如果设置为 None，数据类型将被保持如果初始值是一个 Tensor) 或者 convert_to_tensor 将转换。
- expected_shape: 一个 TensorShape。如果设置，initial_value 期望的形状。
- import_scope: 选项字符串，Name scope 添加到 Variable 仅仅在 protocol buffer 初始化时使用。

ValueError

- ValueError: 如果 variable_def 和 initial_value 被指定。
- ValueError: 如果初始值没有指定或者没有形状同时 validate_shape 是 True。

`__abs__` 计算 tensor 的绝对值, 给定一个复数 x, 这个操作将返回一个 float32 或者 float64 类型的 Tensor, 它的值是元素的模 ($\sqrt{a^2 + b^2}$)

```
1 # tensor 'x' is [[ -2.25 + 4.75j], [-3.25 + 5.75j]]
2 tf.complex_abs(x) => [5.25594902, 6.60492229]
```

参数:

- x: 一个数据类型为 float32,float64,int32,int64,complex64 或者 complex128 的 Tensor 或者 SparseTensor
- name: 操作的名字

Returns 一个 Tensor 或者相同尺寸和类型的的 SparseTensor 作为 x 的绝对值。注意对于 complex65 或者 complex128 输入, 返回的 Tensor 将是分别是 float32 和 float64。

`__add__`

```
1 __add__(  
2     a,  
3     *args  
4 )
```

返回按元素相加的结果, Add 支持广播运算, AddN 不支持。

参数:

- x: 一个 tensor, 必须是下面的数据类型: half, float32, float64, uint8, int8, int16, int32, int64, complex64, complex128, string。
- y: 一个 Tensor 们必须和 x 的类型相同。
- name: 操作的名字

Returns : 和 x 相同类型的 Tensor。

`__adn__`

```
1 __and__(  
2     a,  
3     *args  
4 )
```

返回 xy 按位想与的值, 注意 LogicalAnd 支持广播。

参数:

- x: 一个 bool 型参数。
- y: 一个 bool 型参数。
- name: 操作的名字。

Returns :bool 型的返回值。

__div__

```

1 __div__(  

2     a,  

3     *args  

4 )

```

用 Python2 的语法除两个数

参数:

- x: 数值类型的分子 Tensor。
- y: 数值类型的分母 Tensor。
- name: 操作的名字

返回:x 除以 y 的商。

__floordiv__

x 和 y 按元素相除截取为接近最小负整数 (c 语言实现, 十分高效)。x 和 y 必须有相同的类型, 结果也有相同的类型。参数:

- x: 实数类型的分子 tensor
- y: 实数类型的分母 tensor
- name: 操作的名字

Returns x/y 截取后的结果

Raises :TypeError(如果输入为复数)。

__ge__

```

1 __ge__(  

2     a,  

3     *args  

4 )

```

按元素返回 $x \geq y$ 的值。GreaterEqual 支持广播。

参数:

- x: 一个 Tensor, 必须是 float32, float64, int32, int64, uint8, int16, int8, uint16, half。
- y: 一个和 x 类型相同的 Tensor。
- name: 操作的名字。

name :bool 型的 tensor。

__getitem__

```

1 __getitem__(  

2     var,  

3     slice_spec  

4 )

```

创建一个 slice helper, 从当前变量的内容中创建一个子 tensor。这个函数被允许赋值给一个 IE 片的范围。这类似于 Python 中的 __setitem__ 函数。然而语法的不同用户可以捕获复制操作来分组或者喘气 sess.run(), 例如:

```

1 import tensorflow as tf  

2 A = tf.Variable([[1,2,3], [4,5,6], [7,8,9]], dtype=tf.float32)  

3 with tf.Session() as sess:  

4     sess.run(tf.global_variables_initializer())  

5     print(sess.run(A[:2, :2])) # => [[1,2], [4,5]]  

6  

7     op = A[:2,:2].assign(22. * tf.ones((2, 2)))  

8     print(sess.run(op)) # => [[22, 22, 3], [22, 22, 6], [7,8,9]]

```

注意复制不支持 NumPy 广播语法。

参数:

- var:op.Variable 对象。
- slice_spec:tensor.__getitem__ 参数

Returns : 合适的 tensor 切片, 基于 slice_spec, 作为一个操作, 这个操作也有 assign() 方法用于生成一个赋值操作。

Raises :

- ValueError: 如果切片的范围是负的大小。

– `TypeError`: 如果切片索引不是整数, 或者省略。

__gt__

```

1  __gt__(  
2      a,  
3      *args  
4  )
```

返回 $x > y$ 的 bool Tensor Greater 支持广播运算。

- `x`: 一个 Tensor, 数据类型为 float32, float64, int32, int64, uint8, int16, int8, uint16, half
- `y`: 一个 Tensor 和 `x` 相同的数据类型。
- 操作的名字

Returns :bool 型的 Tensor。

__invert__

```

1  __invert__(  
2      a,  
3      *args  
4  )
```

按元素返回 `x` 的非。

参数:

- `x`: 一个 bool 型的 Tensor。
- `name`: 操作的名字。

Returns :bool 型的 Tensor。

__iter__

阻止迭代的伪方法, 不要调用。注意我们如果注册 `getitem` 作为重载操作, Python 将尝试在 0 到无穷大迭代, 声明这个方法阻止无意识的行为。异常: `TypeError`: 调用的时候。

__le__

```

1 __le__(  

2     a,  

3     *args  

4 )

```

按元素返回 $x \leq y$ 的值，LessEqual 支持广播。

9.2.3 参数

- x: 一个 Tensor，必须是下面的数据类型:float32, float64, int32, int64, uint8, int16, int8, uint16, half。
- y: 一个 Tensor，和 x 的类型一样。
- name: 操作的名字。

Returns : 返回 bool 型的 Tensor。

____it____

```

1 __lt__(  

2     a,  

3     *args  

4 )

```

an 元素返回 $x < y$ 的值。Less 支持广播运算。

参数:

- x: 一个 Tensor，类型如下:float32, float64, int32, int64, uint8, int16, int8, uint16, half。
- y: 一个 Tensor，必须和 x 的类型相同。
- name: 操作的名字。

Returns :bool 型的 Tensor。

____matmul____

```

1 __matmul__(  

2     a,  

3     *args  

4 )

```

矩阵乘法 $a*b$, 两个矩阵的类型都是 float16, float32, float64, int32, complex64, complex128, 矩阵可以通过设置 flag 为 True 为转置矩阵或者伴随矩阵, 默认 flag 为 False。如果两个矩阵包含一些 0, 为了更高效的乘法设置相应的 a_is_sparse 或者 b_is_sparse。这里默认为 False, 优化仅仅对数据类型为 bfloat16 或者 float32 的二维 tensor。例如:

```

1 # 2-D tensor a
2 a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3]) => [[1. 2. 3.]
3                                         [4. 5. 6.]]
4 # 2-D tensor b
5 b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2]) => [[7. 8.]
6                                         [9. 10.]
7                                         [11. 12.]]
8 c = tf.matmul(a, b) => [[58 64]
9                                         [139 154]]
10
11 # 3-D tensor a
12 a = tf.constant(np.arange(1, 13, dtype=np.int32),
13             shape=[2, 2, 3]) => [[[1. 2. 3.]
14                                         [4. 5. 6.]],
15                                         [[7. 8. 9.]
16                                         [10. 11. 12.]]]
17
18 # 3-D tensor b
19 b = tf.constant(np.arange(13, 25, dtype=np.int32),
20             shape=[2, 3, 2]) => [[[13. 14.]
21                                         [15. 16.]
22                                         [17. 18.]],
23                                         [[19. 20.]
24                                         [21. 22.]
25                                         [23. 24.]]]
26 c = tf.matmul(a, b) => [[[94 100]
27                                         [229 244]],
28                                         [[508 532]
29                                         [697 730]]]
30
31 # Since python >= 3.5 the @ operator is supported (see PEP 465).
32 # In TensorFlow, it simply calls the tf.matmul() function, so the
33 # following lines are equivalent:
34 d = a @ b @ [[10.], [11.]]
35 d = tf.matmul(tf.matmul(a, b), [[10.], [11.]])

```

参数:

- a: 一个 Tensor, 数据类型为 float16, float32, float64, int32, complex64, complex128,

rank > 1

- b: 一个和 a rank 和数据类型相同的 Tensor
- transpose_a: 如果为 True, 乘法前先转置。
- transpose_b: 如果为 True, 乘法前先转置。
- adjoint_a: 如果为 True, 乘法前共轭转置。
- adjoint_b: 如果为 True, 乘法前共轭转置。
- a_is_sparse: 如果为 True, a 被当做稀疏矩阵。
- b_is_sparse: 如果为 True, a 被当做稀疏矩阵。
- name: 操作的名字

Returns : 一个和 a, b 有相同类型的 Tensor, a, b 的乘。矩阵乘。

Raises :ValueError: 如果 transpose_a 和 adjoint_a 或者 transpose_b 和 adjoint_b 都设置为 True。

____mod____

```
1 ____mod__(  
2     a,  
3     *args  
4 )
```

参数:

- x: 一个 Tensor, 数据类型为 int32, int64, float32, float64。
- y: 一个 Tensor, 类型和 x 相同。
- name: 操作的名字

Returns : 和 x 相同类型的 Tensor。

____mul____

```

1   __mul__(  

2       a,  

3       *args  

4   )

```

稀疏和非稀疏乘法。

```

1   __neg__(  

2       a,  

3       *args  

4   )

```

计算元素的负值。

参数:

- x: 一个 Tensor, 必须是 half, float32, float64, int32, int64, complex64, complex128。
- name: 操作的名字。

Returns : 和 x 的类型相同

```

1   __or__(  

2       a,  

3       *args  

4   )

```

LogicalOr 支持广播运算, 输入 x: bool Tensor。y:bool:Tensor,name: 操作的名字, 返回值 bool 的 Tensor。

```

1   __pow__(  

2       a,  

3       *args  

4   )

```

按元素计算 x^y ,x,y:loat32, float64, int32, int64, complex64, or complex128,name: 操作的名字。

```

1   __radd__(  

2       a,  

3       *args  

4   )

```

返回 $x+y$ 按元素相加, 支持广播运算, x,y 是一个 half, float32, float64, uint8, int8, int16, int32, int64, complex64, complex128, string 的 Tensor, name 是操作的名字。

```

1 __rand__(
2     a,
3     *args
4 )

```

x 和 y 按元素像与，LogicalAnd 支持广播，x,y 是 bool 型的 tensor,name 是操作的名字。

```

1 __rdiv__(
2     a,
3     *args
4 )

```

x,y 分别是分子分布 tensor, name 是操作的名字。

```

1 __rfloordiv__(
2     a,
3     *args
4 )

```

x,y 分别是分子分母 tensor,name 是操作的名字。x/y 像 0 或者负整数靠近。__rmatmul__,__rmod__,__rmul__
参照上面的 matmul, mod,rmul,or,pow。

```

1 __rsub__(
2     a,
3     *args
4 )

```

按元素 x-y。__rturediv__,__rxor__(x 异或 y),__sub__。

```

1 assign(
2     value,
3     use_locking=False
4 )

```

赋值新的 value, use_locking: 如果为 True 在副职期间用 locking, 返回赋值后的新值。

```

1 assign_add(
2     delta,
3     use_locking=False
4 )

```

添加 delta 被计算后保持新值。

```

1 assign_sub(
2     delta,
3     use_locking=False
4 )

```

计算减去 delta 后只用新值。count_up_to(limit): 增加这个变量直到达到 limit。

eval(sess=None): 在一个绘画中计算返回这个变量，它不是一个图机构的方法，不增加操作到图上。这是一个方面的方法当启动的图上包含有变量的时候，如果没有 session 传入，用默认的会话。

```

1 assign_sub(
2     delta ,
3     use_locking=False
4 )

```

```

1 from_proto(
2     variable_def ,
3     import_scope=None
4 )
5 Return

```

从 variable_def 创建的一个变量对象。

get_shape(): 变量形状的别名。

initialized_value(): 返回初始化变量后的值。

```

1 # Initialize 'v' with a random tensor.
2 v = tf.Variable(tf.truncated_normal([10, 40]))
3 # Use initialized_value to guarantee that v has been
4 # initialized before its value is used to initialize w.
5 # The random values are picked only once.
6 w = tf.Variable(v.initialized_value() * 2.0)

```

```

1 load(
2     value ,
3     session=None
4 )

```

载入值进变量，写新的值进变量的存储区而不是增加操作到图上，这个方便的方法要求图上的所有变量在图被启动前绘画已经启动否则用默认的会话。

```

1 v = tf.Variable([1, 2])
2 init = tf.global_variables_initializer()
3
4 with tf.Session() as sess:
5     sess.run(init)
6     # Usage passing the session explicitly.
7     v.load([2, 3], sess)
8     print(v.eval(sess)) # prints [2 3]
9     # Usage with the default session. The 'with' block

```

```

10 # above makes 'sess' the default session.
11 v.load([3, 4], sess)
12 print(v.eval()) # prints [3 4]

```

ValueError: Session is not passed and no default session

read_value(): 从当前上下文读入变量返回变量的值，可以不同于 value() 如果他的值在另一个设备上，具有控制依赖等等。返回一个包含变量的 Tensor。

```

1 scatter_sub(
2     sparse_delta,
3     use_locking=False
4 )

```

从变量中减去 IndexedSlices, parse_delta: 从这个变量减去的 IndexedSlices, use_locking: 如果为 True 在操作过程中用 locking。如果 parse_delta 不是一个 IndexedSlices 将出现 ValueError。set_shape(shape): 重载变量的形状。

to_proto(export_scope=None): 转化变量为一个 variableDef protocol buffer。export_scope: 字符串，移除的范围的名字，返回一个 variableDef protocol buffer，如果 Variable 不在指定范围内为 None。

value(): 返回这些变量的快照，你不需要调用这个放放，当所有的操作需要变量的值时通过 convert_to_tensor() 自动调用它。返回一个保持变量值得 Tensor，你不能指定一个新的值给这个 tensor 当它没有引用变量的时候。为了避免复制如果利用的返回值在同一个设备上作为变量，这实际上返回实时的值而不是复制的值。用过使用更新变量。如果使用在不同的设备上使用将得到不同的值，返回包含变量值的 Tensor。

9.3 tf.image

9.3.1 tf.image.decode_gif

```
tf.image.decode_gif(contents,name=None)
```

- contents: 一个字符串 Tensor, GIF 编码的图像。
- name: 操作的名字。
- 返回一个 8 位无符号的 Tensor, 四维形状为 [num_frames,height,width,3], 通道顺序是 RGB。

9.3.2 tf.image.decode_jpeg

```
tf.image.decode_jpeg(contents,channels=None,ratio=None,fancy_upscaling=None,  
try_recover_truncated=None,acceptable_fraction=None,dct_methed=None,name=None)
```

解码 JPEG 编码的图像为无符号的 8 位整型 tensor。

- contents: 一个字符串 tensor, JPEG 编码的图像。
- channels: 一个整数默认为, 0 代表编码图像的通道数 (JPEG 编码的图像), 1 代表灰度图, 3 带秒 RGB 图。
- radio: 一个整数, 默认为 1, 取值可以是 1,2,4,8, 表示缩减图像的比例。
- fancy_upscaling:bool 型, 默认为 True, 表示用慢但是更好的提高色彩浓度。
- try_recover_truncated:bool 型, 默认是 False, 如果时 True 尝试从截断的输入恢复图像。
- acceptable_fraction:float 型, 默认是 1, 可接受的最小的截断输入的因子。
- dct_methed:string 类型, 默认为 “”. 指定一个解压算法, 默认是 “” 由系统自行指定。可用的值有 [“INTEGER_FAST”,“INTEGER_ACCURATE”]
- name: 操作的名字。
- 返回值为一个 8 位无符号整型 Tensor, 3 维形状 [height,width,channels]

9.3.3 tf.image.encode_jpeg

`tf.image.encode_jpeg(image,format=None,quality=None,progressive=None,optimize_size=None,chroma_downsampling=None,density_uint=None,x_density=None,y_density=None,xmp_metadata=None,`

- `image`: 一个 3 维 [height,width,channels], 8 位无符号整型 Tensor。
- `format:string` 类型, 可以为 "", "grayscale", "rgb", 默认为 ""。如果 `format` 没有指定或者不为空字符串, 默认格式从 `image` 的通道中选, 1: 输出灰度图, 3: 输出 RGB 图。
- `quality`: 整型, 默认值为 95, 代表压缩质量值 [0,100], 值越大越好, 单速度越慢。
- `optimize_size:bool` 型, 默认为 False, 如果为 True 用 CPU/RAM 减少尺寸同时保证质量。
- `chroma_downsampling:bool` 型, 默认为 True。
- `density_unit`: 一个字符串, 可以为 "in", "cm", 指定 `x_density` 和 `y_density` in 每 inch 的像素, cm 表示每厘米的像素。
- `x_density`: 一个整数, 默认为 300, 每个 density 单位的水平像素。
- `y_density`: 一个整数, 默认为 6300, 数值方向上每 density 单位的像素。
- `xmp_metadata:string` 类型, 默认为 "", 如果为空, 嵌入 XMP metadata 到图像头部。
- `name`: 操作的名字。
- `name`: 操作的名字。
- 返回 0 维字符串型 JPEG 编码的 Tensor。

9.3.4 tf.image.decode_png

`tf.image.decode_png(contents,channels=None,dtype=None,name=None)` 解码 PNG 编码的图像为 8 位或者 16 位无符号整型 Tensor。

- `contents`: 一个 0 维 PNG 编码的图像的字符串的 Tensor。
- `channels`: 整型默认为 0, 代表解码图像的通道, 0 用 PNG 编码图像数, 1: 代表输出灰度图像。3: 代表输出 RGB 图像。4: 代表输出 RGBA 图像。
- `dtype:tf.DType`, 值可以为 `tf.uint8,tf.uint16`, 默认为 `tf.uint8`。
- `name`: 操作的名字。
- 返回 3 维 [height,width,channels] 的 Tensor。

9.3.5 tf.image.encode_png

```
tf.image.encode_png(image,compression=None,name=None)
```

- 一个 8 位或者 16 位的 3 维 Tensor, 形状为 [height,width,channels]
- compression: 一个整数, 默认为-1, 表示压缩等级。
- name: 操作的名字。
- 返回一个 0 维 string 型的 PNG-encoded 的 Tensor。

9.3.6 tf.image.decode_image

```
tf.image.decode_image(contents,channels=None,name=None)
```

- contents: 0 维编码图像的字符串。
- channels: 整数, 默认为 0, 解码图像的通道数。
- name: 操作的名字。
- 返回 JPEG,PNG 的 8 位无符号的形状为 [height,width,num_channels], GIF 文件的形状为 [num_frames,height,width,3]
- ValueError: 通道数不正确。

9.3.7 tf.image.resize_images

```
tf.image.resize_images(images,size,method=ResizeMethod.BILINEAR,align_corners=False)
```

- images: 形状为 [batch,height,width,channels]4 维 Tensor,3 为 Tensor, 形状为 [height,width,channels]
- size: 一位 32 整型 Tensor 元素为 new_height,new_width, 新的图像尺寸。
- method: ResizeMethod, 默认为 ResizeMethod.BILINEAR
 - ResizeMethod.BILINEAR: 二进制插值。
 - ResizeMethod.NEAREST_NEIGHBOR:
 - ResizeMethod.BICUBIC:
 - ResizeMethod.AREA:
- align_corners:bool 型, 如果为真提取对齐四个角, 默认为 False。

- 异常
 - ValueError: 图像形状和函数要求的不一样。
 - ValueError:size 是不可用的形状或者类型。
 - ValueError: 指定的方法不支持。
- 如果图像时 4 维 [batch,new_height,new_height,channels], 如果图像是 3 维, 形状为 [new_height,new_width,channels]

9.4 layer

9.4.1 tf.layers.average_pooling1d

函数功能: 一维数据的平均池化。

```
1 average_pooling1d(  
2     inputs,  
3     pool_size,  
4     strides,  
5     padding='valid',  
6     data_format='channels_last',  
7     name=None  
8 )
```

参数:

- 池化的 Tensor, rank 必须是 3。
- pool_size: 一个正数或者是一个整数列表或元组, 代表池化窗口大小。
- strides: 一个整数, 指定池化操作的步数。
- padding: 一个字符串, padding 的方法, 可以是'valid' 或者是'same'。
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,length,channels),channels_first 形状为 (batch,channels,length)
- name: 字符串, layer 名字。
- 返回值: 输出 tensor, rank 为 3。

9.4.2 tf.layers.average_pooling2d

函数功能: 二维数据的平均池化

```
1 average_pooling2d(  
2     inputs,  
3     pool_size,  
4     strides,  
5     padding='valid',  
6     data_format='channels_last',  
7     name=None  
8 )
```

参数:

- inputs: 池化的 Tensor, rank 必须为 4
- pool_size: 两个元素的正数或者元组。指定池化窗口的大小。可以是单个整数表示，指定所有的空间维度为相同的值。
- padding: 一个字符串, padding 的方法, 可以是'valid' 或者是'same'
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,height,width,channels), channels_first 形状为 (batch,channels,height,width)
- name: 字符串, layer 名字。
- 返回值: 输出 tensor。

9.4.3 tf.layers.average_pooling3d

函数功能: 三维输入的平均池化

```

1 average_pooling3d(
2     inputs ,
3     pool_size ,
4     strides ,
5     padding='valid' ,
6     data_format='channels_last' ,
7     name=None
8 )

```

参数:

- inputs: 池化的 Tensor, rank 必须为 5
- pool_size: 正数或者元组。列表 (3 个元素) 指定池化窗口的大小。可以三个元素的整数, 列表或者元组, 指定所有的空间维度为相同的值。
- padding: 一个字符串, padding 的方法, 可以是'valid' 或者是'same'
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,depth,height,width,channels), channels_first 形状为 (batch,depth,channels,height,width)
- name: 字符串, layer 名字。
- 返回值: 输出 tensor。

9.4.4 tf.layers.batch_normalization

函数功能:batch normalization layer 的函数接口。

```

1 batch_normalization(
2     inputs,
3     axis=-1,
4     momentum=0.99,
5     epsilon=0.001,
6     center=True,
7     scale=True,
8     beta_initializer=tf.zeros_initializer(),
9     gamma_initializer=tf.ones_initializer(),
10    moving_mean_initializer=tf.zeros_initializer(),
11    moving_variance_initializer=tf.ones_initializer(),
12    beta_regularizer=None,
13    gamma_regularizer=None,
14    training=False,
15    trainable=True,
16    name=None,
17    reuse=None,
18    renorm=False,
19    renorm_clipping=None,
20    renorm_momentum=0.99,
21    fused=False
22 )

```

batch normalization layer 的函数[参考](#),当训练的时候 moving_mean 和 moving_variance 需要被更新。默认的更新操作在 `tf.GraphKeys.UPDATE_OPS`, 因此她们需要被添加到 `train_op`, 例如

```

1 update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
2 with tf.control_dependencies(update_ops):
3     train_op = optimizer.minimize(loss)

```

参数:

- inputs:2 维以上 Tensor, 进行 normalization, 第一个维度是 `batch_size`, 如果 `data_format` 是 NHWC 的最后一维 `data_format` 是 NCHW 的第二维
- decay: 滑动平均的衰退状态, 接近于 1, 典型值是 0.999, 0.99, 0.9。低的 `decay` 值是 0.9, 如果模型训练性能好但是在验证和测试性能差, 尝试着设置 `zeros_debias_moving_mean=True` 提高稳定性。
- center, bool 型为 True beta 偏移到正则化的 tensor, 如果为 False, beta 被忽略。

- scale:bool 型, 如果为 True, 乘上 gamma, 如果为佳 gamma 不用。当下一层为线性 (nn.relu) 时 scaling 可能被下一层使用而禁用。
- epsilon: 小的浮点数证驾到方差项防止除零错误。
- activation_fn: 激活函数, 默认设置为 None 跳过激活函数使用线性激活函数。
- param_initializers:beta,gamma,moving mean 和 moving variance 优化选项
- param_regularizers:beta 和 gamma 的正则化选项。
- updates_collections: 收集计算的更新操作。updates_ops 需要执行 train_op。如果为 None, 控制依赖将被增加保证更新在适当的位置计算。
- is_training: 这层是否在训练模式如果在训练状态它将用指数移动平均求和加统计矩到 moving_mean 和 moving_variance。当他不在在训练模式的时候她将用 moving_mean 和 moving_variance 的值。
- reuse: 是否层和它的变量被重用, 重用层的范围必须给定。
- variables_collections: 变量的选项收集。
- outputs_collections: 收集到增加输出。
- trainable: 如果为 True 增加变量到图 GraphKeys.TRAINABLE_VARIABLES 上。
- batch_weights: 有 batch_size 形状的 tensor, 包含每个批的频率, 如果设置了值, 批用权重均值和方差归一化 (可以用来收集训练选中样本的偏置)。
- fused: 如果为 True 用一个更快的基于 nn.fused_batch_norm 的融合实现。如果为 None, 如果可能用 fused 实现。
- data_format: 一个字符串, NHWC(默认)NVCHW 也支持。
- zero_debias_moving_mean:moving_mean 用一个 zeros_debias, 它创建一个新的变量 moving_mean/biased 和'moving_mean/local step'
- scope: 变量范围的选项。
- renorm: 是否批重新归一化, 在训练过程中增加额外的变量, 对于这个值得推倒时相同的

- renorm_clipping: 用剪切重新归一化映射 key' rmax', 'rmin', 'dmax' 到标量 Tensor 的词典，相关的 (r,d) 被用作'corrected_value = normalized_value * r + d', r 被剪切到 [rmin,rmax],d 到 [-dmax,dmax], 不指定 rmax,rmin,dmax,dmin 相应的被设置为 inf 和 0.
- renorm_decay:momentum 在重新归一化中用于更新移动平均和标准差，太小（将增加噪声）太大（走样的评估）都将影响训练，decay 用于在推理时得到均值和方差。
- 返回一个代表输出操作的 Tensor。
- 异常
 - ValueError: 如果 batch_weights 不是 None 但是 fused 是 True 时。
 - ValueError: 如果 data_format 不是 NHWC 或者 NCHW。
 - ValueError: 如果 inputs 的 rank 没有指定。
 - ValueError: 如果输入的 channels 或者 rank 没有指定。

9.4.5 conv1d

函数功能: 一维卷积层的函数接口，这个层创建一个卷积核和输出卷积输出一个 tensor。如果 use_bias 是 True(bias_initializer 被提供的话)，bias 向量被创建添加到输出。最后如果激活函数不是 None，激活函数也被用到输出。

```

1 conv1d(
2     inputs ,
3     filters ,
4     kernel_size ,
5     strides=1,
6     padding='valid' ,
7     data_format='channels_last' ,
8     dilation_rate=1 ,
9     activation=None ,
10    use_bias=True ,
11    kernel_initializer=None ,
12    bias_initializer=tf.zeros_initializer() ,
13    kernel_regularizer=None ,
14    bias_regularizer=None ,
15    activity_regularizer=None ,
16    trainable=True ,
17    name=None ,
18    reuse=None
19 )

```

- input: 输入 tensor。
- filters: 整数, 输出空间的维数 (卷积核的数量)。
- kernel_size: 一个整数或者元组或者列表, 指定一维卷积窗的大小。
- strides: 一个单个整数, 列表或者元组, 指定 stride 的长度, 指定任何不等于 1 的常数和指定任何 dilation_rate 值部位 1 不兼容。
- padding:valid 或者 same。
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,length,channels), channels_first 形状为 (batch,channels,length)
- dilation_rate: 一个整数或者元组或者列表, 指定腐蚀卷积的腐蚀率, 指定任何值 dilation_rate!=1 和任何 strides 值!=1 不兼容。
- activation: 激活函数, 设置 None 维持线性激活函数。
- use_bias:bool 型, 是否层用 bias。
- kernel_initializer: 卷积核初始化器。
- bias_initializer: 初始化偏置向量, 如果为 None 将没有 bias 被用。
- activation_regularizer: 输出的正则化函数
- trainable:bool 型, 如果为 True 增加变量到 GraphKeys.TRAINABLE_VARIABLES
- name:layer 的名字。
- reuse: 是否重用之前层的相同名字的权重。
- 输出 tensor。

9.4.6 conv2d

函数功能: 这层创建一个卷积核和输入相卷积输出。如果 use_bias 是 True(bias_initializer 提供了), bias 向量被创建添加到输出, 最后 activation 不是 None, 他被应用到输出。

```

1 conv2d(
2     inputs ,
3     filters ,
4     kernel_size ,
5     strides=(1, 1),

```

```
6     padding='valid',
7     data_format='channels_last',
8     dilation_rate=(1, 1),
9     activation=None,
10    use_bias=True,
11    kernel_initializer=None,
12    bias_initializer=tf.zeros_initializer(),
13    kernel_regularizer=None,
14    bias_regularizer=None,
15    activity_regularizer=None,
16    trainable=True,
17    name=None,
18    reuse=None
19 )
```

- input: 输入 tensor。
- filters: 整数, 输出空间的维数 (卷积核的数量)。
- kernel_size: 2 个整数元素或者元组或者列表, 指定二维卷积窗的宽和高, 可以一用一个整数指定所有空间维度相同。
- strides: 一个整数, 列表或者元组, 指定 stride 的长度, 指定任何不等于 1 的常数和指定任何 dilation_rate 值不为 1 不兼容。
- padding: valid 或者 same。
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,height,width,channels), channels_first 形状为 (batch,channels,height,width)
- dilation_rate: 两个整数或者元组或者列表, 指定腐蚀卷积的腐蚀率, 可以指定单个整数指定所有的空间维数相等, 指定任何值 dilation_rate!=1 和任何 strides 值!=1 不兼容。
- activation: 激活函数, 设置 None 维持线性激活函数。
- use_bias: bool 型, 是否层用 bias。
- kernel_initializer: 卷积核初始化器。
- bias_initializer: 初始化偏置向量, 如果为 None 将没有 bias 被用。

- activation_regularizer: 输出的正则化函数
- trainable:bool 型, 如果为 True 增加变量到 GraphKeys.TRAINABLE_VARIABLES
- name:layer 的名字。
- reuse: 是否重用之前层的相同名字的权重。
- 输出 tensor。

9.4.7 conv2d_transpose

函数功能: 二维卷积层的接口函数, 你希望用变形到正常卷积相反的方向你需要转置卷积, 有时候一些卷积的输出形状和输入形状相同但是维持链接样式是兼容的。

```

1  conv2d_transpose(
2      inputs ,
3      filters ,
4      kernel_size ,
5      strides=(1, 1),
6      padding='valid',
7      data_format='channels_last',
8      activation=None,
9      use_bias=True,
10     kernel_initializer=None,
11     bias_initializer=tf.zeros_initializer(),
12     kernel_regularizer=None,
13     bias_regularizer=None,
14     activity_regularizer=None,
15     trainable=True,
16     name=None,
17     reuse=None
18 )

```

- input: 输入 tensor。
- filters: 整数, 输出空间的维数 (卷积核的数量)。
- kernel_size: 2 个正整数组成的元组或者列表, 指定卷积核的宽和高, 可以一用一个整数指定所有空间维度相同。
- strides: 一个两个正整数组成的列表或者元组, 指定 stride 的长度, 指定任何不等于 1 的常数和指定任何 dilation_rate 值不为 1 不兼容。

- padding:valid 或者 same。
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,height,width,channels), channels_first 形状为 (batch,channels,height,width)
- dilation_rate: 两个整数或者元组或者列表, 指定腐蚀卷积的腐蚀率, 可以指定单个整数指定所有的空间维数相等, 指定任何值 dilation_rate!=1 和任何 strides 值!=1 不兼容。
- activation: 激活函数, 设置 None 维持线性激活函数。
- use_bias:bool 型, 是否层用 bias。
- kernel_initializer: 卷积核初始化器。
- bias_initializer: 初始化偏置向量, 如果为 None 将没有 bias 被用。
- activation_regularizer: 输出的正则化函数
- trainable:bool 型, 如果为 True 增加变量到 GraphKeys.TRAINABLE_VARIABLES
- name:layer 的名字。
- reuse: 是否重用之前层的相同名字的权重。
- 输出 tensor。

9.4.8 conv3d

函数功能: 三维卷积的函数接口。这层创建一个卷积核和输入卷积生成输出 tensor。如果 use_bias 是 True, bias_initializer 被提供, 偏置向量创建添加到输出, 最终如果激活函数不是 None, 激活函数被用在输出上。

```

1 conv3d(
2     inputs ,
3     filters ,
4     kernel_size ,
5     strides=(1, 1, 1),
6     padding='valid',
7     data_format='channels_last',
8     dilation_rate=(1, 1, 1),
9     activation=None,
10    use_bias=True,
```

```
11     kernel_initializer=None,
12     bias_initializer=tf.zeros_initializer(),
13     kernel_regularizer=None,
14     bias_regularizer=None,
15     activity_regularizer=None,
16     trainable=True,
17     name=None,
18     reuse=None
19 )
```

- input: 输入 tensor。
- filters: 整数, 输出空间的维数 (卷积核的数量)。
- kernel_size: 3 个正整数组成的元组或者列表, 指定卷积核的深度, 宽和高, 可以用一个整数指定所有空间维度相同。
- strides: 三个正整数组成的列表或者元组, 指定 stride 的深度, 宽, 高, 指定单个整数代表所有空间维度相同, 指定任何 stride 不等于 1 的常数和指定任何 dilation_rate 值不为 1 不兼容。
- padding: valid 或者 same。
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,depth,height,width,channels), channels_first 形状为 (batch,depth,channels,height,width)
- dilation_rate: 三个整数组成的元组或者列表, 指定腐蚀卷积的腐蚀率, 可以指定单个整数指定所有的空间维数相等, 指定任何值 dilation_rate!=1 和任何 strides 值!=1 不兼容。
- activation: 激活函数, 设置 None 维持线性激活函数。
- use_bias: bool 型, 是否层用 bias。
- kernel_initializer: 卷积核初始化器。
- bias_initializer: 初始化偏置向量, 如果为 None 将没有 bias 被用。
- activation_regularizer: 输出的正则化函数
- trainable: bool 型, 如果为 True 增加变量到 GraphKeys.TRAINABLE_VARIABLES

- name:layer 的名字。
- reuse: 是否重用之前层的相同名字的权重。
- 输出 tensor。

9.4.9 conv3d_transpose

函数功能: 三维卷积函数接口

```
1 conv3d_transpose(  
2     inputs,  
3     filters,  
4     kernel_size,  
5     strides=(1, 1, 1),  
6     padding='valid',  
7     data_format='channels_last',  
8     activation=None,  
9     use_bias=True,  
10    kernel_initializer=None,  
11    bias_initializer=tf.zeros_initializer(),  
12    kernel_regularizer=None,  
13    bias_regularizer=None,  
14    activity_regularizer=None,  
15    trainable=True,  
16    name=None,  
17    reuse=None  
18 )
```

- input: 输入 tensor。
- filters: 整数, 输出空间的维数 (卷积核的数量)。
- kernel_size: 3 个正整数组成的元组或者列表, 可以一用一个整数指定所有空间维度相同。
- strides: 三个正整数组成的列表或者元组, 指定单个整数代表所有空间维度相同。
- padding: valid 或者 same。
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,depth,height,width,channels), channels_first 形状为 (batch,depth,channels,height,width)

- dilation_rate: 三个整数组成的元组或者列表，指定腐蚀卷积的腐蚀率，可以指定单个整数指定所有的空间维数相等，指定任何值 `dilation_rate!=1` 和任何 `strides` 值!=1 不兼容。
- activation: 激活函数，设置 `None` 维持线性激活函数。
- use_bias:bool 型，是否层用 bias。
- kernel_initializer: 卷积核初始化器。
- bias_initializer: 初始化偏置向量，如果为 `None` 将没有 bias 被用。
- activation_regularizer: 输出的正则化函数
- trainable:bool 型，如果为 `True` 增加变量到 `GraphKeys.TRAINABLE_VARIABLES`
- name:layer 的名字。
- reuse: 是否重用之前层的相同名字的权重。
- 输出 tensor。

9.4.10 dense

函数功能: 这个层实现操作`output = activation(input.kernel+bias)`, 这里 `activation` 传入 `activation` 的激活函数 (如果不为 `None`)，`kernel` 是一个层创建的权重矩阵，`bias` 是一个层创建的偏置向量。

```

1  dense(
2      inputs ,
3      units ,
4      activation=None ,
5      use_bias=True ,
6      kernel_initializer=None ,
7      bias_initializer=tf.zeros_initializer() ,
8      kernel_regularizer=None ,
9      bias_regularizer=None ,
10     activity_regularizer=None ,
11     trainable=True ,
12     name=None ,
13     reuse=None
14 )

```

- input: 输入 tensor。

- unit: 整数或者长整数输出空间的维数。
- activation: 激活函数, 设置为 None 表示非线性函数。
- use_bias:bool 型, 当前层是否使用 bias
- kernel_initializer: 权重矩阵的初始化函数。
- bias_initializer: 偏置的初始化函数。
- kernel_regularizer: 权重矩阵的正则化函数。
- bias_regularizer: 偏置的正则化函数。
- activation_regularizer: 输出的正则化函数。
- trainable:bool 型, 如果为 True 增加变量到 GraphKeys.TRAINABLE_VARIABLES
- name:layer 的名字。
- reuse: 是否重用之前层的相同名字的权重。
- 输出 tensor。

9.4.11 dropout

函数功能: 设置随机丢弃值得比率, 帮助阻止过拟合。这个单位被 $\frac{1}{1-rate}$, 因此他们的和在训练和推理时不改变。

```

1 dropout(
2     inputs ,
3     rate=0.5 ,
4     noise_shape=None ,
5     seed=None ,
6     training=False ,
7     name=None
8 )

```

- input: 输入 tensor。
- dropout 比率, 值在 0-1 之间, 例如 rate=0.1 表示丢掉输入的 10%。
- noise_shape:int32 类型的一维 tensor 代表二进制 dropout mask 乘上输入, 例如, 如果你的输入形状为 (batch_size, timesteps, features), 你想 dropout mask 和所有的 timesteps, 你可以用 noise_shape=[batch_size, 1, features]

- seed: 一个 Python 整数用于创建随机数种子。
- training: python bool 或者 TensorFlow bool 标量 tensor(例如 placeholder), 是否在训练模式 (dropout) 或者在推理模式 (返回没修改的输入) 返回输出
- name: layer 的名字。
- 输出 tensor。

9.4.12 max_pool1d

函数功能: 一维输入的最大池化层。

```
1 max_pooling1d(  
2     inputs,  
3     pool_size,  
4     strides,  
5     padding='valid',  
6     data_format='channels_last',  
7     name=None  
8 )
```

- inputs: 需要池化的输入 tensor, rank 必须为 3
- pool_size: 一个整数或者列表或者元组, 代表池化窗口的大小
- strides: 一个整数或者元组或者列表, 指定池化操作的 stride。
- padding: 一个字符串可以为'valid' 或者'same'
- data_format: 一个字符串, 默认为 channels_last 或者 channels_first. 输入维度顺序, channels_last 相关输入的形状为 (batch, length, channels), channels_first 输出形状为 (batch, channels, length)
- name: 一个字符串, layer 的名字
- 输出一个三维 tensor。

9.4.13 max_pool2d

函数功能: 二维输入的最大池化

```

1 max_pooling2d(
2     inputs ,
3     pool_size ,
4     strides ,
5     padding='valid' ,
6     data_format='channels_last' ,
7     name=None
8 )

```

- inputs: 需要池化的输入 tensor, rank 必须为 4
- pool_size: 两个整数组成的元组或者列表 (pool_height,pool_width), 代表池化窗口的大小, 可以为单个整数表示所有空间维度相等。
- strides: 两个整数组成的元组或者列表 (pool_height,pool_width), 指定池化操作的 stride, 可以为单个整数表示所有空间维度相等。
- padding: 一个字符串可以为'valid' 或者'same'
- data_format: 一个字符串, 默认为 channels_last 或者 channels_first. 输入维度顺序, channels_last 相关输入的形状为 (batch, height,width, channels), channels_first 输出形状为 (batch, channels, height,width)
- name: 一个字符串, layer 的名字
- 输出一个三维 tensor。

9.4.14 max_pool3d

函数功能: 三维输入的最大池化层

```

1 max_pooling3d(
2     inputs ,
3     pool_size ,
4     strides ,
5     padding='valid' ,
6     data_format='channels_last' ,
7     name=None
8 )

```

- inputs: 需要池化的输入 tensor, rank 必须为 5

- pool_size: 三个整数组成的元组或者列表 (pool_depth,pool_height,pool_width), 代表池化窗口的大小, 可以为单个整数表示所有空间维度相等。
- strides: 三个整数组成的元组或者列表 (pool_height,pool_width), 指定池化操作的 stride, 可以为单个整数表示所有空间维度相等。
- padding: 一个字符串可以为'valid' 或者'same'
- data_format: 一个字符串, 默认为 channels_last 或者 channels_first. 输入维度顺序, channels_last 相关输入的形状为 (batch, depth,height,width, channels), channels_first 输出形状为 (batch, channels, depth,height,width)
- name: 一个字符串, layer 的名字
- 输出一个三维 tensor。

9.4.15 separable_conv2d

函数功能: 深度方向分隔 2 维卷积层, 这层执行深度方向上通过 chennel 分开的卷积接着在深度方向上混合通道。如果 use_bias 是 True 并且 bias 初始化被提供了, 它增加一个偏置向量到输出, 选项应用激活函数生成最终输出。

- inputs: 需要池化的输入 tensor item filters: 整数, 输出空间的维数 (卷积核的数量)。
- kernel_size: 2 个整数元素或者元组或者列表, 指定二维卷积窗的宽和高, 可以一用一个整数指定所有空间维度相同。
- strides: 两个正整数组成的列表或者元组, 可以一用一个整数指定所有空间维度相同,, 指定任何不等于 1 的常数和指定任何 dilation_rate 值不为 1 不兼容。
- padding: valid 或者 same。
- data_format: 一个字符串 channels_last(默认) 或者 channels_first, 输入维度的顺序, channels_last 输入形状为 (batch,height,width,channels), channels_first 形状为 (batch,channels,height,width)
- dilation_rate: 两个整数或者元组或者列表, 指定腐蚀卷积的腐蚀率, 可以指定单个整数指定所有的空间维数相等, 指定任何值 dilation_rate!=1 和任何 strides 值!=1 不兼容。
- depth_multiplier: 每个输入通道的深度方向卷积输出, 总共的深度方向卷积数等于 num_filters_in*depth_multiplier

- activation: 激活函数, 设置 None 维持线性激活函数。
- use_bias:bool 型, 是否层用 bias。
- depthwise_initializer: 深度方向卷积核的初始化器
- pointwise_initializer:pointwise 卷积核的初始化器。
- depthwise_regularizer:depthwise 卷积核的正则化器。
- pointwise_regularizer:pointwise 卷积核的正则化器。
- bias_regularizer: 偏置向量的正则化器。
- bias_initializer: 初始化偏置向量, 如果为 None 将没有 bias 被用。
- activation_regularizer: 输出的正则化函数
- trainable:bool 型, 如果为 True 增加变量到 GraphKeys.TRAINABLE_VARIABLES
- name:layer 的名字。
- reuse: 是否重用之前层的相同名字的权重。
- 输出 tensor。

9.5 tf.train

提供了训练模型的类和函数。

9.5.1 优化器

优化器类提供方法计算损失函数对于变量的梯度的计算方法，子类集合实现了像 Adagrad 和 GradientDescent 等经典算法。

Optimizer

基础的优化类，定义了增加一个操作到训练模型的 API，你不直接需要这个类而是需要它的一些像 GradientDescentOptimizer, AdagradOptimizer, 或者 MomentumOptimizer 的子类。用法

```

1 # Create an optimizer with the desired parameters.
2 opt = GradientDescentOptimizer(learning_rate=0.1)
3 # Add Ops to the graph to minimize a cost by updating a list of variables.
4 # "cost" is a Tensor, and the list of variables contains tf.Variable
5 # objects.
6 opt_op = opt.minimize(cost, var_list=<list of variables>)

```

在训练程序的过程中你需要返回操作。

```

1 # Execute opt_op to do one step of training:
2 opt_op.run()

```

在应用他们之前处理梯度

条用 minimize() 计算梯度，应用它们在变量上。如果你想在应用他们之前处理你可以按照下面的步骤使用优化器。

1. 用 comput_gradients() 计算梯度。
2. 按照你的希望处理梯度。
3. 用 apply_gradients() 处理梯度。

```

1 # Create an optimizer.
2 opt = GradientDescentOptimizer(learning_rate=0.1)
3
4 # Compute the gradients for a list of variables.
5 grads_and_vars = opt.compute_gradients(loss, <list of variables>)
6

```

```

7 # grads_and_vars is a list of tuples (gradient, variable). Do whatever you
8 # need to the 'gradient' part, for example cap them, etc.
9 capped_grads_and_vars = [(MyCapper(gv[0]), gv[1]) for gv in grads_and_vars]
10
11 # Ask the optimizer to apply the capped gradients.
12 opt.apply_gradients(capped_grads_and_vars)

```

minimize() 的 compute_gradients() 接受一个 gate_gradients 参数控制 gradient 应用中的并行度。

GATE_NONE: 并行的计算，应用梯度，在执行过正中最大化并行程度，在结果中一些非重复性的代价。例如两个梯度的矩阵乘法依赖于输入值:GATE_NONE 可能被应用到输入前其他梯度被计算导致非重复性的结果。

GATE_OP: 对于每个 Op，在他们使用之前确保所有的梯度被计算了。为了避免 Op 的 race 为多个输入生成梯度 condition，这里梯度依赖于输入。

GATE_GRAPH: 确保在它们任何一个被使用前所有变量的梯度被计算，提供了最小的并行化但是如果你想在应用他们之前处理所有的梯度这是很有用的。Slots

像 MomentumOptimizer 和 AdagradOptimizer 之类的优化器子类，结合变量训练分配管理额外的变量。这称为 Slots，Slots 有名字，你可以要求优化器它使用的名字。当你有一个 slot 名字你可以对变量要求优化器创建保留 slot 值。当你调试训练算法报告 slots 统计信息时很管用。方法

```

__init__
1 __init__(
2     use_locking,
3     name
4 )

```

创建一个新的优化器，他必须通过子类的构造体调用。

参数:

- use_locking:bool, 如果为 True 用 lock 阻止当前变量更新。
- name: 非空字符串为 optimizer 创建的累加器的名字。
- ValueError: 名字格式不对。

apply_gradients

```

1 apply_gradients(
2     grads_and_vars,
3     global_step=None,
4     name=None
5 )

```

应用梯度到变量尚，这是 minimize() 的第二部分，他返回应用梯度的 Op。

参数:

- grads_add_vars: 返回 compute_gradients() 的 (梯度, 变量) 对列表。
- global_step: 变量被更新后此选项变量增加 1.
- name: 返回操作的名字。默认传递名字给优化器构造函数。

S 返回: 应用指定梯度的操作。如果 global_step 不是 None，操作增加 global_step

- 异常

- TypeError: 如果 grads_and_vars 不对。
- ValueError: 如果变量的梯度为 none。

computer_gradients

```

1 compute_gradients(
2     loss ,
3     var_list=None ,
4     gate_gradients=GATE_OP ,
5     aggregation_method=None ,
6     colocate_gradients_with_ops=False ,
7     grad_loss=None
8 )

```

计算损失关于 bar_list 的梯度，第一部分是 minimize(). 返回一个 (gradient,variable) 对这里 gradient 是变量的梯度。注意梯度可以是 tensor, IndexedSlices 或者 None (如果没有变量的梯度)

- loss: 包含需要最小化的值的 tensor。
- var_list:tf.Variable 更新最小化 loss 的列表或者元组。默认图中的变量列表在 GraphKey.TRAINABLE_VARIABLES 下。
- gate_gradients: 如何 gate 梯度计算,可以是 GATE_NONE,GATE_OP 或者是 GATE_GRAPH。
- aggregation_method: 指定结合梯度的方法，可用值定义在 AggregationMethod。
- colocate_gradients_with_ops: 如果为 True，尝试随着相关操作 colocating 梯度。
- grad_loss: 一个保持住 loss 梯度的 Tensor。

S 返回 (gradient,variable) 对，变量总是被呈现但是梯度可能是 None。