

# 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 进阶	15
2.1	模型存储和加载	15
2.2	tf.estimator 快速导航	15
2.2.1	完成神经网络源代码	15
2.2.2	载入 CSV 数据进入 TensorFlow	18
2.2.3	构造神经网络分类器	19
2.2.4	描述训练的输入 pipeline	20
2.2.5	为 iris 训练集拟合 DNNClassifier	20
2.2.6	评估模型的精度	21
2.2.7	分类新的样本	21
2.3	用 tf.estimator 创建一个输入函数	22
2.3.1	用 input_fn 自定义 Pipeline	22
2.3.2	input_fn 的分解	22
2.3.3	转换特征数据为 Tensor	23

---

2.3.4	传递 input_fn 数据到你的模型 . . . . .	24
2.3.5	波士顿房价的神经网络模型 . . . . .	25
2.3.6	建立 . . . . .	26
2.3.7	导入的房子数据 . . . . .	26
2.3.8	定义特征列创建回归器 . . . . .	27
2.3.9	构建 input_fn . . . . .	27
2.3.10	训练回归器 . . . . .	27
2.3.11	评估模型 . . . . .	28
2.3.12	做出预测 . . . . .	28
2.4	tf.contrib.learn 采集和监控基础 . . . . .	29
2.4.1	建立 . . . . .	29
2.4.2	概览 . . . . .	30
2.4.3	让你的 TensorFlow 能采集 . . . . .	31
2.4.4	配置 Streaming 评估的 ValidationMonitor . . . . .	31
2.4.5	每 N 步评估 . . . . .	31
2.4.6	用 MetricSpec 自定义评估方案 . . . . .	32
2.4.7	用 ValidationMonitor 提前终止 . . . . .	34
2.4.8	用 TensorBoard 可视化采集数据 . . . . .	35
2.5	TensorBoard Histogram Dashboard . . . . .	35
2.5.1	一个简单的例子 . . . . .	35
2.5.2	Overlay Mode . . . . .	37
2.5.3	多个分布 . . . . .	39
2.5.4	更多分布 . . . . .	41
2.5.5	poisson 分布 . . . . .	44
2.5.6	结合所有的数据到一张图向上 . . . . .	45
3	程序员向导 . . . . .	47
3.1	Estimators . . . . .	47
3.1.1	高级 Estimator . . . . .	47
3.1.2	自定义的 Estimator . . . . .	48
3.1.3	预定义的 Estimator 程序结构 . . . . .	48
3.1.4	预定义 Estimators 的好处 . . . . .	49
3.1.5	自定义 Estimators . . . . .	49
3.1.6	推荐的工作流程 . . . . .	50
3.1.7	从 Keras 模型创建 Estimator . . . . .	50
3.2	Tensor . . . . .	50

---

---

3.2.1	Rank . . . . .	51
3.2.2	获取 Tensor 对象的 rank . . . . .	52
3.2.3	Tensor 的切片 . . . . .	52
3.2.4	形状 . . . . .	52
3.2.5	获取 tf.Tensor 对象的形状 . . . . .	52
3.2.6	改变 Tensor 的形状 . . . . .	53
3.2.7	数据类型 . . . . .	53
3.2.8	评价 Tensor . . . . .	53
3.2.9	打印 Tensor . . . . .	54
3.3	Variable . . . . .	54
3.3.1	创建变量 . . . . .	55
3.3.2	变量集合 . . . . .	55
3.3.3	配置设备 . . . . .	56
3.3.4	初始化变量 . . . . .	56
3.3.5	用变量 . . . . .	57
3.3.6	保存和恢复 . . . . .	57
3.3.7	checkpoint 文件 . . . . .	58
3.3.8	保存变量 . . . . .	58
3.3.9	恢复变量 . . . . .	58
3.3.10	选择变量恢复 . . . . .	59
3.3.11	共享变量 . . . . .	59
3.4	图 (Graphs) 和会话 (Session) . . . . .	60
3.4.1	为什么用数据流图? . . . . .	61
3.4.2	什么是 tf.Graph . . . . .	61
3.4.3	建立一个 tf.Graph . . . . .	61
3.4.4	命名操作 . . . . .	62
3.4.5	放置操作在不同的设备上 . . . . .	63
3.4.6	Tensor-like 对象 . . . . .	64
3.4.7	在 tf.Session 执行图 . . . . .	64
3.4.8	创建 tf.Session . . . . .	65
3.4.9	用 tf.Session.run 执行操作 . . . . .	66
3.4.10	GraphDef 和 MetaGraphDef . . . . .	67
3.4.11	可视化你的图 . . . . .	68
3.4.12	用多图编程 . . . . .	69
3.5	保存和恢复 . . . . .	70

---

---

3.5.1	保存和恢复变量 . . . . .	70
3.5.2	保存变量 . . . . .	71
3.5.3	恢复变量 . . . . .	71
3.5.4	选择什么变量存储和恢复 . . . . .	72
3.5.5	保存和恢复模型概览 . . . . .	73
3.5.6	APIs 构建和载入一个 SavedModel . . . . .	73
3.5.7	构建一个 SavedModel . . . . .	73
3.5.8	在 C++ 中载入一个 SavedModel . . . . .	74
3.5.9	标准的常数 . . . . .	75
3.5.10	标准的 MetaGraphDef tags . . . . .	75
3.5.11	标准的 MetaGraphDef tags . . . . .	75
3.5.12	结合 Estimator 使用 SavedModel . . . . .	75
3.5.13	准备服务输入 . . . . .	75
3.5.14	执行导出 . . . . .	77
3.5.15	指定一个自定义模型的输出 . . . . .	77
3.5.16	服务导入的本地模型 . . . . .	77
3.5.17	从一个本地服务器请求预测 . . . . .	78
3.5.18	CLI 查看和执行 SavedModel . . . . .	79
3.5.19	安装 SavedModel CLI . . . . .	79
3.5.20	命令概览 . . . . .	79
3.5.21	显示 . . . . .	80
3.5.22	运行命令 . . . . .	81
3.5.23	保存输出 . . . . .	82
3.5.24	TensorFlow Debugger(tfdbg) Integration . . . . .	82
3.5.25	run 的完整例子 . . . . .	83
3.5.26	SavedModel 目录的结构 . . . . .	84
3.6	导入数据 . . . . .	85
3.6.1	基本的机制 . . . . .	86
3.6.2	数据结构 . . . . .	86
3.6.3	创建一个迭代器 . . . . .	87
3.6.4	消耗迭代器的值 . . . . .	89
3.6.5	读输入数据 . . . . .	91
3.6.6	消耗 TFRecord 数据 . . . . .	91
3.6.7	用 Dataset.map() 处理数据 . . . . .	92
3.6.8	解析 tf.Example protocol buffer 消息 . . . . .	92

---

---

3.6.9	解码图像数据变换大小 . . . . .	93
3.6.10	用专门的 Python logic . . . . .	93
3.6.11	简单的批处理 . . . . .	94
3.6.12	批量的 tensorpadding . . . . .	94
3.6.13	处理多 epoch . . . . .	95
3.6.14	随机打乱输入数据 . . . . .	96
3.6.15	用高级 APIs . . . . .	96
3.7	线程和队列 . . . . .	97
3.7.1	队列用法 . . . . .	98
3.7.2	手动线程管理 . . . . .	99
3.7.3	Coordinator . . . . .	99
3.7.4	QueueRunner . . . . .	100
3.7.5	处理异常 . . . . .	101
3.8	embeddings . . . . .	102
3.8.1	训练一个 embedding . . . . .	102
3.8.2	可视化 Embeddings . . . . .	103
3.8.3	创建 . . . . .	103
3.8.4	metadadata . . . . .	104
3.8.5	图像 . . . . .	105
3.8.6	交互 . . . . .	105
3.8.7	Projections . . . . .	105
3.8.8	导航 . . . . .	106
3.8.9	合作的特性 . . . . .	107
3.8.10	简单的问答 . . . . .	108
4	使用向导 . . . . .	109
4.1	用 GPU . . . . .	109
4.1.1	手工配置设备 . . . . .	110
4.1.2	允许 GPU 的内存增长 . . . . .	110
4.2	如何利用 Inception 的最后一层重新训练新的分类 . . . . .	112
4.2.1	训练花 . . . . .	112
4.2.2	瓶颈 . . . . .	113
4.2.3	训练 . . . . .	113
4.2.4	用 TensorBoard 可视化 . . . . .	114
4.2.5	用重新训练的模型 . . . . .	114
4.2.6	在你自己的分类上训练 . . . . .	114

---

---

4.2.7	创建一个训练图像集合 . . . . .	115
4.2.8	训练步骤 . . . . .	115
4.2.9	扭曲 . . . . .	115
4.2.10	超参数 . . . . .	116
4.2.11	训练, 验证, 测试集 . . . . .	116
4.2.12	更对模型架构 . . . . .	117
4.3	TF layer 向导: 建立一个卷积神经网络 . . . . .	117
4.3.1	开始 . . . . .	117
4.3.2	介绍卷积神经网络 . . . . .	118
4.3.3	建立 CNN MNIST 分类器 . . . . .	118
4.3.4	输入层 . . . . .	119
4.3.5	第一层卷积层 . . . . .	119
4.3.6	池化层 1 . . . . .	120
4.3.7	二层卷积和池化 . . . . .	120
4.3.8	Dense layer . . . . .	121
4.3.9	Logits Layers . . . . .	121
4.3.10	常见的预测 . . . . .	121
4.3.11	计算 Loss . . . . .	122
4.3.12	配置训练操作 . . . . .	123
4.3.13	增加评估度量 . . . . .	123
4.4	训练评估 CNN MNIST 分类器 . . . . .	123
4.4.1	载入训练和测试数据 . . . . .	123
4.4.2	创建 Estimator . . . . .	124
4.4.3	建立 Logging Hook . . . . .	124
4.4.4	选练模型 . . . . .	125
4.4.5	评估模型 . . . . .	125
4.4.6	运行模型 . . . . .	125
4.5	卷积神经网络 . . . . .	126
4.5.1	概览 . . . . .	126
4.5.2	目标 . . . . .	126
4.5.3	本教程重点 . . . . .	126
4.5.4	模型架构 . . . . .	127
4.5.5	代码组织 . . . . .	127
4.5.6	CIFAR-10 模型 . . . . .	128
4.5.7	模型输入 . . . . .	128

---

---

4.5.8 模型预测 . . . . .	129
4.5.9 开始执行并训练模型 . . . . .	131
4.5.10 评估模型 . . . . .	133
4.5.11 在多个 GPU 机卡上训练模型 . . . . .	133
4.5.12 在多设备中设置变量和操作 . . . . .	134
4.5.13 启动并在多个 GPU 上训练模型 . . . . .	134
4.5.14 下一步 . . . . .	135
4.6 RNN . . . . .	135
4.6.1 The Problem Long-Term Dependencies . . . . .	136
4.6.2 LSTM 网络 . . . . .	136
4.6.3 LSTMs 想法的核心 . . . . .	137
4.6.4 一步步的设置 . . . . .	138
4.6.5 LSTM 的多种变体 . . . . .	139
4.7 向量字表示 . . . . .	141
4.7.1 Vector Representation of Words . . . . .	141
4.7.2 处理噪声的对比训练 . . . . .	141
4.7.3 Skip-gram 模型 . . . . .	143
4.7.4 训练过程 . . . . .	146
4.7.5 嵌套学习结果可视化 . . . . .	147
4.7.6 嵌套学习的评估: 类比推理 . . . . .	147
4.7.7 优化实现 . . . . .	148
4.7.8 RNN . . . . .	148
4.7.9 下载及准备数据 . . . . .	149
4.7.10 LSTM . . . . .	149
4.7.11 截断反向传播 . . . . .	149
4.7.12 输入 . . . . .	150
4.7.13 损失函数 . . . . .	150
4.7.14 多个 LSTM 层堆叠 . . . . .	150
4.7.15 编译并运行代码 . . . . .	151
4.8 图像识别 . . . . .	151
4.8.1 用 Python API . . . . .	153
4.8.2 用 C++ API . . . . .	153
4.8.3 更多学习资源 . . . . .	161
4.9 TensorFlow 实现大规模线性模型 . . . . .	161
4.9.1 什么是线性模型 . . . . .	161

---

---

4.9.2 为什么你想用线性模型? . . . . .	161
4.9.3 tf.estimator 将如何构建线性模型 . . . . .	162
4.9.4 特征交叉 . . . . .	163
4.9.5 Bucketization . . . . .	163
4.9.6 线性 estimator . . . . .	164
4.9.7 广泛深入的学习 . . . . .	164
4.10 tensorflow 线性模型导航 . . . . .	165
4.10.1 读取调查数据 . . . . .	165
4.10.2 转换数据为 Tensors . . . . .	167
4.10.3 多列交叉的交叉列 . . . . .	170
4.10.4 逻辑回归如何工作 . . . . .	173
4.11 TensorFlow 广泛深入的学习 . . . . .	173
4.11.1 建立 . . . . .	174
4.11.2 定义基本特征列 . . . . .	175
4.11.3 宽模型: 具有交叉特征列的线性模型 . . . . .	176
4.11.4 深层模型: 嵌入式神经网络 . . . . .	177
4.11.5 将宽和深度模型结合为一体 . . . . .	177
4.11.6 训练和评估模型 . . . . .	178
4.12 移动平台 . . . . .	180
4.12.1 TensorFlow LiteVS TensorFlow Mobile . . . . .	180
4.12.2 介绍 TensorFlow Lite . . . . .	180
4.12.3 TensorFlow Lite 包含什么? . . . . .	180
4.13 介绍 TensorFlow Mobile . . . . .	181
4.13.1 关于这个向导 . . . . .	181
4.13.2 常用的机器学习情景 . . . . .	181
4.13.3 语音识别 . . . . .	181
4.13.4 图像识别 . . . . .	181
4.13.5 对象定位 . . . . .	182
4.13.6 手势识别 . . . . .	182
4.13.7 光学字符识别 . . . . .	182
4.13.8 翻译 . . . . .	182
4.13.9 文本分类 . . . . .	183
4.13.10 语音合成 . . . . .	183
4.13.11 移动机器学习和云 . . . . .	183
4.13.12 你应该拥有什么软件和硬件? . . . . .	183

---

---

4.13.13 在开始之前你需要做什么? . . . . .	183
4.13.14 你得问题是否是移动机器学习能解决的? . . . . .	184
4.13.15 创建标记的数据集 . . . . .	184
4.13.16 选择一个高效的模型 . . . . .	184
4.13.17 下一步 . . . . .	184
4.13.18 为什么需要一个新的专为移动平台设计的库? . . . . .	185
4.13.19 TensorFlow Lite 开发者预览重点 . . . . .	185
4.13.20 开始 . . . . .	186
4.13.21 重新训练 Inception V3 或者 MobileNet 用于用户自定义的数据集 . . . . .	186
4.13.22 TensorFlow Lite 架构 . . . . .	186
4.13.23 将来的工作 . . . . .	188
4.13.24 下一步 . . . . .	188
4.14 在 Android 上构建 TensorFlow . . . . .	188
4.14.1 使用 Android Studio 构建 Demo . . . . .	188
4.14.2 用 Android Studio 添加 TensorFlow 到你的 app 上 . . . . .	189
4.14.3 使用 Bazel 构建 demo . . . . .	189
4.14.4 Android 样例 App . . . . .	190
4.14.5 Android Inference Library . . . . .	190
4.15 在 iOS 上构建 TensorFlow . . . . .	191
4.15.1 使用 CocoaPods . . . . .	191
4.15.2 创建你自己的 app . . . . .	191
4.15.3 运行样例 . . . . .	191
4.15.4 iOS 样本详情 . . . . .	192
4.15.5 Troubleshooting . . . . .	192
4.15.6 从源代码构建 TensorFlow iOS 库 . . . . .	192
4.16 整合 TensorFlow 库 . . . . .	193
4.16.1 链接库 . . . . .	193
4.16.2 Android . . . . .	193
4.16.3 iOS . . . . .	193
4.16.4 全局结构体 magic . . . . .	194
4.16.5 Protobuf 问题 . . . . .	195
4.16.6 在同样的 app 中有多个版本的 protobufs . . . . .	195
4.16.7 调用 TensorFlow API . . . . .	196
4.16.8 Android . . . . .	196
4.16.9 iOS 和 Raspberry Pi . . . . .	196

---

---

4.17 为移动部署准备模型 . . . . .	197
4.17.1 保存的文件格式有什么不同? . . . . .	197
4.17.2 如何获取一个能在移动端运行的模型? . . . . .	198
4.17.3 用图转换工具 . . . . .	199
4.17.4 移除仅仅训练节点 . . . . .	199
4.17.5 什么操作应该被包含在移动端? . . . . .	201
4.17.6 定位实现 . . . . .	202
4.17.7 添加实现构建 . . . . .	202
4.17.8 为移动端优化 . . . . .	202
4.17.9 TensorFlow 最低的设备要求是什么? . . . . .	202
4.17.10 Speed . . . . .	203
4.17.11 模型大小 . . . . .	203
4.17.12 二进制文件的大小 . . . . .	204
4.17.13 如何探测你的模型 . . . . .	205
4.17.14 探测你的 app . . . . .	207
4.17.15 可视化模型 . . . . .	208
4.17.16 线程 . . . . .	208
4.17.17 使用移动数据重新训练 . . . . .	208
4.17.18 减少模型载入时间和内存 footprint . . . . .	209
4.17.19 从简单的复制保护模型文件 . . . . .	210
4.18 为移动部署准备模型 . . . . .	211
4.18.1 保存的文件格式有什么不同? . . . . .	211
 5 扩展 . . . . .	213
5.1 TensorFlow 架构 . . . . .	213
5.2 概述 . . . . .	213
5.2.1 Client . . . . .	215
5.2.2 Distributed master . . . . .	216
5.2.3 Worker Service . . . . .	218
5.3 内核实现 . . . . .	219
5.3.1 代码 . . . . .	219
5.4 自定义文件读取器 . . . . .	219
5.4.1 写一个 Reader 用于文件格式 . . . . .	219
5.4.2 写一个操作用于记录格式 . . . . .	222
5.5 用 tf.estimator 创建 Estimator . . . . .	223
5.5.1 预先要求 . . . . .	224

---

---

5.5.2	一个 Abalone 年龄预测器 . . . . .	224
5.5.3	开始 . . . . .	225
5.5.4	载入 abalone csv 数据到 TensorFlow 数据集 . . . . .	225
5.5.5	实例化一个 Estimator . . . . .	227
5.5.6	构造 model_fn . . . . .	228
5.5.7	结合 tf.feature_column 和 tf.layers 配置神经网络 . . . . .	230
5.5.8	为模型定义一个损失 . . . . .	231
5.5.9	定义为 model 训练操作 . . . . .	232
5.5.10	完整的 abalone model_fn . . . . .	233
5.5.11	运行 Abalone 模型 . . . . .	234
5.5.12	附加资源 . . . . .	235
5.6	TensorFlow 用其他语言 . . . . .	236
5.6.1	背景 . . . . .	236
5.6.2	概览 . . . . .	236
5.6.3	当前状态 . . . . .	236
5.6.4	运行一个预定义的图 . . . . .	237
5.6.5	图的构造 . . . . .	238
5.6.6	处理常数 . . . . .	239
5.6.7	可选参数 . . . . .	239
5.6.8	Name scopes . . . . .	239
5.6.9	包装器 . . . . .	239
5.6.10	其它的考虑 . . . . .	240
5.6.11	梯度, 函数和控制流 . . . . .	240
5.7	一个 TensorFlow 模型文件得开发者工具 . . . . .	240
5.7.1	Protocol Buffers . . . . .	240
5.7.2	GraphDef . . . . .	240
5.7.3	Nodes . . . . .	241
5.7.4	Freezing . . . . .	242
5.7.5	权重格式 . . . . .	242
6	性能向导 . . . . .	245
6.0.1	一般的最佳实践 . . . . .	245
6.0.2	数据格式 . . . . .	247
6.0.3	从源代码构建安装 . . . . .	248
6.1	高性能模式 . . . . .	256
6.1.1	输入 pipeline . . . . .	256

---

---

6.1.2 并行化 I/O 读取 . . . . .	256
6.1.3 并行化图像处理 . . . . .	257
6.1.4 并行化 CPU 到 GPU 数据转化 . . . . .	257
6.1.5 软件 pipeline . . . . .	257
6.1.6 构建高性能模型的最佳实践 . . . . .	258
6.1.7 用 NHWC 和 NCHW 构建模型 . . . . .	258
6.1.8 使用融合的 Batch-Normalization . . . . .	258
6.1.9 变量分布和梯度聚合 . . . . .	259
6.1.10 参数服务器变量 . . . . .	259
6.1.11 复制的变量 . . . . .	260
6.1.12 在分布式系统上训练时复制变量 . . . . .	260
6.1.13 NCCL . . . . .	261
6.1.14 Staged 变量 . . . . .	262
6.1.15 执行脚本 . . . . .	262
6.1.16 基本的命令行参数 . . . . .	262
6.1.17 分布式的命令行参数 . . . . .	263
6.1.18 分布式例子 . . . . .	263
6.2 基准测试 . . . . .	264
6.2.1 概览 . . . . .	264
6.2.2 图形分类模型的结果 . . . . .	264
6.2.3 在 NVIDIA DGX-1(NVIDIA Tesla P100) . . . . .	264
6.2.4 用 Tesla K80 分布式的训练 . . . . .	265
6.2.5 结合真是数据训练比较 . . . . .	265
6.2.6 详细的 NVIDIA DGX-1(NVIDIA Tesla P100) . . . . .	266
6.2.7 Amazon Ec2 详情 (NVIDIA Tesla K80) . . . . .	270
6.2.8 在 Amazon EC2 上 (NVIDIA Tesla K80) . . . . .	272
6.2.9 方法论 . . . . .	275
 7 常用的 python 模块 . . . . .	277
7.1 Argparse . . . . .	277
7.1.1 ArgumentParser 对象 . . . . .	278
7.1.2 prog . . . . .	278
7.1.3 add_argument() 方法 . . . . .	284
7.2 path . . . . .	311
7.2.1 函数说明 . . . . .	311
7.2.2 例子 . . . . .	313

---

---

7.2.3 常见问题 . . . . .	314
7.3 正则表达式介绍 . . . . .	324
7.4 RE 库的主要功能函数 . . . . .	331
7.4.1 re 表达式中的 flags . . . . .	332
7.5 常用的 sys 函数 . . . . .	337
7.6 collections . . . . .	344
7.6.1 namedtuple . . . . .	344
7.6.2 deque . . . . .	344
7.6.3 defaultdict . . . . .	345
7.6.4 OrderedDict . . . . .	345
7.6.5 Counter . . . . .	346
7.7 base64 . . . . .	347
7.8 struct . . . . .	349
7.9 hashlib . . . . .	350
7.10 itertools . . . . .	352
7.10.1 cycle . . . . .	352
7.10.2 chain() . . . . .	353
7.11 contextlib . . . . .	354
7.12 XML . . . . .	357
7.13 HTMLParser . . . . .	358
7.14 ZipFile . . . . .	359
7.15 url . . . . .	360
7.15.1 urllib.request . . . . .	360
7.16 requests . . . . .	361
7.16.1 发送请求 . . . . .	361
7.16.2 requests 库的 7 个主要方法 . . . . .	361
7.16.3 request 对象的属性 . . . . .	361
7.16.4 理解 encoding 和 apparent_encoding . . . . .	362
7.16.5 理解 Requests 库的异常 . . . . .	362
7.16.6 HTTP 协议 . . . . .	362
8 Bazel . . . . .	365
8.1 Bazel start . . . . .	365
8.1.1 用工作空间 . . . . .	365
8.1.2 创建一个构建文件 . . . . .	365
8.1.3 下一步 . . . . .	366

---

---

8.2 构建 C++ 工程 . . . . .	366
8.2.1 你将学习 . . . . .	366
8.2.2 准备 . . . . .	367
8.2.3 构建 Bazel . . . . .	367
8.2.4 明白 BUILD 文件 . . . . .	368
8.2.5 构建工程 . . . . .	368
8.2.6 回顾依赖图 . . . . .	369
8.2.7 提炼你的 Bazel 构建 . . . . .	369
8.2.8 指定多个构建目标 . . . . .	369
8.2.9 用多个包 . . . . .	370
8.2.10 用标签访问目标 . . . . .	372
8.2.11 进一步阅读 . . . . .	372
8.3 常用的 C++ 构建情况 . . . . .	372
8.3.1 一个目标中有多个文件 . . . . .	372
8.3.2 用 include . . . . .	373
8.3.3 添加包含路径 . . . . .	373
8.3.4 包含外部的库 . . . . .	374
8.3.5 写, 运行一个 C++ Test . . . . .	376
8.3.6 为预编译库添加依赖 . . . . .	376
<b>9 实践部分</b> . . . . .	379
9.1 TensorFlow 例子 . . . . .	379
9.1.1 CNN 手写体数据识别 . . . . .	379
9.1.2 mnist 数据集 . . . . .	379
9.1.3 卷积神经网络处理序列数据 . . . . .	386
9.1.4 LSTM 处理序列数据 . . . . .	394
<b>10 Tensorflow 技巧</b> . . . . .	405
10.1 文件读取 . . . . .	405
10.1.1 批量读取压缩图片为指定格式 . . . . .	405
<b>11 Tensorflow API</b> . . . . .	407
11.1 tf.check_numerics . . . . .	407
11.2 tf.clip_by_value . . . . .	407
11.3 tf.app.flags . . . . .	408
11.3.1 DEFINE_boolean . . . . .	408

---

---

11.3.2	DEFINE_boolean	408
11.3.3	DEFINE_float	408
11.3.4	DEFINE_integer	408
11.3.5	DEFINE_string	409
11.3.6	tf.convert_to_tensor	409
11.3.7	tf.gather	410
11.3.8	tf.one_hot	411
11.3.9	tf.placeholder	413
11.3.10	tf.py_func	414
11.3.11	tf.read_file	414
11.3.12	tf.squeeze	415
11.3.13	tf.metrics	415
11.3.14	tf.split	416
11.3.15	tf.stack	417
11.3.16	tf.reshape	417
11.3.17	tf.random_crop	418
11.3.18	tf.random_gamma	419
11.3.19	tf.random_normal	420
11.3.20	tf.random_normal_initializer	420
11.3.21	tf.random_possion	421
11.3.22	random_shuffle	422
11.3.23	tf.random_uniform	423
11.3.24	tf.random_uniform_initializer	423
11.3.25	tf.transpose	424
11.3.26	tf.one_hot	425
11.3.27	tf.tile	427
11.3.28	tf.unstack	428
11.3.29	tf.contrib.rnn	429
11.4	tf.Vairable	430
11.4.1	Variable 类	430
11.4.2	方法	431
11.4.3	参数	437
11.5	tf.image	444
11.5.1	adjust_brightness	444
11.5.2	adjust_contrast	444

---

---

11.5.3	adjust_gamma	444
11.5.4	adjust_hug	445
11.5.5	adjust_saturation	446
11.5.6	centrol_crop	446
11.5.7	decode_bmp	447
11.5.8	tf.image.decode_gif	447
11.5.9	tf.image.decode_jpeg	448
11.5.10	tf.image.encode_jpeg	448
11.5.11	tf.image.decode_png	449
11.5.12	tf.image.encode_png	449
11.5.13	tf.image.decode_image	450
11.5.14	tf.image.resize_images	450
11.6	tf.feature_cloumn	451
11.6.1	bucketized_column	451
11.6.2	categorical_column_with_hash_bucket	452
11.6.3	categorical_column_with_identity	453
11.6.4	categorical_column_with_vocabulary_file	454
11.6.5	categorical_column_with_vocabulary_list	455
11.6.6	cross_column	457
11.6.7	embedding_column	459
11.6.8	indicator_column	460
11.6.9	input_layer	460
11.6.10	linear_model	461
11.6.11	make_sparse_example_spec	462
11.6.12	numeric_column	463
11.6.13	weighted_catrgorical_column	464
11.7	layer	466
11.7.1	tf.layers.average_pooling1d	466
11.7.2	tf.layers.average_pooling2d	466
11.7.3	tf.layers.average_pooling3d	467
11.7.4	tf.layers.batch_normalization	468
11.7.5	conv1d	470
11.7.6	conv2d	471
11.7.7	conv2d_transpose	473
11.7.8	conv3d	474

---

---

## 目录

11.7.9 conv3d_transpose . . . . .	476
11.7.10 dense . . . . .	477
11.7.11 dropout . . . . .	478
11.7.12 max_pool1d . . . . .	479
11.7.13 max_pool2d . . . . .	479
11.7.14 max_pool3d . . . . .	480
11.7.15 separable_conv2d . . . . .	481
11.8 tf.train . . . . .	483
11.8.1 优化器 . . . . .	483
11.8.2 tf.train.slice_input_producer . . . . .	486
11.8.3 tf.train.shuffle_batch . . . . .	486



# 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。

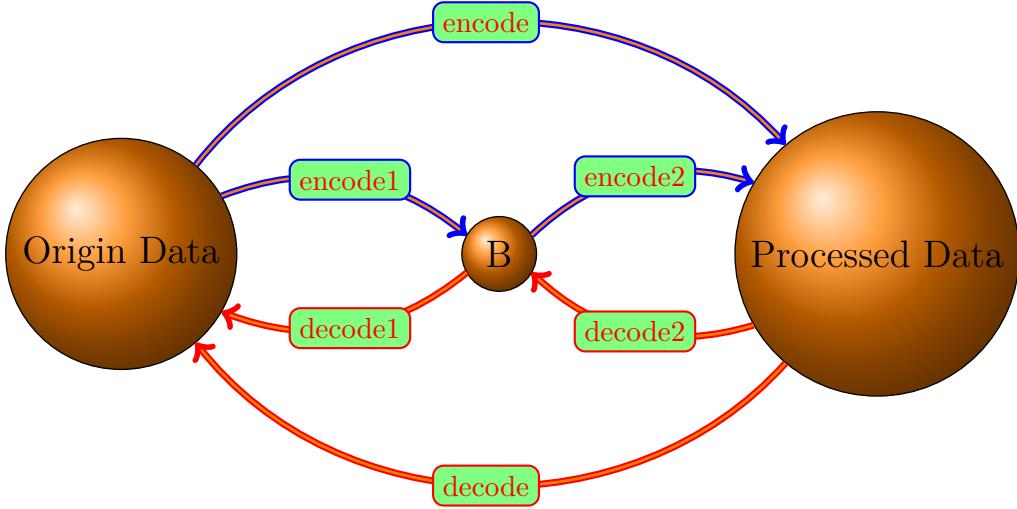
#### 1.1.2 自动降噪编码

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損数据，这样的数据有两个作用：

1. 通过破損数据和非破損数据相比，破損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)

```

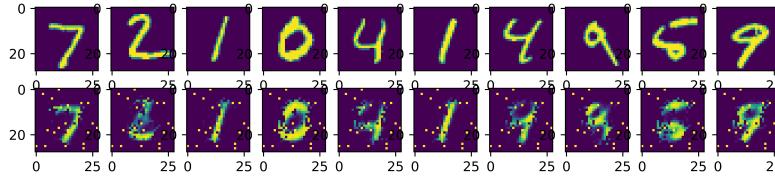


图 1.1: 原图和自编码解码后的图像

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

```

```

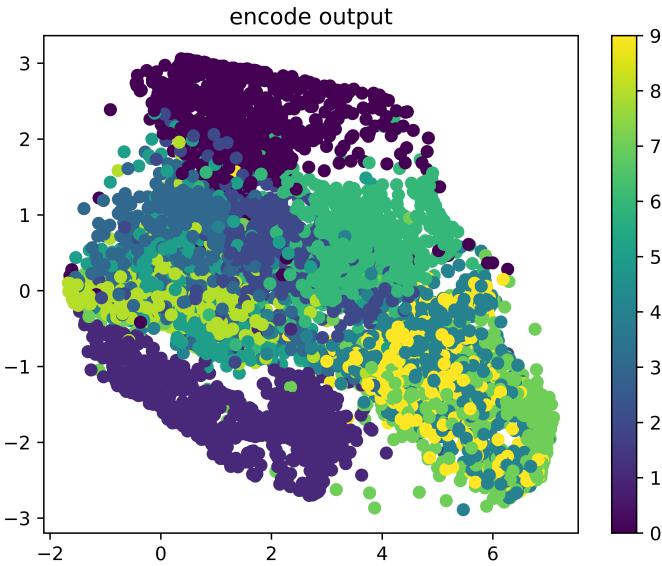
23 batch_size = 256
24 display_step = 1
25 n_input = 784 # MNIST data input (img shape: 28*28)
26 X = tf.placeholder("float", [None, n_input])
27 n_hidden_1 = 128
28 n_hidden_2 = 64
29 n_hidden_3 = 10
30 n_hidden_4 = 2
31 weights = {
32     'encoder_h1': tf.Variable(tf.truncated_normal([n_input, n_hidden_1],)),
33     'encoder_h2': tf.Variable(tf.truncated_normal([n_hidden_1, n_hidden_2],)),
34     'encoder_h3': tf.Variable(tf.truncated_normal([n_hidden_2, n_hidden_3],)),
35     'encoder_h4': tf.Variable(tf.truncated_normal([n_hidden_3, n_hidden_4],)),
36     'decoder_h1': tf.Variable(tf.truncated_normal([n_hidden_4, n_hidden_3],)),
37     'decoder_h2': tf.Variable(tf.truncated_normal([n_hidden_3, n_hidden_2],)),
38     'decoder_h3': tf.Variable(tf.truncated_normal([n_hidden_2, n_hidden_1],)),
39     'decoder_h4': tf.Variable(tf.truncated_normal([n_hidden_1, n_input],)),
40 }
41 biases = {
42     'encoder_b1': tf.Variable(tf.random_normal([n_hidden_1])),
43     'encoder_b2': tf.Variable(tf.random_normal([n_hidden_2])),
44     'encoder_b3': tf.Variable(tf.random_normal([n_hidden_3])),
45     'encoder_b4': tf.Variable(tf.random_normal([n_hidden_4])),
46     'decoder_b1': tf.Variable(tf.random_normal([n_hidden_3])),
47     'decoder_b2': tf.Variable(tf.random_normal([n_hidden_2])),
48     'decoder_b3': tf.Variable(tf.random_normal([n_hidden_1])),
49     'decoder_b4': tf.Variable(tf.random_normal([n_input])),
50 }
51 def encoder(x):
52     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encoder_h1']),
53                                 biases['encoder_b1']))
54     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['encoder_h2']),
55                                 biases['encoder_b2']))
56     layer_3 = tf.nn.sigmoid(tf.add(tf.matmul(layer_2, weights['encoder_h3']),
57                                 biases['encoder_b3']))
58     layer_4 = tf.add(tf.matmul(layer_3, weights['encoder_h4']),
59                     biases['encoder_b4'])
60     return layer_4
61 def decoder(x):
62     layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['decoder_h1']),
63                                 biases['decoder_b1']))
64     layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['decoder_h2']),
65                                 biases['decoder_b2']))

```

```

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  $\phi_i$ ，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  $\lambda$  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  $\phi_i$  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：

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

### 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  $\phi_i$  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  $\phi$ ，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  $\phi$ ，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  $\sigma^2$  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  $\phi$  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：

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

跟之前一样，我们可以通过减小  $a_i$  或增大  $\phi$  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  $\phi$  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， $\Sigma$  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。 $\mathbf{X}$  的单向量矩阵  $\mathbf{W}$  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  $\Sigma$  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

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

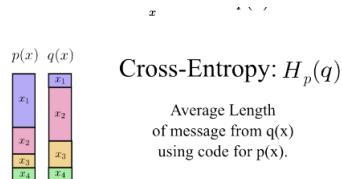


图 1.2: Cross\_Entropy\_exp

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:

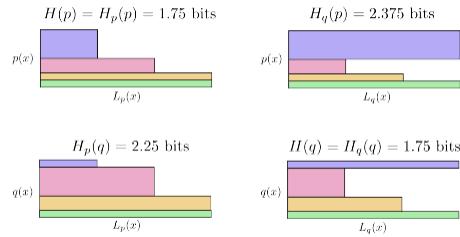


图 1.3: 比较四种情况算得的信息 bit

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。

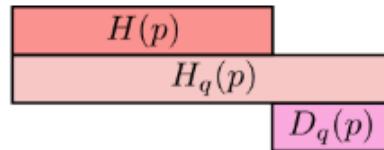


图 1.4: p 对 q 的交叉熵和 p 的熵

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。

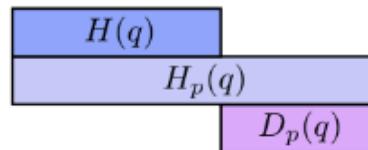


图 1.5: q 对 p 的交叉熵和 q 的熵

### 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 p 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 都满足对于  $Q(i) > 0$  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 可按积分方式定义为:

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

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 进阶

### 2.1 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。它包含权重和其它程序中定义的变量，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，只包含图形结构，不包含权重，然后使用 tf.import\_graph\_def() 加载 图形。

### 2.2 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，你将用 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

#### 2.2.1 完成神经网络源代码

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sat Sep  2 17:05:23 2017
4 win10 tensorflow-gpu 1.3
5 @author: Alien
6 """
7 import os
8 import urllib.request
9
10 import numpy as np
11 import tensorflow as tf
12 #Ignore warning
13 os.environ['TF_CPP_MIN_LOG_LEVEL']= '2'
14
15 # Data sets
16 IRIS_TRAINING = "iris_training.csv"
17 IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"
18
19 IRIS_TEST = "iris_test.csv"
20 IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"
21
22 def main():
23     # If the training and test sets aren't stored locally, download them.
24     if not os.path.exists(IRIS_TRAINING):
25         raw = urllib.request.urlopen(IRIS_TRAINING_URL).read().decode('utf-8')
26         with open(IRIS_TRAINING, "wb") as f:
27             f.write(raw)
28
29     if not os.path.exists(IRIS_TEST):
30         raw = urllib.request.urlopen(IRIS_TEST_URL).read().decode('utf-8')
31         with open(IRIS_TEST, "wb") as f:
32             f.write(raw)
33
34 # Load datasets.
35 training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
36     filename=IRIS_TRAINING,
37     target_dtype=np.int,
38     features_dtype=np.float32)
39 test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
40     filename=IRIS_TEST,
41     target_dtype=np.int,
42     features_dtype=np.float32)
43

```

```

44 # Specify that all features have real-value data
45 feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]
46
47 # Build 3 layer DNN with 10, 20, 10 units respectively.
48 classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
49                                         hidden_units=[10, 20, 10],
50                                         n_classes=3,
51                                         model_dir="./iris_model")
52
53 # Define the training inputs
54 train_input_fn = tf.estimator.inputs.numpy_input_fn(
55     x={"x": np.array(training_set.data)},
56     y=np.array(training_set.target),
57     num_epochs=None,
58     shuffle=True)
59
60 # Train model.
61 classifier.train(input_fn=train_input_fn, steps=2000)
62
63 # Define the test inputs
64 test_input_fn = tf.estimator.inputs.numpy_input_fn(
65     x={"x": np.array(test_set.data)},
66     y=np.array(test_set.target),
67     num_epochs=1,
68     shuffle=False)
69
70 # Evaluate accuracy.
71 accuracy_score = classifier.evaluate(input_fn=test_input_fn)[ "accuracy"]
72
73 print("\nTest Accuracy: {} \n".format(accuracy_score))
74
75 # Classify two new flower samples.
76 new_samples = np.array([
77     [6.4, 3.2, 4.5, 1.5],
78     [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
79 predict_input_fn = tf.estimator.inputs.numpy_input_fn(
80     x={"x": new_samples},
81     num_epochs=1,
82     shuffle=False)
83
84 predictions = list(classifier.predict(input_fn=predict_input_fn))
85 predicted_classes = [p[ "classes"] for p in predictions]
86
87 print(

```

```

87     "New Samples, Class Predictions:    {}\n"
88     .format(predicted_classes))
89
90 if __name__ == "__main__":
91     main()

```

下面的章节将详细介绍代码。

### 2.2.2 载入 CSV 数据进入 TensorFlow

Iris data set 包含有 150 行 iris 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 import os
2 import urllib.request
3
4 import numpy as np
5 import tensorflow as tf
6 #Ignore warning
7 os.environ['TF_CPP_MIN_LOG_LEVEL']= '2'
8
9 # Data sets
10 IRIS_TRAINING = "iris_training.csv"
11 IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"
12
13 IRIS_TEST = "iris_test.csv"
14 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，下载它们:

```

1 if not os.path.exists(IRIS_TRAINING):
2     raw = urllib.request.urlopen(IRIS_TRAINING_URL).read().decode('utf-8')
3     with open(IRIS_TRAINING, "wb") as f:
4         f.write(raw)
5
6 if not os.path.exists(IRIS_TEST):
7     raw = urllib.request.urlopen(IRIS_TEST_URL).read().decode('utf-8')
8     with open(IRIS_TEST, "wb") 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.data 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。

### 2.2.3 构造神经网络分类器

tf.estimator 提供多种预定义方法, 称为 Estimator, 你可以通过它在你的数据上运行训练, 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=". ./iris_model")

```

上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 10,20,10 个神经元。
- `n_classes=3`, 三个目标类，对应三个 iris 种类。
- `model_dir=("./iris_model")`: 训练模型中保存 checkpoint 文件的路径

#### 2.2.4 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)

```

#### 2.2.5 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)

```

状态模型被保存在 `classifier`，意味着你可以反复训练，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。

花萼长度	花萼宽度	花瓣长度	花瓣宽度
6.4	3.2	4.5	1.5
5.8	3.1	5.0	1.7

### 2.2.6 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，下面的代码传递 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, 这个错误通知分类器停止评估，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%。

### 2.2.7 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：

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分类：

```

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)

```

## 2.3. 用 TF.ESTIMATOR 创建一个输入函数

```
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。

## 2.3 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，然后模型将为神经网络回归器实现一个 input\_fn 函数，评估预测房价数据

### 2.3.1 用 input\_fn 自定义 Pipeline

input\_fn 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)
```

### 2.3.2 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，像数据清洗和特征缩放，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：

feature\_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)

### 2.3.3 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, 形状为  $3 \times 6$ , dense\_shape=[2,3,4] 指定 3 维 tensor, 形状为  $2 \times 3 \times 4$  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] 代表二维 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。）

## 2.3. 用 TF.ESTIMATOR 创建一个输入函数

- 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。

下面的代码定义一个二维  $3 \times 5$  的 SparseTensor, 索引为 [0,1] 的位置的值为 6, [2,4] 位置的值为 0.5, 其它值为 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]]
```

### 2.3.4 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:

## 2.3. 用 TF.ESTIMATOR 创建一个输入函数

```
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\_prediction）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)
```

### 2.3.5 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 你将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

### 2.3.6 建立

下载数据集**boston\_train.csv**,**boston\_test.csv**和**boston\_predict.csv**

### 2.3.7 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:

```

1 COLUMNS = ["crim", "zn", "indus", "nox", "rm", "age",
2             "dis", "tax", "ptratio", "medv"]
3 FEATURES = ["crim", "zn", "indus", "nox", "rm",
4             "age", "dis", "tax", "ptratio"]
5 LABEL = "medv"
6
7 training_set = pd.read_csv("boston_train.csv", skipinitialspace=True,
8                             skiprows=1, names=COLUMNS)
9 test_set = pd.read_csv("boston_test.csv", skipinitialspace=True,

```

```
10         skiprows=1, names=COLUMNS)
11 prediction_set = pd.read_csv("boston_predict.csv", skipinitialspace=True,
12                             skiprows=1, names=COLUMNS)
```

### 2.3.8 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的特征包含连续的值，你可以用 `tf.contrib.layers.real_value_column()` 创建它们的 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

```
1 regressor = tf.estimator.DNNRegressor(feature_columns=feature_cols,
2                                         hidden_units=[10, 10],
3                                         model_dir="/boston_model")
```

### 2.3.9 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`。

### 2.3.10 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`:

## 2.3. 用 TF.ESTIMATOR 创建一个输入函数

```
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:Create CheckpointSaverHook.
2 INFO:tensorflow:Saving checkpoints for 1 into ./boston_model_demo/model.ckpt.
3 INFO:tensorflow:loss = 58842.2, step = 1
4 INFO:tensorflow:global_step/sec: 379.343
5 INFO:tensorflow:loss = 10089.6, step = 101 (0.264 sec)
6 INFO:tensorflow:global_step/sec: 414.38
7 INFO:tensorflow:loss = 12056.9, step = 201 (0.241 sec)
8 INFO:tensorflow:global_step/sec: 417.317
9 ...
10 INFO:tensorflow:loss = 2884.69, step = 4801 (0.271 sec)
11 INFO:tensorflow:global_step/sec: 388.267
12 INFO:tensorflow:loss = 5111.75, step = 4901 (0.257 sec)
13 INFO:tensorflow:Saving checkpoints for 5000 into ./boston_model_demo/model.ckpt.
14 INFO:tensorflow:Loss for final step: 4082.04.
15 INFO:tensorflow:Starting evaluation at 2017-11-21-05:42:23
16 INFO:tensorflow:Restoring parameters from ./boston_model_demo/model.ckpt-5000
17 INFO:tensorflow:Finished evaluation at 2017-11-21-05:42:23
18 INFO:tensorflow:Saving dict for global step 5000: average_loss = 13.7577,
      global_step = 5000, loss = 1375.77
19 Loss: 1375.769165
20 INFO:tensorflow:Restoring parameters from ./boston_model_demo/model.ckpt-5000
```

### 2.3.11 评估模型

下一步看看模型在测试数据集上的性能，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 Loss: 1375.769165
```

### 2.3.12 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: [array([ 34.70497513], dtype=float32), array([ 17.63309288], dtype=
2   float32), array([ 23.71421814], dtype=float32), array([ 35.60274506], dtype=
3   float32), array([ 15.12172413], dtype=float32), array([ 19.79147911], dtype=
4   float32)]

```

## 2.4 tf.contrib.learn 采集和监控基础

当我们训练模型的时候实时跟踪评估是有价值的, 在这个导航中, 你将学习如何用 TensorFlow 的采集能力和 Monitor API 在分类 iris 花的过程中省查程序。这个导航的代码基于上一上。

### 2.4.1 建立

```

1 from __future__ import absolute_import
2 from __future__ import division
3 from __future__ import print_function
4
5 import os
6
7 import numpy as np
8 import tensorflow as tf
9
10 # Data sets
11 IRIS_TRAINING = os.path.join(os.path.dirname(__file__), "iris_training.csv")
12 IRIS_TEST = os.path.join(os.path.dirname(__file__), "iris_test.csv")
13
14 def main(unused_argv):
15     # Load datasets.
16     training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
17         filename=IRIS_TRAINING, target_dtype=np.int, features_dtype=np.float32)
18     test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
19         filename=IRIS_TEST, target_dtype=np.int, features_dtype=np.float32)
20
21     # Specify that all features have real-value data

```

```

22 feature_columns = [tf.contrib.layers.real_valued_column("", dimension=4)]
23
24 # Build 3 layer DNN with 10, 20, 10 units respectively.
25 classifier = tf.contrib.learn.DNNClassifier(feature_columns=feature_columns,
26                                              hidden_units=[10, 20, 10],
27                                              n_classes=3,
28                                              model_dir="/tmp/iris_model")
29
30 # Fit model.
31 classifier.fit(x=training_set.data,
32                  y=training_set.target,
33                  steps=2000)
34
35 # Evaluate accuracy.
36 accuracy_score = classifier.evaluate(x=test_set.data,
37                                       y=test_set.target)["accuracy"]
38 print('Accuracy: {:.3f}'.format(accuracy_score))
39
40 # Classify two new flower samples.
41 new_samples = np.array(
42     [[6.4, 3.2, 4.5, 1.5], [5.8, 3.1, 5.0, 1.7]], dtype=float)
43 y = list(classifier.predict(new_samples, as_iterable=True))
44 print('Predictions: {}'.format(str(y)))
45
46 if __name__ == "__main__":
47     tf.app.run()

```

复制上面的代码到一个文件下载相关的训练和测试数据在同一目录。下面你将更新代码增加采集和监控能力。

## 2.4.2 概览

上一张我们实现了一个神经网络分类器分类 iris 样本为三种类别，但是当代码运行的时候，输出没有跟踪训练进程，仅仅打印处结果：

```

1 Accuracy: 0.933333
2 Predictions: [1 2]

```

没有任何采集，模型就像一个黑盒子，你不能看到 TensorFlow 随着梯度下降发生了什么，模型是否收敛或者是否审查决定是否应该提前停止，处理这个问题的一个方法是分隔模型为多个 fit 调用在更小的时间步获得更精确的评估，然而日常使用不推荐因为它会极大地降低模型的训练。幸运的是 tf.contrib.learn 提供了另一个解：一个Monitor API设计用来帮助你在训练中采集度量和评估你的模型，下面的章节你将学习如何在 TensorFlow 中采集，建

立 ValidationMonitor 做 streaming 评估，用 TensorBoard 可视化你的衡量标准。

### 2.4.3 让你的 TensorFlow 能采集

TensorFlow 有 5 个不同的等级采集消息。分别是 DEBUG, INFO, WARN, ERROR 和 FATAL，当你配置好级别后 TensorFlow 将输出所有和你级别和更高级别的相关消息。例如你设置级别为 DEBUG 你将从上面五个级别得到采集信息。默认，TensorFlow 被被配置的采集级别为 WARN，但是当跟踪模型训练时，你将想要调整级别为 INFO，将提供额外的反馈作为 fit 操作，增加下面行到你的代码：

```
1 tf.logging.set_verbosity(tf.logging.INFO)
```

当你运行代码的时候，你将看到如下额外的采集输出：

```
1 INFO:tensorflow:loss = 1.18812, step = 1
2 INFO:tensorflow:loss = 0.210323, step = 101
3 INFO:tensorflow:loss = 0.109025, step = 201
```

在 INFO 级别采集 tf.contrib.learn 自动每 100 步输出 train-loss metric 到标准输出。

### 2.4.4 配置 Streaming 评估的 ValidationMonitor

采集训练的损失对于帮你理解你的模型是否收敛是很有用的，但是如果你想了解训练过程发生了什么？tf.contrib.learn 提供几个更高级别的 Monitor，你可以添加到你的 fit 操作以进一步在模型训练中记录度量或者调试低级 TensorFlow 操作。包括：

Monitor	描述
CaptureVariable	每 n 步保存一个指定变量的值到集合
PrintTensor	在每个训练步采集指定 tensor 的值。
SummarySave	用 tf.summary.FileWriter 在训练中每 n 步为一个指定的 <a href="#">tf.Summary protocol buffer</a>
ValidationMonitor	在训练中每 n 步采集一个指定的评估方案，如果想要，在确定条件下实现 early stopping。

### 2.4.5 每 N 步评估

对于神经网络分类器你也许想采集训练损失同时像评估测试数据，看看模型的泛化能力。你可以结合配置一个 ValidationMonitor 和测试数据 (test\_set.data,test\_set.target)，用 every\_n\_steps 设置评估的频率。every\_n\_steps 默认值为 100，这里设置 every\_n\_step 为 50 每 50 步评估模型训练。

```

1 validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
2     test_set.data,
3     test_set.target,
4     every_n_steps=50)

```

放这段代码在初始化 classifier 之前。ValidationMonitor 依靠保存 checkpoint 文件指定计算操作, 你将想要修改 classifier 去增加一个包含有 save\_checkpoints\_secs 的 tf.contrib.learn.RunConfig(指定训练过程多少秒保存 checkpoint)。因为 iris 数据集很小, 这样训练和快, 可以设置 save\_checkpoints\_secs 文件为 1 (每一秒保存一个 ckcheckpoint 文件), 确保一个高效的 checkpoint。

```

1 classifier = tf.contrib.learn.DNNClassifier(
2     feature_columns=feature_columns,
3     hidden_units=[10, 20, 10],
4     n_classes=3,
5     model_dir="/tmp/iris_model",
6     config=tf.contrib.learn.RunConfig(save_checkpoints_secs=1))

```

注意 model\_dir 指定了一个可用的目录 (/tmp/iris\_model) 存储模型数据, 这个路径相比自动生成的路径在之后将会很容易被访问, 每次你运行代码的时候任何/tmp/iris\_model 中的数据将被载入, 模型将继续从上次停止的地方运行, 为了重新训练模型在运行代码前先删掉/tmp/iris\_model, 最后添加你的 validation\_monitor, 更新包含 monitor 参数的 fit 调用

```

1 classifier.fit(x=training_set.data,
2                  y=training_set.target,
3                  steps=2000,
4                  monitors=[validation_monitor])

```

当你返回代码, 你应该看到下面输出:

```

1 INFO:tensorflow:Validation (step 50): loss = 1.71139, global_step = 0, accuracy
      = 0.266667
2 ...
3 INFO:tensorflow:Validation (step 300): loss = 0.0714158, global_step = 268,
      accuracy = 0.966667
4 ...
5 INFO:tensorflow:Validation (step 1750): loss = 0.0574449, global_step = 1729,
      accuracy = 0.966667

```

## 2.4.6 用 MetricSpec 自定义评估方案

默认没有评估方案指定, ValidationMonitor 将采集损失和精度, 但是你可以每 50 步自定义度量列表, 为了指定明确的方案你在评估运行时指定明确的方案, 你可以增加一个 metrics 参数到 ValidationMonitor 构造体, metrics 结果一个字典 (key/value) 这里 key 是

你想采集的度量的名字，相对应的值是 MetricSpec 对象，MetricSpec 构造体接受 4 个参数：

- metric\_fn: 函数计算返回度量的值。这可以是 tf.contrib.metric 模型中预定义的可用的函数，像 tf.contrib.streaming\_precision 或者 tf.contrib.metrics.streaming\_recall。你可以定义你的个性化的度量函数（必须接受 predictions 和 labels 作为参数，weights 参数每选择性提供。函数必须返回下面两种格式的值）
  - 一个 tensor
  - 一个操作对 (value\_op, update\_op)，这里 value\_op 返回 metric 值 update\_op 执行一个对应的操作更新内部模块的状态。
- prediction\_key: 包含模型返回的 label 的 tensor，由模型的 input\_fn 指定，正如 prediction\_key，如果 input\_fn 返回一个 tensor 或者单输入的字典，在导航中的 iris 例子，DNNClassifier 没有 input\_fn(x,y 数据直接传递给 fit)，因此不需要提供 label\_key。
- weights\_key: 包含有 metric\_fn 权重输入的 tensor。

下面的代码创建一个 validation\_metric 字典在模型评估中定义三个度量。

- accuracy: 用 tf.contrib.metrics.streaming\_accuracy 作为 metric\_fn。
- precision: 用 tf.contrib.metrics.streaming\_precision 作为 metric\_fn
- recall: 用 tf.contrib.metrics.streaming\_recall 作为 metric\_fn

```

1 validation_metrics = {
2     "accuracy":
3         tf.contrib.learn.MetricSpec(
4             metric_fn=tf.contrib.metrics.streaming_accuracy,
5             prediction_key=tf.contrib.learn.PredictionKey.CLASSES),
6     "precision":
7         tf.contrib.learn.MetricSpec(
8             metric_fn=tf.contrib.metrics.streaming_precision,
9             prediction_key=tf.contrib.learn.PredictionKey.CLASSES),
10    "recall":
11        tf.contrib.learn.MetricSpec(
12            metric_fn=tf.contrib.metrics.streaming_recall,
13            prediction_key=tf.contrib.learn.PredictionKey.CLASSES)
14 }
```

在 ValidationMonitor 构造体前增加上面的代码，然后 ValidationMonitor 构造体增加 metrics 参数去采集 validation\_metrics 中定义的 accuracy, precision 和 recall metrics(损失总是被采集不需要明确的指定)

```

1 validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
2     test_set.data,
3     test_set.target,
4     every_n_steps=50,
5     metrics=validation_metrics)

```

返回代码你应该在你的输出日志中看到 precision 和 recall:

```

1 INFO:tensorflow:Validation (step 50): recall = 0.0, loss = 1.20626, global_step
= 1, precision = 0.0, accuracy = 0.266667
2 ...
3 INFO:tensorflow:Validation (step 600): recall = 1.0, loss = 0.0530696,
global_step = 571, precision = 1.0, accuracy = 0.966667
4 ...
5 INFO:tensorflow:Validation (step 1500): recall = 1.0, loss = 0.0617403,
global_step = 1452, precision = 1.0, accuracy = 0.966667

```

#### 2.4.7 用 ValidationMonitor 提前终止

注意上面的输出每 600 步模型已经得到 precision 和 recall 1.0。抛出问题说明模型是否可以从 early stopping 中获得好处、另外采集运算的度量，当指定条件满足时 Validation-Monitor 容易通过下面的参数实现 early stopping。

参数	描述
early_stopping_metrics	度量在给定的条件 early_stopping_rounds 和 early_stopping_metric_minimize 下出发 early stopping, 默认是"loss"
early_stopping_metric_minimize	如果想要模型行为最小化 early_stopping_metric 为 True, 如果想最大化它的值为 False, 默认为 True
early_stopping_rounds	如果 early_stopping_metric 不见效或者增加, 训练将被停止, 默认为 None 表示永不提前停止

按照下面修改 ValidationMonitor 构造体, 指定是否损失 200 步 (early\_stopping\_rounds=200) 没有减少, 模型训练在到达这个点将立即终止, 并不完成 fit 中指定的 2000 步。

```

1 validation_monitor = tf.contrib.learn.monitors.ValidationMonitor(
2     test_set.data,
3     test_set.target,
4     every_n_steps=50,
5     metrics=validation_metrics,
6     early_stopping_metric="loss",

```

## 2.5. TENSORBOARD HISTOGRAM DASHBOARD

```
7     early_stopping_metric_minimize=True,  
8     early_stopping_rounds=200)
```

如果模型训练提前结束将返回下面的代码:

```
1 ...  
2 INFO:tensorflow:Validation (step 1150): recall = 1.0, loss = 0.056436,  
      global_step = 1119, precision = 1.0, accuracy = 0.966667  
3 INFO:tensorflow:Stopping. Best step: 800 with loss = 0.048313818872.
```

特别是这里的训练步数为 1150，对于之前 200 步，损失不在减少，总体 800 产生最小的损失与之对应测试数据集。这意味着额外的超参数标准被减少也许将进一步提升模型。

### 2.4.8 用 TensorBoard 可视化采集数据

读 ValidationMonitor 生成的日志提供了丰富的训练过程中的原始数据，可视化这些数据有助于得到更深的见解。例如精确度如何随着步数改变。你可以用 TensorBoard 设置，命令行参数 logdir 画图，运行下面的命令行 tensorboard --logdir=/tmp/iris\_model/浏览器中导航到 <http://0.0.0.0:6006>，如果你点击精确度范围，你将看到一个像下面的图，绘制精度和步数。



## 2.5 TensorBoard Histogram Dashboard

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

### 2.5.1 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。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，这里均值随着时间增大。

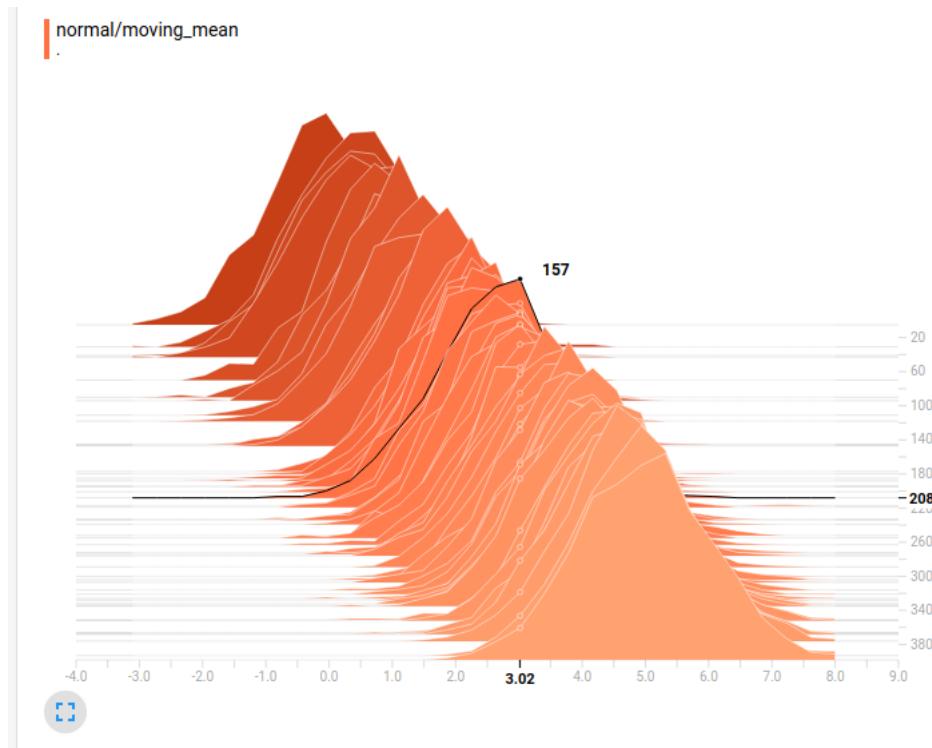
```
1 import tensorflow as tf  
2 k = tf.placeholder(tf.float32)
```

## 2.5. TENSORBOARD HISTOGRAM DASHBOARD

```
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
```



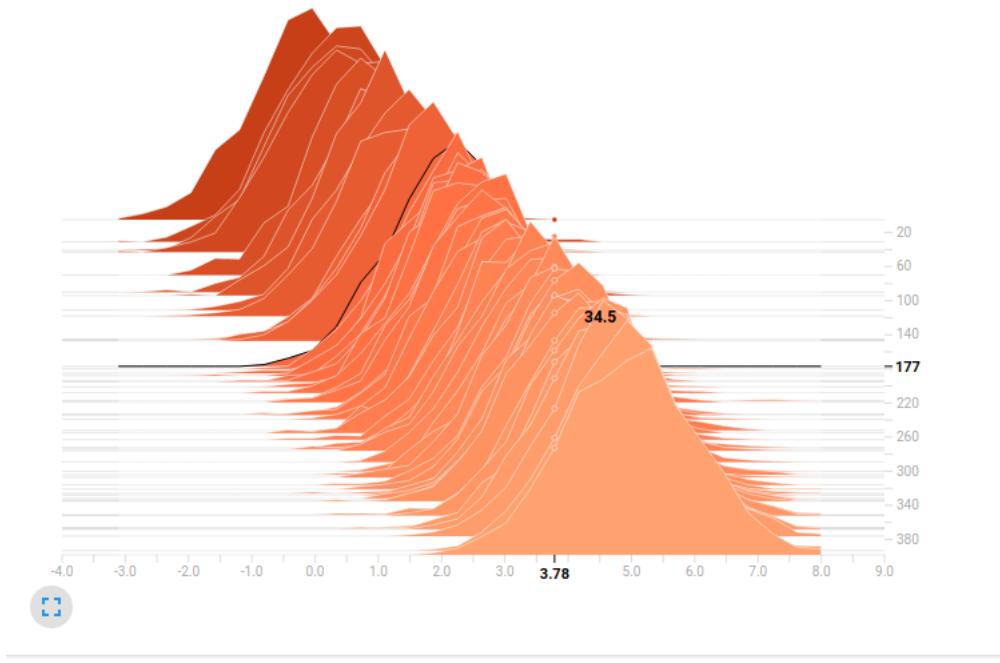
`tf.summary.histogram` 接受任一尺寸和大小的 Tensor，压缩它们进入直方数据结构组成一些小的数据宽度和数量组层的 bin 将该够，例如我们像组成数 [0.5,1.1,1.3,2.2,2.9,2.99] 成 3 个 bin，我们可以创建三个 bin：一个包含 0 到 1 之间的一切 (0.5)，一个包含 1-2(1.1,1.3) 之间，一个包含 2-3(2.2,2.9,2.99)

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，甚至它们有相同的元素，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。

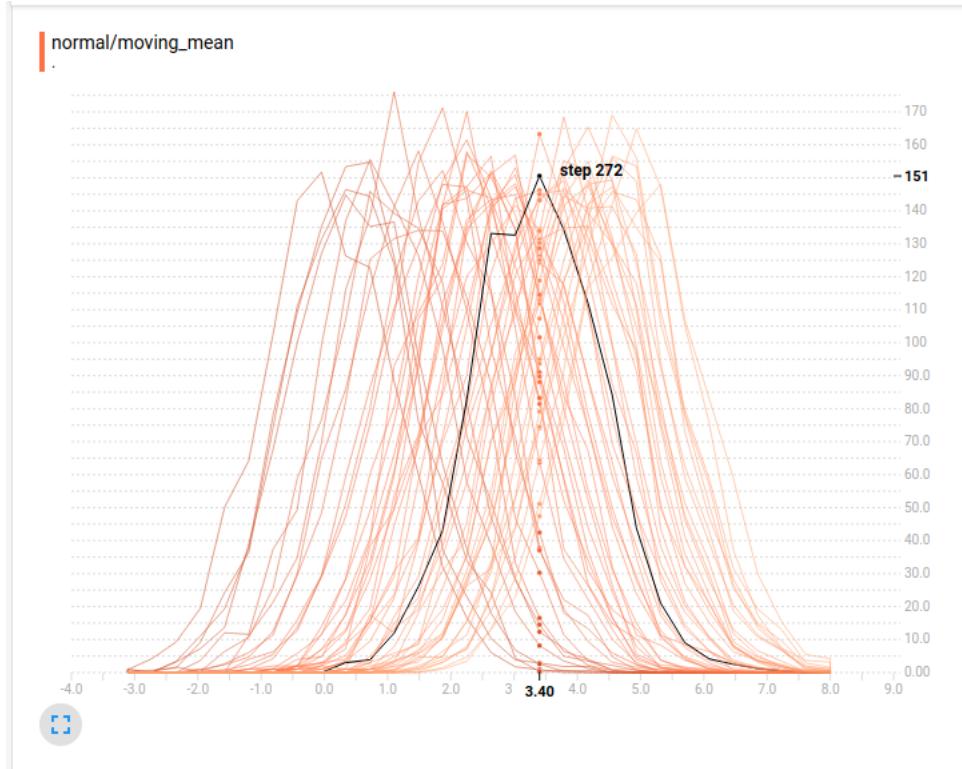


你也许注意到注意到直方切片在统计步数和时间上不总是偶数，这是因为 TensorBoard 用[reservoir sampling](#)保持直方图的子集，为了节约内存，Reservior sampling 保证每个采样有一个相等的可能性被包含进去，但是因为它时一个随机算法，采样并不在每个偶数步发生。

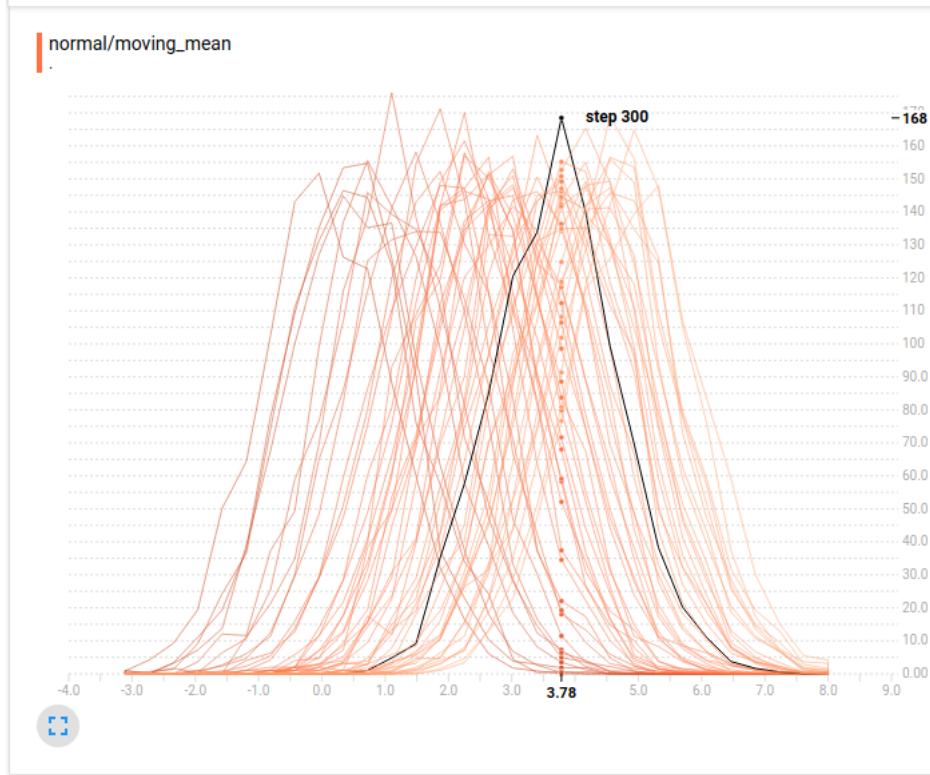
### 2.5.2 Overlay Mode

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，而是所有的图华仔一个相同的 y 轴上。

## 2.5. TENSORBOARD HISTOGRAM DASHBOARD



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，浅色线时最近的新的时间不，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。

### 2.5.3 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：

```

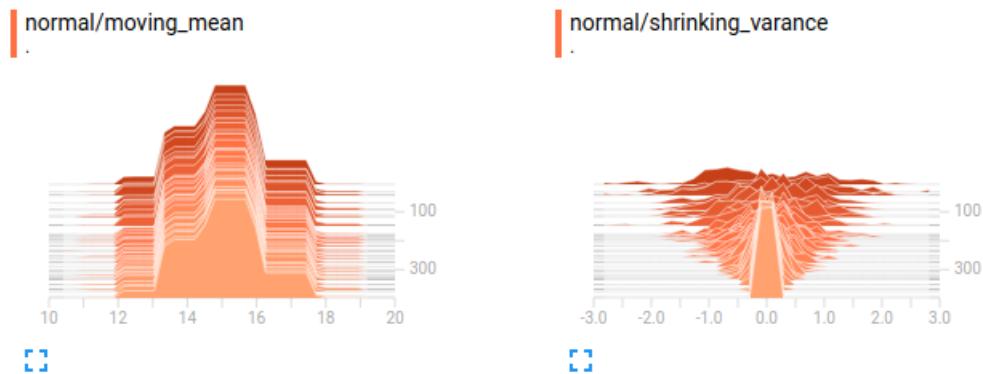
1 import tensorflow as tf
2 k = tf.placeholder(tf.float32)
3 mean_moving_normal = tf.random_normal(shape=[1000],mean=(3*5),stddev=1)
4 tf.summary.histogram('normal/moving_mean',mean_moving_normal)
5 variance_shrinking_normal = tf.random_normal(shape=[100],mean=0,stddev=1-(k))
6 tf.summary.histogram('normal/shrinking_varance',variance_shrinking_normal)
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):

```

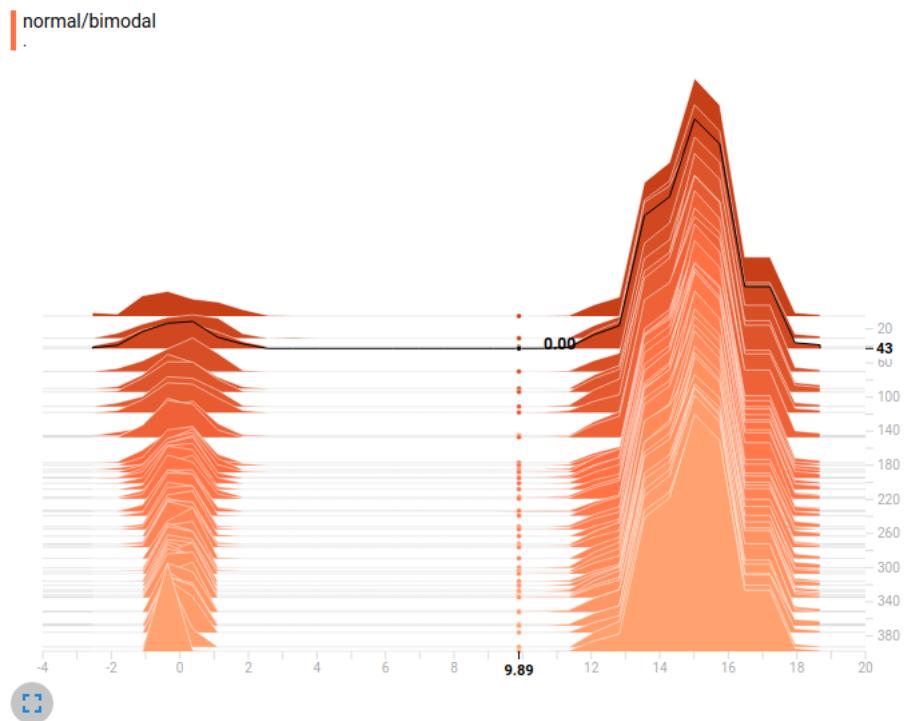
## 2.5. TENSORBOARD HISTOGRAM DASHBOARD

```
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：



### 2.5.4 更多分布

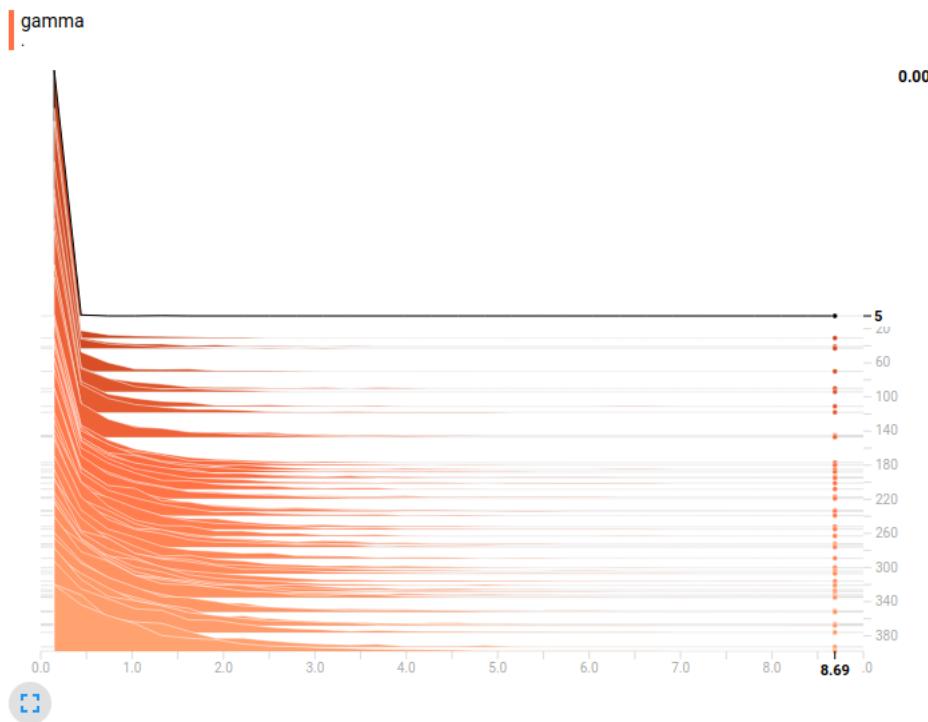
生成可视化更多分布，结合它们到表中：

```
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)
```

---

## 2.5. TENSORBOARD HISTOGRAM DASHBOARD

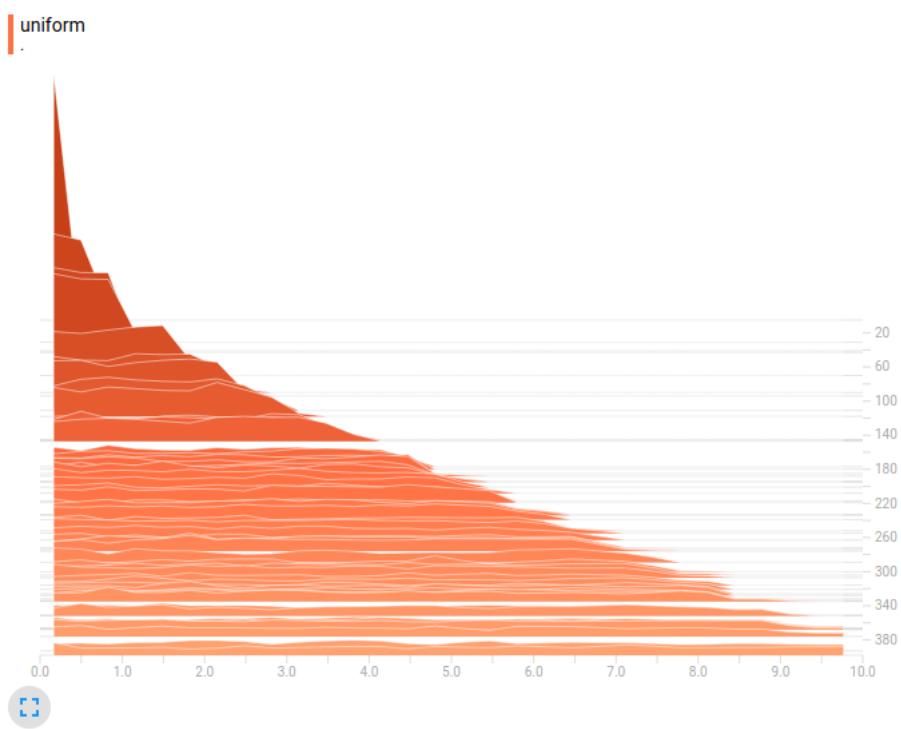
---



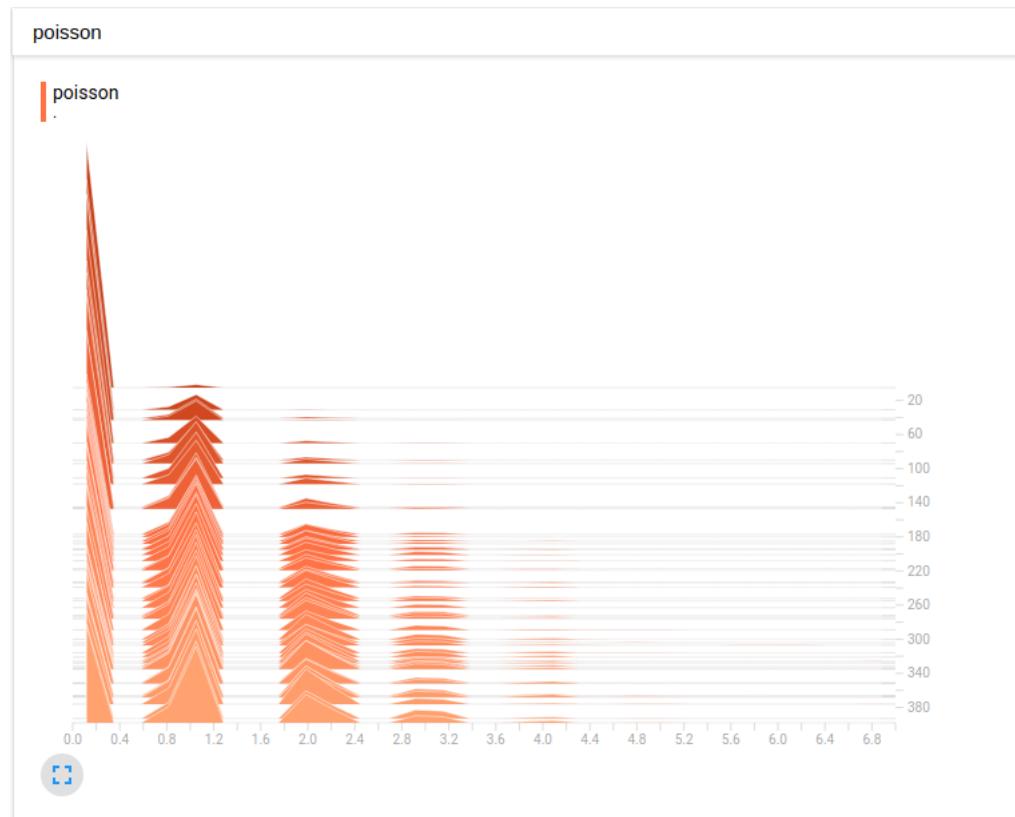
---

## 2.5. TENSORBOARD HISTOGRAM DASHBOARD

---

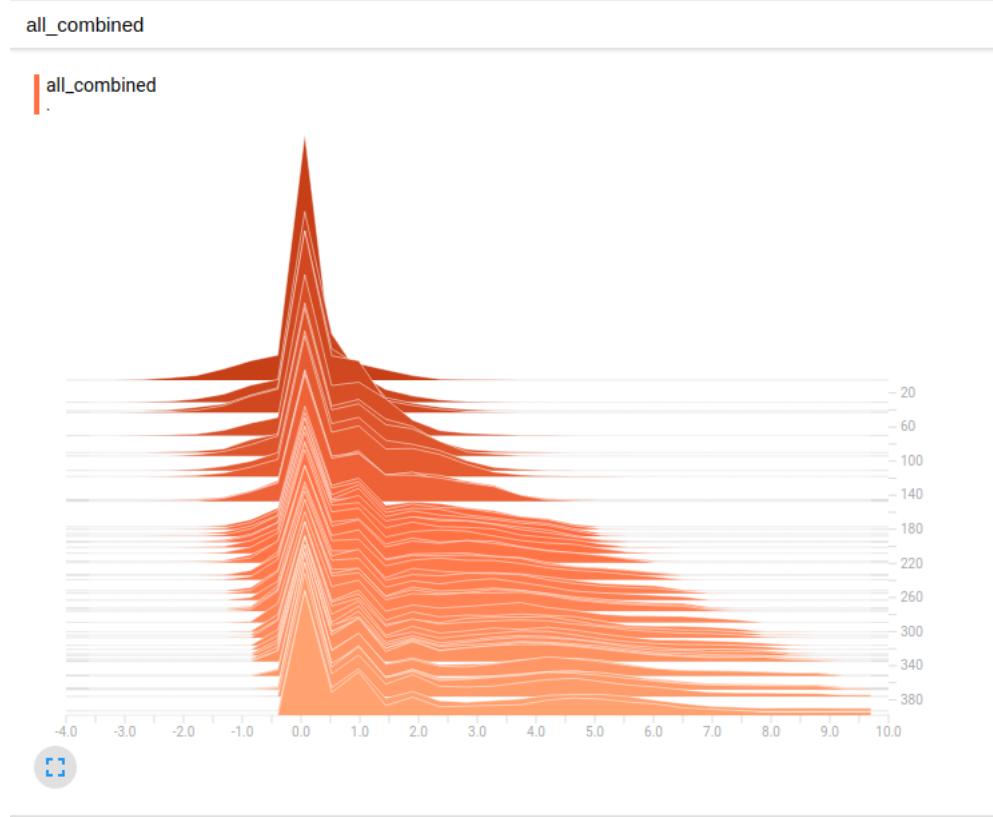


### 2.5.5 poisson 分布



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。

### 2.5.6 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=%4f'%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=%4f'%l,fontdict={'size':15,'color':'red'})
47         plt.show()
48     save()
49     tf.reset_default_graph()
50     reload()

```

# Chapter 3

## 程序员向导

### 3.1 Estimators

这篇文章介绍简化机器学习程序的高级 TensorFlow API [Estimators](#)。Estimator 封装下面的行为：

- training
- evaluation
- prediction
- export for serving

你既可以用我们提供的预先定义好的 Estimator 也可以写你自己的自定义的 Estimator。所有的自定义的或预先提供的 Estimators 都基于[tf.estimator.Estimator](#)类。

注意：TensorFlow 在[tf.contrib.learn.Estimator](#) 包含一个你不应该使用的废弃的 Estimator。

#### 3.1.1 高级 Estimator

Estimators 提供下面的好处：

- 你可以不用改变你的模型在本地主机或者多服务器环境下运行 Estimator-based。更进一步，你可以在 CPU, GPU 或者没有记录你的模型的 TPU 运行 Estimator-base 的模型
- Estimators 简化了模型开发者回见的共同实现

- 你可以用高级直接代码开发一个艺术模型的状态，简单来讲，通常用 Estimator 创建模型比 TensorFlow 低级 API 要更简单。
- Estimators 本身构建在简化自定义的 tf.layers 中
- Estimators 为你构建一个图。换句话说，你不用构件图
- Estimator 提供了一个安全的分布式的训练循环控制如何或者什么时候做：
  - 构建图
  - 初始化变量
  - 开始队列
  - 处理异常
  - 创建 checkpoint 文件并且从失败中恢复
  - 为 TensorBoard 保存总结文件

当用 Estimator 写一个应用时，你必须从模型中分开输入 pipeline。分开简化了在不同数据集上的试验。

### 3.1.2 自定义的 Estimator

自定义的 Estimator 使得你能在高于 TensorFlow APIs 的高级概念级别上工作。你不用担心创建计算图和会话因为 Estimator 为你处理所有的 plumbing。预定义的 Estimator 为你创建管理图和会话对象。进一步，预定义的 Estimators 让你做最小的代码改动结合不同的模型架构试验。DNNClassifier，例如一个自定义的 Estimators 类通过 dense，前馈神经网络训练分类模型。

### 3.1.3 预定义的 Estimator 程序结构

一个预先定义的 Estimator 的 TensorFlow 程序通常由四步组成：

1. 写一个或者更多的数据集导入函数。例如也许创建一个函数导入训练集创建另一个函数导入测试集。每个数据集导入函数必须有两个对象：

- 一个包含对应特征数据的 key 为特征名字的 value 是 Tensor 的词典
- 一个 Tensor 包含一个或者更多标签

例如，下面的代码吧、描述了基本的输入函数框架：

```
1 def input_fn(dataset):
2     ... # manipulate dataset, extracting feature names and the label
3     return feature_dict, label
```

(查看[Importing Data](#)获取详细信息)

2. 定义特征列。每个`tf.feature_column`确定特征名字，它的类型和输入预处理。例如，下面的代码创建三个包含整数和浮点数据的特征列。三个特征列也指定一个 lambda 程序调用原始数据的缩放：

```

1 # Define three numeric feature columns.
2 population = tf.feature_column.numeric_column('population')
3 crime_rate = tf.feature_column.numeric_column('crime_rate')
4 median_education = tf.feature_column.numeric_column('median_education',
5           normalizer_fn='lambda x: x - global_education_mean')

```

3. 实例化相关的预定义 Estimator。例如有一个预先定义的名字为 `LinearClassifier` 的 Estimator 实现：

```

1 # Instantiate an estimator, passing the feature columns.
2 estimator = tf.estimator.Estimator.LinearClassifier(
3     feature_columns=[population, crime_rate, median_education],
4 )

```

4. 调用一个训练，评估，推理方法。例如，所有的 Estimator 提供一个训练方法训练模型。

```

1 # my_training_set is the function created in Step 1
2 estimator.train(input_fn=my_training_set, steps=2000)

```

### 3.1.4 预定义 Estimators 的好处

预定义的 Estimators 编码最佳实践提供下面的好处：

- 最佳决定不同计算图的不同部分应该运行在单台机器或者集群上实现方案的实践
- 最佳事件 (总结) 和普遍的有用的总结时间

如果你不用自定义的 Estimator，你自己必须实现上面的特征。

### 3.1.5 自定义 Estimators

每个 Estimator (预定义或者是自定义) 的核心是模型函数，它包含了构建训练，评估，预测图的方法。然后当你使用一个预定义的 Estimator 有些人已经实现了模型函数，当依赖一个自定义的 Estimator 时，你必须自己写模型函数。一个[companion document](#) 解释如何写一个模型函数。

### 3.1.6 推荐的工作流程

推荐的工作流程如下:

1. 假设存在一个合适的预定义的 Estimator，用它构建你的第一个模型，用它的结果建立一个 baseline
2. 构建测试你的 pipeline，包含使用这个预定义的 Estimator 的关于你的数据的完整性和真实性
3. 如果合适的 Estimator 可用，运行试验决定哪个 pre-made Estimator 产生最好的结果
4. 可能的话，进一步提高你的模型构建你自定义的 Estimator

### 3.1.7 从 Keras 模型创建 Estimator

你可以转化已经存在的 Keras 模型为 Estimator。这样做能使得你的 Keras 模型获得 Estimator's 的力量，想分布式的训练。如下面样例调用 `tf.keras.estimator.model_to_estimator`

```

1 # Instantiate a Keras inception v3 model.
2 keras_inception_v3 = tf.keras.applications.inception_v3.InceptionV3(weights=None
   )
3 # Compile model with the optimizer, loss, and metrics you'd like to train with.
4 keras_inception_v3.compile(optimizer=tf.keras.optimizers.SGD(lr=0.0001, momentum
   =0.9),
5                               loss='categorical_crossentropy',
6                               metric='accuracy')
7 # Create an Estimator from the compiled Keras model.
8 est_inception_v3 = tf.keras.estimator.model_to_estimator(keras_model=
   keras_inception_v3)
9 # Treat the derived Estimator as you would any other Estimator. For example,
10 # the following derived Estimator calls the train method:
11 est_inception_v3.train(input_fn=my_training_set, steps=2000)
```

更多细节查看 `tf.keras.estimator.model_to_estimator`。

## 3.2 Tensor

Tensorflow 正如它的名字表达的一样是定义 tensor 的计算。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

### 3.2.1 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
```

### 3.2.2 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。
```

### 3.2.3 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 `[]` 获取一个 `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]
```

### 3.2.4 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，每一维的形状可能知道可能不知道。

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 <sub>n-1</sub> ]	N 维 tensor 的形状 [D <sub>0</sub> ,D <sub>1</sub> ,...,D <sub>n-1</sub> ]	

### 3.2.5 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` 属性得到它的形状。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，这里显示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])
```

### 3.2.6 改变 Tensor 的形状

tensor 的元素个数是所有形状值的乘积。标量的元素形状总是 1. 因此，因为有相同元素不同形状的 tensor，转变它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!
```

### 3.2.7 数据类型

`tf.Tensor` 不可能有一个以上的数据类型。然而序列化任意数据结构作为字符串存储在 `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。

### 3.2.8 评价 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 sess = tf.Session()
2 constant = tf.constant([1,2,3])
3 tensor = constant*constant
4 print(tensor.eval(session=sess))

```

eval 方法仅仅当 tf.Session() 被激活时可用。Tensor.eval 然后会得到一个和 tensor 相同数据类型的 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.

```

其它的模型结构在计算 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 依赖于队列中的值，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。

### 3.2.9 打印 Tensor

出于调试目的，你想要打印 tf.Tesor 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。

## 3.3 Variable

Tensorflow 变量是在你的程序中表现共享，永久状态的最好的方法，Vaiables 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它的值可能被改变。

和 tf.Tensor 不同在于 tf.Variable 存在于 session.run 之外。一个 tf.Vaiable 存储永久 tensor, 指定操作允许你读和修改它的值, 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。

### 3.3.1 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, 与 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。

```
1 my_variable = tf.get_variable("my_int_variable", [1, 2, 3], dtype=tf.int32,
                               initializer=tf.zeros_initializer)
```

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]))
```

所以当你用 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 Tensor 的形状。

### 3.3.2 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它们的方法。因此 TensorFlow 提供了 collections(集合) 代表有名字的 tensor 列表或者其它对象, 像 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.GraphKeys.
    LOCAL_VARIABLES])
2 # TensorFlow1.4 后续版本使用
3 # my_local = tf.get_variable("my_local", shape=(), collections=[tf.GraphKeys.
    GLOBAL_VARIABLES])
```

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")
```

### 3.3.3 配置设备

像任何其它 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，在最坏的情况下让每个 worker 独立的复制每个变量。为此我们提供了 tf.train.replica\_device\_setter。自动放置变量到参数 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
```

### 3.3.4 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，因为它让你从 checkpoint 载入模型不用重复运行代价高昂的初始化器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.GLOBAL\_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时，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（用一个变量值的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)
```

### 3.3.5 用变量

为了在 TensorFlow 图中使用 `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，任何时候一个用在表达式中的变量自动转化一个tf.Tensor到它的值。
```

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 tf.global_variables_initializer().run()
4 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()
```

### 3.3.6 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。这个构造器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`。

### 3.3.7 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。

### 3.3.8 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.assign(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(session = sess)
14     dec_v2.op.run(session = sess)
15     save_path = saver.save(sess, './model.ckpt')
16     print("Model saved in file:%s"%save_path)

```

### 3.3.9 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它们。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())

```

### 3.3.10 选择变量恢复

如果你不传递任何参数给 `tf.train.Saver()`, saver 处理图上所有的变量。每个变量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 恢复前面 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时恢复变量的一个子集, 你必须对其它变量执行初始化操作。

### 3.3.11 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 Veriable scope 允许你控制变量重用调用函数, 隐式的创建使用变量。它也允许你在你的文件结构上命名你的变量以方便理解。

```

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", biase_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, 我们想要一些卷基层, 重复的调用这些函数将 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，并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)

```

因为用来提取 scope 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)

```

## 3.4 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。这导在低级的编程模型中你首先定义数据流图，然后创建一个 TensorFlow 会话在本地或者远程设备上运行图的一部分。

这个向导对于你想直接用低级编程模型是很有用的。更高级的 API 像 `tf.estimator.Estimator` 和 Keras 对于用户端隐藏了图和会话的细节。但是如果你想明白这些 API 是如何实现的，这个向导也许很有用。

### 3.4.1 为什么用数据流图 ?

数据流图对于并行编程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 分割你的程序到不同的机器上的设备 (CPUs, GPUs, TPUs) 上是可能的，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代码，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。

### 3.4.2 什么是 `tf.Graph`

一个 `tf.Graph` 包含两个相关的种类信息：

- Graph structure: 图的节点和边，只是单个操作如何被组合在一起，但是不是但是没有规定他们如何使用。图结构想一个集合代码：查看它可以传递有用的信息，但是它不包含源代码传递的所有有用信息。
- Graph collection: TensorFlow 提供一个通常的机制存储在 `tf.Graph` 中的 metadada 集合。`tf.add_to_collection` 函数是你结合一个有 key 的 (这里 `tf.GraphKeys` 定义一些标准的 key) 对象列表。一些 TensorFlow 库的一部分使用这设备：例如当你创建一个 `tf.Variable`，这被默认添加到代表全局变量和可训练变量的集合中。当你后来创建一个 `tf.train.Saver` 或者 `tf.train.Optimizer`，这个变量在集合中用作默认参数。

### 3.4.3 建立一个 `tf.Graph`

大多数的 TensorFlow 的开始阶段构造一个数据流图，在这个阶段，你利用 TensorFlow 的 API 函数构造 `tf.Operation`(节点) 和 `tf.Tensor`(边) 对象，添加它们到 `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，返回一个代表这个常量值的 `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)` 添加到图上，一个操作将存储一个可写的 `tensor` 变量在 `tf.Session.run` 调用前保存值。变量对象包装这个操作，可以像一个 `tensor` 使用，将读当前存储得值。`tf.Variavle` 对象也有像 `assign` 和 `assign_add` 方法创建 `tf.Operation` 对象，当执行的时候更新存储值。（查看 [Variable](#) 获取关于变量的更多信息）
- 调用 `tf.train.Optimizer.minimize` 将添加操作和 `tensor` 到默认的图上计算梯度，返回一个 `tf.Operation`，当运行的时候，使用这些题都到一些变量上。

多数程序依赖于默认的图，查看 [处理多图](#) 获取更多高级使用情况。高级的想 `tf.estimator.Estimator` API 管理你的行为上的默认的图，for example 也许创建不同的图训练和评估。

**注意:** 调用在 TensorFlow API 中多数函数添加操作和 `tensor` 到默认的图上，但是不执行实际的操作。相反，你组合这些韩勇直到你有一个 `tf.Tensor` 胡哦哦这 `tf.Operation` 表示总体积算（想执行梯度下降和传递对象到 `tf.Session` 执行计算。）

#### 3.4.4 命名操作

`tf.Graph` 对象给它包含的 `tf.Operation` 对象定义了一个 namespace。TensorFlow 自动为你图上的操作选择一个独一无二的名字，而且指定操作名字使程序易读和方便调试。TensorFlow API 提供了两个操作来覆盖操作的名字：

- 每个 API 函数在创建一个新的 `tf.Operation` 或者返回一个新的 `tf.Tensor` 时接收一个 `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="c") #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")

```

图的可视化使用 `name scope` 组织操作减少图可视化的复杂度。查看 [可视化你的图](#) 了解更多信息。

注意 tf.Tensor 对象在 tf.Operation 后被隐含的声明产生 tensor 作为输出。一个 tensor 名字来自”<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。

### 3.4.5 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来在一个特别的上下文中请求所有的操作放置在相同的设备上。指定格式如下:

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

这里:

- <JOB\_NAME> 是一个 alpha 数字, 不是以数字开头
- <DEVICE\_TYPE> 是一个注册的设备。
- <TASK\_INDEX> 一个非负整数代表 job 中的任务的索引
- <JOB\_NAME> 查看[tf.train.ClusterSpec](#) 查看更多关于 jobs 和 tasks 的解释。
- <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不同的 GPU。

你不需要指定设备的每一部分, 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, 你也许指定 job 的名字和 task ID 放置变量到参数服务器 job(”/job:ps”), 其它的操作到 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` 给你很高的灵活度来选择放置单个操作或者更大范围的 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` 放置操作 data-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` 对象和其它操作:

```

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"

```

### 3.4.6 Tensor-like 对象

一些 TensorFlow 操作接收一个或者更多的 `tf.Tensor` 对象作为参数。例如，`tf.matmul` 得到 `tf.Tensor` 对象，`tf.add_n` 得到一个 `tf.Tensor` 列表对象。为了方便使用这些函数在 `tf.Tensor` 中接收一个 tensor-like 对象，用 `tf.convert_to_tensor` 方法转换它为 `tf.Tensor`，Tensor-like 包含下面的元素类型:

- `tf.Tensor`
- `tf.Variable`
- `numpy.ndarray`
- `list(tensor-like 对象的列表)`
- Python 标量:`bool, float, int, str。`

你可以用[tf.register\\_tensor\\_conversion\\_function](#)。

默认，每次你用相同的 tensor-like 对象 TensorFlow 将创建一个新的 `tf.Tensor`。如果 tensor-like 对象很大 (`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.convert_to_tensor` 在 tensor-like 对象，用 `tf.Tensor` 返回。

### 3.4.7 在 `tf.Session` 执行图

TensorFlow 用 `tf.Session` 类代表客户程序（通常是 Python 程序），尽管类似的接口其他语言中 (C++) 也可用，一个 `tf.Session` 对象提供访问在本地机器上的设备，分布式的

TensorFlow 运行环境中访问远程设备。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。

### 3.4.8 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), 它是通常用作上下文管理 (with), 自动关闭会话。但是你应该确定调用 tf.Session.close 在程序结束后释放资源。

高级 API 像 tf.train.MonitoredTrainingSession 或者 tf.estimator.Estimator 将为你创建和管理一个 tf.Session。APIs 接收 target 和 config 参数 (或者作为 tf.estimator.RunConfig 的一部分), 含义如下: tf.Session.\_\_init\_\_ 接收三个参数:

- 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 server 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连接 tf.train.Server。[distributed TensorFlow](#) 步数向导描述其他的常见情形。
- 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 (查看[Programming with multiple graphs](#) 获取更多详情) 中用多个图, 你可以在创建会话的时候指定 tf.Graph。
- config 这个参数允许你指定 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 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 设备上, 然后放他们在 CPU 上。
- cluster\_def: 当用分布式的 TensorFlow, 这个选项允许你指定用于计算的机器, 提供不同 job 名字的映射任务索引和网络地址。查看[tf.train.ClusterSpec.as\\_cluster\\_def](#) 获取更多详情
- 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 显存分配以至于它的随着现存分配渐渐增长, 而不是启动的时候分配尽量多的内存。

### 3.4.9 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，可以传递一个或者更多的 `tf.Operation` 和 `tf.Tensor` 或者 `tensor-like` 类型的对象，像 `tf.Variable`。这些 `fetch` 决定了 `tf.Graph` 的使用与计算 `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能操作不同的子图执行：

```

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` 字典，字典映射 `tf.Tenor`(常见的 `tf.placeholder()`) 对象为合适执行的值(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` 也接收一个选项 `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=metdadada)
10    # 打印每个设备执行的子图
11    print(metadata.partition_graphs)

```

### 3.4.10 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。但是没有描述它们如何被使用，图的结构像集合代码，查看它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，`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`, 它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，包含所有操作的描述和它们之间的边。`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.save` 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` 建议用于高级应用。例如用 `tf.train.MetaGraphDef` Saved-Model library 包装 `tf.Graph` 和一系列训练模型参数以至于它们能被后续服务使用。

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 python `tf.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`。
- 你可以传递它们的名字到 `return_elements` 列表从导入的图中返回 `tf.Tensor` 或者 `tf.Operation` 对象

### 3.4.11 可视化你的图

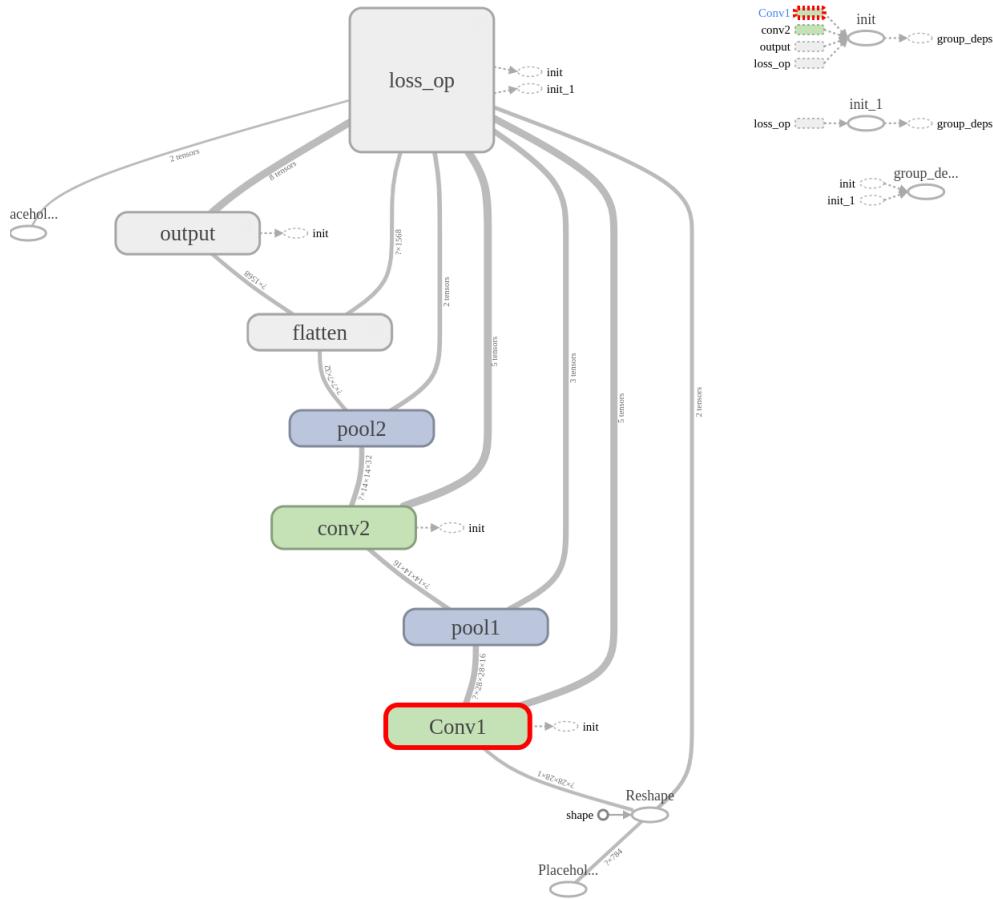
TensorFlow 提供一个工具帮助你理解你图中的代码。图的可视化是 TensorBoard 的一个组件它在你的浏览器中生成你的图的结构。最简单的创建可视化的方法是当创建 `tf.summary.FileWriter` 时传递 `tf.Graph`

```

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` 时, 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, 查看你的图的高级可视化结果。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。图可视化操作利用 `scope` 的名字分组相关的操作到高级节点。你可以点击橙色的 + 按钮展开里面的子图。



### 3.4.12 用多图编程

注意当训练一个模型的时候，通常的使用你的代码的方式是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.train.Saver` 用 `tf.Variable` 对象的名字在不同的 checkpoint 中识别。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图，下面的情况在高级使用情况可能很有用：

- 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 将通过添加”`_1`”, ”`_2`” 形成独一无二的名字，

因此如果它们的名字已经被传出去了，用多个操作创建图给你对操作的名字更好的控制。

- 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 建立子图可能更高效，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())

```

## 3.5 保存和恢复

这章解释如何保存和恢复变量和模型。

### 3.5.1 保存和恢复变量

一个 TensorFlow 变量提供通过你的程序表示和操作共享，永久状态的最佳方法。（查看[Variable 获取详情](#)）这章解释如何保存和恢复变量，注意，Estimator 自动保存和恢复变量（在 model\_dir）

tf.train.Saver 类提供方法保存和恢复模型。tf.train.Saver 构造体提供方法保存和恢复模型。tf.train.Saver 构造体添加 save 和 restore 操作到图上，或者一个指定的列表图上的变量。Saver 对象提供方法运行这些操作，指定 checkpoint 文件写入或者读取的路径。

saver 将保存所有的你的模型上定义的变量。如果你载入一个模型不知道如何构建他的图（例如，如果你编写一个常规程序载入模型），然后读保存和恢复模型概览章节。

TensorFlow 保存变量在二进制的 checkpoint 文件，通俗讲，映射名称到 tensor 值。

### 3.5.2 保存变量

结合 `tf.train.Saver()` 创建一个 Saver 关喀模型中的所有变量。例如，下面的代码段展示如何调用 `tf.train.Saver.save` 方法保存变量到 checkpoint 文件：

```

1 # Create some variables.
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
5 inc_v1 = v1.assign(v1+1)
6 dec_v2 = v2.assign(v2-1)
7
8 # Add an op to initialize the variables.
9 init_op = tf.global_variables_initializer()
10
11 # Add ops to save and restore all the variables.
12 saver = tf.train.Saver()
13
14 # Later, launch the model, initialize the variables, do some work, and save the
15 # variables to disk.
16 with tf.Session() as sess:
17     sess.run(init_op)
18     # Do some work with the model.
19     inc_v1.op.run()
20     dec_v2.op.run()
21     # Save the variables to disk.
22     save_path = saver.save(sess, "/tmp/model.ckpt")
23     print("Model saved in file: %s" % save_path)

```

### 3.5.3 恢复变量

`tf.train.Saver` 对象不仅保存变量到 checkpoint 文件，也恢复变量。注意当你从文件恢复变量时你不是必须提前初始化他们。例如，下面的代码段展示如何调用 `tf.train.Saver.restore` 方法从 checkpoint 恢复变量。

```

1 tf.reset_default_graph()
2
3 # Create some variables.
4 v1 = tf.get_variable("v1", shape=[3])
5 v2 = tf.get_variable("v2", shape=[5])
6
7 # Add ops to save and restore all the variables.

```

```

8 saver = tf.train.Saver()
9
10 # Later, launch the model, use the saver to restore variables from disk, and
11 # do some work with the model.
12 with tf.Session() as sess:
13     # Restore variables from disk.
14     saver.restore(sess, "/tmp/model.ckpt")
15     print("Model restored.")
16     # Check the values of the variables
17     print("v1 : %s" % v1.eval())
18     print("v2 : %s" % v2.eval())

```

### 3.5.4 选择什么变量存储和恢复

如果你没有传递任何参数给 `tf.train.Saver()`, `saver` 处理所有的在图中的变量。每个变量被存储在变量创建的传递的名字下。

明确为 checkpoint 文件中的变量指定名字有事很有用。例如，你也许有训练的模型的变量名字为”weights”，你想恢复他的值为一个名称为”params”的变量。

有时仅仅在保存和恢复用于模型的变量的子集时有用。例如，你已经训练了一个 5 层神经网络，你现在想训练一个新的 6 层模型重用存在的 5 层训练的权重。你可以用 `saver` 恢复前五层的权重。

你可以按照下面通过传递 `tf.train.Saver()` 结构体容易的自定名字和变量保存和载入变量:

- 一个变量列表 (将存储在他们的名字下)
- 一个 Python 字典，`keys` 是使用的名字和是管理的变量

稍后继续保存/恢复例子:

```

1 tf.reset_default_graph()
2 # Create some variables.
3 v1 = tf.get_variable("v1", [3], initializer = tf.zeros_initializer())
4 v2 = tf.get_variable("v2", [5], initializer = tf.zeros_initializer())
5
6 # Add ops to save and restore only 'v2' using the name "v2"
7 saver = tf.train.Saver({"v2": v2})
8
9 # Use the saver object normally after that.
10 with tf.Session() as sess:
11     # Initialize v1 since the saver will not.
12     v1.initializer.run()

```

```

13 saver.restore(sess, "/tmp/model.ckpt")
14
15 print("v1 : %s" % v1.eval())
16 print("v2 : %s" % v2.eval())

```

注意:

- 如果你需要保存和恢复模型变量的不同子集，你可以创建一些 Saver 对象。同样的变量可以在多个 saver 对象中被坚持；他得知仅仅当 Saver.restore() 方法运行时改变。
- 如果你仅仅在会话开始时恢复一个模型变量的子集，你必须为其他变量运行一个初始化操作。查看[tf.variables\\_initializer](#)获取更多信息。
- 为了查看在 checkpoint 中的变量，你可以使用[inspect\\_checkpoint](#)库，特别是 print\_tensors\_in\_checkpoint 函数
- 默认。Saver 为每个变量使用[tf.Variable.name](#)的值。然而，当你创建一个 Saver 对象的时候，你也许小可为在 checkpoint 文件中的变量选择名字。

### 3.5.5 保存和恢复模型概览

当你想保存和载入变量，图和图的 metadata-basically，当你想保存和恢复你的默认，我们推荐使用 SavedModel。SavedModel 是一个语言中立，可恢复的，封闭的序列化。SavedModel 使得高级系统和工具产生，消耗，变换 TensorFlow 模型可行，包含 tf.saved\_model APIs，Estimator APIs 有一个 CLI。

### 3.5.6 APIs 构建和载入一个 SavedModel

这章集中注意力在 SPIs 用于构建和载入一个 SavedModel，特别是当使用低级 TensorFlow APIs 的时候。

### 3.5.7 构建一个 SavedModel

我们提供一个 SavedModel [builder](#)的 Python 实现。SavedModelBuilder 类提供函数保存多个 MetaGraphDef。一个 MetaGraph 是一个数据流图，加他的关联的变量，assert 和签名。一个 MetaGraphDef 是表示 MetaGraph 的 proto buffer。一个 signature 是来自图的输入和输出集合。

如果声明需要被保存和写入或者复制到磁盘，他们可以被提供然后首先 MetaGraphDef 被添加。如果多 MetaGraphDef 关联一个同样名字的 assert，仅仅第一个版本保留。

每个 MetaGraphDef 添加到 SavedModel 必须被用户指定的 tags 注释。tags 提供一个均值企鹅报指定的 MetaGraphDef 载入和恢复没见着变量的共享集合和 assert。这些

tags 通常注释一个 MetaGraphDef 集合他的功能 (例如 servinf 和训练), 可选的硬件指定 aspect(例如 GPU)。例如, 下面的代码按时一个典型的使用 SavedModelBuilder 去构建一个 SavedModel:

```

1 export_dir = ...
2 ...
3 builder = tf.saved_model_builder.SavedModelBuilder(export_dir)
4 with tf.Session(graph=tf.Graph()) as sess:
5     ...
6     builder.add_meta_graph_and_variables(sess,
7                                         [tag_constants.TRAINING],
8                                         signature_def_map=foo_signatures,
9                                         assets_collection=foo_assets)
10 ...
11 # Add a second MetaGraphDef for inference.
12 with tf.Session(graph=tf.Graph()) as sess:
13     ...
14     builder.add_meta_graph([tag_constants.SERVING])
15 ...
16 builder.save()

```

subsection 在 Python 中载入一个 SavedModel Python 版本的 SavedModel 载入器为 SavedModel 提供载入和恢复能力。载入操作要求下面的信息:

```

1 \item 在会话中恢复图定义和变量
2 \item tags 用于确保MetaGraphDef载入
3 \item SavedModel目录

```

在载入时, 变量自己 assets, 签名用作指定 MetaGraphDef 将被存储在应用 session。

```

1 export_dir = ...
2 ...
3 with tf.Session(graph=tf.Graph()) as sess:
4     tf.saved_model.loader.load(sess, [tag_constants.TRAINING], export_dir)
5     ...

```

### 3.5.8 在 C++ 中载入一个 SavedModel

C++ 版本的 SavedModel 载入器提供一个 API 从一个路径载入 SavedModel, 尽管允许 SessionOptions 和 RunOptions。你必须指定 tags 结合图被载入。载入的 SavedModel 版本被作为 SavedModelBundle 访问并且包含 MetaGraphDef 和 Session 被载入。

```

1 const string export_dir = ...
2 SavedModelBundle bundle;
3 ...

```

---

```
4 LoadSavedModel(session_options, run_options, export_dir, {kSavedModelTagTrain},
5 &bundle)
```

### 3.5.9 标准的常数

SavedModel 为各种使用情况提供灵活的构建和载入 TensorFlow 图。对多数使用情况，SavedModel 的 API 提供一些在 Python 和 C++ 中的常数，容易通过工具重用和共享。

### 3.5.10 标准的 MetaGraphDef tags

你也许设置 tag 唯一的确认一个保存在 SavedModel 的 MetaGraphDef。一些常用的 tags 在下面指定：

- [Python](#)
- [C++](#)

### 3.5.11 标准的 MetaGraphDef tags

一个[SignatureDef](#) 是一个 protocol buffer 定义图支持的计算签名。通常输入 keys 输出 keys，方法名称在：

- [Python](#)
- [C++](#)

### 3.5.12 结合 Estimator 使用 SavedModel

在训练一个 Estimator 模型后，你也许想从模型创建一个服务接受请求返回结果。你可以在你的机器上本地运行这样的一个服务或者规模化部署在云端。

为了准备训练的 Estimator 服务，你必须导出它为标准的 SavedModel 格式。

- 指定输出节点和对应的可以被服务的（分类，回归，预测）[APIs](#)
- 导出你的模型为 SavedModel 格式
- 从本地 server 和请求的预测服务模型

### 3.5.13 准备服务输入

在训练中，一个[input\\_fn\(\)](#) 接受数据通过模型准备他。在服务时间，类似的，一个 `serv-ing_input_receiver_fn()` 接受推理请求为模型准备他们。函数有下面的目的：

- 添加一个 Placeholder 到图上服务系统接获取推理请求
- 为了添加任何额外的操从输入格式转化将来模型希望的 Tensor

一个函数返回一个`tf.estimator.export.ServingInputReceiver`对象，打包 placeholder 和将来 Tensor 的结果在一起。一个典型的样例是推理请求以序列化的 tf.Example 到达，因此 `serving_input_receiver_fn()` 创建一个单独的字符串 placeholder 接收他们，这 `serving_input_receiver_fn()` 单后表示添加一个`tf.parse_examples`操作到图上代表解析 tf.Examples。当写像 `serving_input_receiver_fn()` 的时候，你必须传递一个解析的规范到`tf.parse_example`告诉解析器什么特征的名字被期望和如何映射他们为 Tensor。一个解析器说明接受一个来自特征名字到`tf.FixedLenFeature`和`tf.VarLenFeature`字典。注意这解析说明应该不包含任何标签和权重列，因为这将在服务时间可用，对比下解析规范在训练时间用在 `input_fn()`。

结合

```

1 feature_spec = { 'foo': tf.FixedLenFeature(...),
2                  'bar': tf.VarLenFeature(...) }
3
4 def serving_input_receiver_fn():
5     """An input receiver that expects a serialized tf.Example."""
6     serialized_tf_example = tf.placeholder(dtype=tf.string,
7                                            shape=[default_batch_size],
8                                            name='input_example_tensor')
9     receiver_tensors = { 'examples': serialized_tf_example }
10    features = tf.parse_example(serialized_tf_example, feature_spec)
11    return tf.estimator.export.ServingInputReceiver(features, receiver_tensors)
```

`tf.estimator.export.build_parsing_serving_input_receiver_fn`利用函数提供通常情况的输入接收器。

**注意:** 当在本地服务器上训练一个用于预测 API 模型是，解析步骤不需要应为模型将接受原始数据。

甚至如果你请求不解析或者其他其他的输入处理，如果服务系统将直接 feed 特征 Tensor，你必须始终提供一个 `serving_input_receiver_fn()` 为将来的 Tensor 创建 placeholder 传递他们。`tf.estimator.export.build_raw_serving_input_receiver_fn` 为这提供效用。

如果有些效用不满足你的需要，你能写自己的 `serving`、`—input`、`—receiver`、`—fn`。一种情况是如果你的训练 `input_fn()` 需要合并一些处理逻辑必须能在服务时间概括这些逻辑。为了减少训练服务偏斜，我们推荐压缩这样的处理在一个函数中，然后同 `input_fn()` 和 `serving_input_receiver_fn()` 中调用。

注意 `serving_input_receiver_fn()` 也决定签名的输入比例。当写一个的 `serving_input_receiver_fn()` 的时候，你必须告诉解析器签名希望和如何映射他们到你的模型希望的输入。通过对比，签名的输出比例由模型决定。

### 3.5.14 执行导出

为了导出你训练的 Estimator，调用`tf.estimator.Estimator.export_savedmodel` 结合导出的基础路径和 `serving_input_receiver_fn`。`estimator.export_savedmodel(export_dir_base, serving_input_receiver_fn)` 这个方法通过调用 `serving_input_receiver_fn()` 获取特征 Tensor，构建一个新的图。然后调用这个 Estimator 的 `model_fn()` 生成基于这些特征的模型图。它开始一个新的 Session， 默认恢复最近的 checkpoint。（如果需要的话，一个不同的 checkpoint 能被传递）最后它在哥顶的 `export_dir_base`（例如 `export_dir_base/<timestamp>`）创建一个时间戳导出目录，写一个 SavedModel 进去包含一个从这会话保存的单独的 MetaGraphDef。

**注意:** 垃圾收集旧的导入是你的责任。否则连续的导入将在 `export_dir_base` 下累积。

### 3.5.15 指定一个自定义模型的输出

当写一个自定义的 `model_fn`，你必须填充`tf.estimator.EstimatorSpec export_outputs` 返回的值的元素。这是一个字典 `{name:output}` 描述用于导出和服务的输出签名。

通常情况下做一个单独的预测，这字典包含一个元素，`name` 是不重要的。在 multi-header 模型，每个 head 被一个在这个字典中的输入表示。在这种情况下 `name` 是你的选择的字符串能在服务时间被用于请求一个指定的 head。每个 `output` 值必须是一个 `ExportOutput` 对象如:`tf.estimator.export.ClassificationOutput`,`tf.estimator.export.RegressionOutput` 或者`tf.estimator.export.PredictOutput`。

**注意:** 在 multi-header 情形下，一个 `SignatureDef` 将被 `model_fn` 返回的 `export_outputs` 字典的每个元素生成。这些 `SignatureDefs` 当提供给对象的 `ExportOutput` 输入是在输入时不同。输入总是有 `serving_input_receiver_fn` 提供。一个推理请求也许通过 `name` 指定 head。一个 head 必须使用`signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY` 指示当推理请求没有指定的时候那个 `SignatureDef` 将被服务。

### 3.5.16 服务导人的本地模型

对于本地不熟，你可以用TensorFlow Serving服务你的吗 O 型，一个开源的项目载入 SavedModel 和利用它作为gRPC服务。

首先安装 TensorFlow Serving

然后构建运行本地的模型服务器，替代 `$export_dir_base` 结合你上面导出的 SavedModel:

```
1 bazel build //tensorflow_serving/model_servers:tensorflow_model_server
```

```
2 bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server --port=9000
--model_base_path=$export_dir_base
```

现在你有一个服务器通过 gRPC 的 9000 端口监听推理。

### 3.5.17 从一个本地服务器请求预测

服务器根据[PredictionService](#)对应 gRPC API 服务定义响应请求。(嵌套的 protocol buffer 被定义在[neighboring files](#)). 从 API 服务定义中, gRPC 框架在多用语言中提供远程访问 API 生成客户库文件。在一个用 Bazel 构建的项目中, 这枯萎自动构建并且提供这样的依赖 (例如使用 Python)

```
1 deps = [
2     "//tensorflow_serving/apis:classification_proto_py_pb2",
3     "//tensorflow_serving/apis:regression_proto_py_pb2",
4     "//tensorflow_serving/apis:predict_proto_py_pb2",
5     "//tensorflow_serving/apis:prediction_service_proto_py_pb2"
6 ]
```

Python 客户端代码可以然后导入库:

```
1 from tensorflow_serving.apis import classification_pb2
2 from tensorflow_serving.apis import regression_pb2
3 from tensorflow_serving.apis import predict_pb2
4 from tensorflow_serving.apis import prediction_service_pb2
```

**注意:**prediction\_service\_pb2 定义服务为一个整体因此总是被要求。然而一个典型的客户将需要仅仅 classification\_pb2, regression\_pb2 和 prediction\_pb2, 依赖请求的类型。

发送一个 gRPC 请求然后通过集合 protocol buffer 包含请求数据然后传递数据到服务桩 (stub)。注意现在请求 protocol buffer 被创建空的然后通过[generated protocol buffer API](#)填充。

```
1 from grpc.beta import implementations
2
3 channel = implementations.insecure_channel(host, int(port))
4 stub = prediction_service_pb2.beta_create_PredictionService_stub(channel)
5
6 request = classification_pb2.ClassificationRequest()
7 example = request.input.example_list.examples.add()
8 example.features.feature['x'].float_list.value.extend(image[0].astype(float))
9
10 result = stub.Classify(request, 10.0) # 10 secs timeout
```

在这个例子中的返回结果是一个 ClassificationResponse protocol buffer。这是一个框架例子，请查看 TensorFlow serving 文档和[例子](#)查看详情。

**注意:**ClassificationRequest 和 RegressionRequest 包含一个 tensorflow.serving.Input protocol buffer, 它包含已了 tensorflow.Example protocol buffers 列表。PredictRequest, 相比之下, 通过 TensorProto 包含一个从特征名称到值的映射。对应, 当用 Classify 和 Regress SPIs, TensorFlow servingfeed 序列化的 tf.Examples 到图上, 因此你的 serving\_input\_receiver\_fn 应该包含一个 tf.parse\_example() 操作。当用一般的 Predict API 的时候, 然而 TensorFlow Serving feed 原始的特征数据到图上, 因此通过 serving\_input\_receiver\_fn() 应该被使用。

#### 3.5.18 CLI 查看和执行 SavedModel

你可以使用 SavedModel 命令行接口 (CLI) 查看和执行一个 SavedModel。例如, 你可以使用 CLI 查看模型的 SignatureDef。CLI 是你能快速确认模型的输入 tensor 的形状和数据类型。然而, 如果你想测试你的模型, 你可以通过在 sample 输入中传递 sample 输入用 CLI 做一个明确的选择 (例如, Python 表达式) 然后获取输出。

#### 3.5.19 安装 SavedModel CLI

一般来说, 你可以用两种方式安装 TensorFlow:

- 安装一个预先编译好的 TensorFlow 二进制
- 通过从源代码构建 TensorFlow

如果你通过预定义的 TensorFlow 二进制安装 TensorFlow, 然后 SavedModel CLI 已经安装在你的系统上, 路径为 bin

saved\_model\_cli。如果你通过 TensorFlow 源代码安装, 你必须寻星下面的命令构建 saved\_model\_cli:

```
1 bazel build tensorflow/python/tools:SavedModel_cli
```

#### 3.5.20 命令概览

SavedModel CLI 在 SavedModel 中的一个 MetaGraphDef 中支持下面的命令:

- show 显示在 SavedModel 中的 MetaGraphDef 一个计算
- run 在 MetaGraphDef 上的一个计算

### 3.5.21 显示

一个 SavedModel 包含一个或者更多的 MetaGraphDef，确保他们的 tag-sets。为了服务模型，你也许想知道每个模型的 SignatureDef 种类，什么是他们的输入和输出。show 目录让你用层级结构顺序检查 SavedModel 的内容。这里是语法：

```
1 usage: saved_model_cli show [–h] —dir DIR [—all]
2 [—tag_set TAG_SET] [—signature_def SIGNATURE_DEF_KEY]
```

例如，下面的命令显示所有在 SavedModel 中可用的 MetaGraphDef tag-sets：

```
1 saved_model_cli show —dir /tmp/saved_model_dir
2 The given SavedModel contains the following tag-sets:
3 serve
4 serve, gpu
```

下面的命令显示所有在 MetaGraphDef 中可用的 SignatureDef keys：

```
1 saved_model_cli show —dir /tmp/saved_model_dir —tag_set serve
2 The given SavedModel ‘MetaGraphDef’ contains ‘SignatureDefs’ with the
3 following keys:
4 SignatureDef key: “classify_x2_to_y3”
5 SignatureDef key: “classify_x_to_y”
6 SignatureDef key: “regress_x2_to_y3”
7 SignatureDef key: “regress_x_to_y”
8 SignatureDef key: “regress_x_to_y2”
9 SignatureDef key: “serving_default”
```

如果一个 MetaGraphDef 有多个 tags 在 tag-set 中，你必须指定所有的 tags，每个 tag 通过逗号隔开。例如：saved\_model\_cli show --dir /tmp/saved\_model\_dir --tag\_set serve,gpu 为了为一个 SignatureDef 指定所有的输入输出 TensorInfo，传递在 SignatureDef key 到 signature\_def 选项。当你想知道 tensor key value，数据类型和稍后执行计算图的输入 tensor 的形状时很有用，例如：

```
1 saved_model_cli show —dir \
2 /tmp/saved_model_dir —tag_set serve —signature_def serving_default
3 The given SavedModel SignatureDef contains the following input(s):
4 inputs[‘x’] tensor_info:
5   dtype: DT_FLOAT
6   shape: (-1, 1)
7   name: x:0
8 The given SavedModel SignatureDef contains the following output(s):
9 outputs[‘y’] tensor_info:
10  dtype: DT_FLOAT
11  shape: (-1, 1)
12  name: y:0
```

```
13 Method name is: tensorflow/serving/predict
```

为了显示在 SavedModel 中所有可用信息，使用`--all` 选项。例如：

```
1 saved_model_cli show --dir /tmp/saved_model_dir --all
2 MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
3
4 signature_def['classify_x2_to_y3']:
5 The given SavedModel SignatureDef contains the following input(s):
6 inputs['inputs'] tensor_info:
7   dtype: DT_FLOAT
8   shape: (-1, 1)
9   name: x2:0
10 The given SavedModel SignatureDef contains the following output(s):
11 outputs['scores'] tensor_info:
12   dtype: DT_FLOAT
13   shape: (-1, 1)
14   name: y3:0
15 Method name is: tensorflow/serving/classify
16
17 ...
18
19 signature_def['serving_default']:
20 The given SavedModel SignatureDef contains the following input(s):
21 inputs['x'] tensor_info:
22   dtype: DT_FLOAT
23   shape: (-1, 1)
24   name: x:0
25 The given SavedModel SignatureDef contains the following output(s):
26 outputs['y'] tensor_info:
27   dtype: DT_FLOAT
28   shape: (-1, 1)
29   name: y:0
30 Method name is: tensorflow/serving/predict
```

### 3.5.22 运行命令

调用 `run` 命令运行一个计算图，产地输入然后显示输出，这里是语法：

```
1 usage: saved_model_cli run [-h] --dir DIR --tag_set TAG_SET --signature_def
2                           SIGNATURE_DEF_KEY [--inputs INPUTS]
3                           [--input_expressions INPUT_EXPRESSIONS] [--outdir OUTDIR]
4                           [--overwrite] [--tf_debug]
```

`run` 命令提供下面两种方法传递给模型：

- `--inputs` 选项是你能在文件中传递 numpy 数据
- `--input_expressions` 选项使你能传递 Python 表达式

#### `--inputs`

在文件中传递输入数据，指定`--inputs` 选项，接受下面的常规格式: `--inputs <INPUTS>` 这里 INPUTS 可以是下面的格式:

- `<input_key>=<filename>`
- `<input_key>=<filename>[<varia_name>]`

你可以传递多个 INPUTS，如果你传递多个输入，使用分号分割 INPUTS。`saved_model_cli` 使用 `numpy.load` 载入文件名，文件名必须是下面格式:

- `.npz`
- `.npy`
- `pickle` 格式

一个.npy 文件总是包含一个 numpy 数组，一次当从.npy 文件载入时，内容将被直接由指定的输入 tensor 指定。如果你用.npy 文件指定 `variable_name`, `variable_name` 将被忽略警告问题。当如.npz(zip) 文件载入时，你也能选择指定一个 `variable_name` 确认对输入 tensor key 的变量和 zip 文件载入。如果你不指定 `variable_name`, SavedModel CLI 将检查仅仅一个包含 zip 文件的文件为指定的输入 tensor key 载入它。当从 pickle 文件载入是，如果没有 `variable_name` 被在方括号指定，在 pickle 中无论什么江北传递到指定的输入 tensor key。否则，SavedModel CLI 将假设一个目录存储 pickle 文件和 `variable_name` 将被使用的值。

#### `--inputs_expressions`

为了通过 Python 表达式传递输入，指定`--input_expr` 选项。对于当你没有数据文件的时候是很有用的，但是想结合简单的数据粗略的检查模型匹配模型的 SignatureDef 的形状，例如: `'input_key=[[1], [2], [3]]'` 另外 Python 表达式，你也许传递 numpy 函数。例如: `input_key=np.ones((32, 32, 3))` (注意 numpy 模块对于你导入为 np 已经可用)

### 3.5.23 保存输出

默认，SavedModel CLI 写输出到标准输出流。如果一个目录传递给`--outdir` 选项，输出将被保存在给定目录下输出 tensor keys 重命名为 npy 文件。

### 3.5.24 TensorFlow Debugger(tfdbg) Integration

如果`--tf_debug` 选项被设置，SavedModel CLI 将使用 TensorFlow Debugger(tfdbg) 查看运行 SavedModel 时中间 Tensor 和运行图或者子图。

### 3.5.25 run 的完整例子

给定：

- 你的模型简单的添加  $x_1$  和  $x_2$  得到输出  $y$
- 所有在模型中的 tensor 形状为 (-1,1)
- 你有两个 npy 文件：
  1. /tmp/data\_data1.npy 包含一个 numpy 数组 [[1],[2],[3]]
  2. /tmp/my\_data2.npy 包含另一个 numpy 数组 [[0.5],[0.5],[0.5]]

下面的目录运行两个 npy 文件得到输出  $y$ :

```

1 saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
2 --signature_def x1_x2_to_y --inputs x1=/tmp/my_data1.npy;x2=/tmp/my_data2.npy \
3 --outdir /tmp/out
4 Result for output key y:
5 [[ 1.5]
6 [ 2.5]
7 [ 3.5]]

```

稍微改动先前的例子，这里用两个.npy 文件，你现在有一个.npz 文件和 pickle 文件。进一步，你想覆盖任何存在的输出文件，这里是命令：

```

1 saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
2 --signature_def x1_x2_to_y \
3 --inputs x1=/tmp/my_data1.npz[x];x2=/tmp/my_data2.pkl --outdir /tmp/out \
4 --overwrite
5 Result for output key y:
6 [[ 1.5]
7 [ 2.5]
8 [ 3.5]]

```

你也许指定 Python 表达式而不是一个输入文件。例如，下面命令用 Python 表达式代替输入  $x_2$ :

```

1 saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
2 --signature_def x1_x2_to_y --inputs x1=/tmp/my_data1.npz[x] \
3 --input_exprs 'x2=np.ones((3,1))'
4 Result for output key y:
5 [[ 2]
6 [ 3]
7 [ 4]]

```

为了在 TensorFlow Debugger 上运行模型，使用下面的命令：

```
1 saved_model_cli run --dir /tmp/saved_model_dir --tag_set serve \
2 --signature_def serving_default --inputs x=/tmp/data.npz[x] --tf_debug
```

### 3.5.26 SavedModel 目录的结构

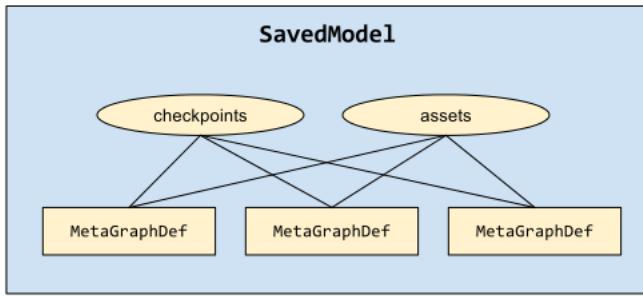
当你茂村一个模型为 SavedModel 格式的时候，TensorFlow 创建一个下面子目录和文件组的 SavedModel 目录：

```
1 assets/
2 assets.extra/
3 variables/
4     variables.data-?????-of-?????
5     variables.index
6 saved_model.pb|saved_model.pbtxt
```

这里：

- assert 是一个包含扩展文件的子文件夹，像词汇。Assets 被复制到 SavedModel 位置当载入一个指定的 MetaGraphDef 是被读取
- assets.extra 是一个子文件夹，这里高级库和用户能添加他们自己的 assets 和模型共存，但是不通过图载入。这子文件夹不通过 SavedModel 库管理
- variables 是一个包含来自 tf.train.Saver 的子文件夹
- saved\_model.pb 或者 saved\_model.pbtxt 是 SavedModel protocol 协议。它包含图定义作为 metaGraphDef protocol buffer

一个单个的 SavedModel 可以表示多张图。在这种情况下，所有的在 SavedModel 中的图共享一个单个的 checkpoint(变量) 和 assets。例如下面的图显示了一个 SavedModel 包含三个 MetaGraphDef，所有的 MetaGraphDef 共享 checkpoint 和 assets：



每张图介个指定的 tags，使得在载入和恢复操作时确认可行。

## 3.6 导入数据

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，随机扰动每张图片，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，转换它们为查找表 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 和图像数据的 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 图像构

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。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。

### 3.6.1 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。

### 3.6.2 数据结构

一个数据集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件的 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件的类型和形状。这些参数的嵌套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_shapes)#{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,)})

```

给每个元素的组成命名是很方便的，例如它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): ...)
```

### 3.6.3 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 当前支持一下迭代器：

- 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，但是它们不支持参数化。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
```

initilizable 迭代器要求你使用前明确的运行 iterator.initializer 操作。为此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 #Base environ pass
2 max_value = tf.placeholder(tf.int64, shape=[])
3 dataset = tf.contrib.data.Dataset.range(max_value)
4 iterator = dataset.make_initializable_iterator()
5 next_element = iterator.get_next()
6 sess = tf.Session()
7 # Initialize an iterator over a dataset with 10 elements.
8 sess.run(iterator.initializer, feed_dict={max_value: 10})
9 for i in range(10):
10     value = sess.run(next_element)
11     assert i == value
12 # Initialize the same iterator over a dataset with 100 elements.
13 sess.run(iterator.initializer, feed_dict={max_value: 100})
14 for i in range(100):
15     value = sess.run(next_element)
16     assert i == value

```

一个 reinitializable 迭代器可以通过多个不同的 Dataset 对象初始化。例如你也许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 将用于相同结构（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 时选择什么通过 feed\_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集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 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})
```

### 3.6.4 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它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` 对象，传递表达式的结果给 `tf.Session.run` 得到表达式的结果和下一个迭代器。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]),
3                                                       tf.random_uniform([4, 100])))
4 dataset3 = tf.contrib.data.Dataset.zip((dataset1, dataset2))
5 iterator = dataset3.make_initializable_iterator()
6 sess.run(iterator.initializer)
7 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。

### 3.6.5 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()` 转换它们为 `tf.Tensor` 对象创建一个 `Dataset`。

```
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))
```

注意上面的代码将在你的 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,
10                                                        labels_placeholder))
11    # [Other transformations on 'dataset' ...]
12    dataset = ...
13    iterator = dataset.make_initializable_iterator()
14    sess.run(iterator.initializer, feed_dict={features_placeholder: features,
15                                              labels_placeholder: labels})
```

### 3.6.6 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。像 `TFRecordDataset`，`TextLineDataset` 接收 `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行有一些注释。这些行可以用 Dataset.skip() 移除和 Dataset.filter() 转换。为了在分割的文件应用这些转换，我们用 Dataset.flat\_map() 为每个文件创建一个迭代的 Dataset

```

1 #TensorFlow 1.3 GPU ubuntu14.0.4 调试通过
2 import tensorflow as tf
3 filename = ['./t1.txt', './t2.txt']
4 filename = tf.constant(filename)
5 dataset = tf.contrib.data.Dataset.from_tensor_slices(filename)
6 dataset = dataset.flat_map(lambda filename:(
7     tf.contrib.data.TextLineDataset(filename).skip(1).filter(lambda line:tf.
8         not_equal(tf.substr(line,0,1), '#'))))
9 iterator = dataset.make_one_shot_iterator()
10 next_element = iterator.get_next()
11 sess = tf.Session()
12 for i in range(10):
13     print(sess.run(next_element))
```

### 3.6.7 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 接收 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。

### 3.6.8 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 pipline 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)
```

### 3.6.9 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，因此它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     #这里需要根据读取图片的数据编码格式，条用不同的解码函数
6     image_decoded = tf.image.decode_png(image_string)
7     image_resized = tf.image.resize_images(image_decoded , [28 , 28])
8     return image_resized , label
9
10 # A vector of filenames .
11 filenames = tf.constant(["/var/data/image1.jpg" , "/var/data/image2.jpg" , ...])
12
13 # 'labels[i]' is the label for the image in filenames[i]
14 labels = tf.constant([0 , 37, ...])
15 dataset = tf.contrib.data.Dataset.from_tensor_slices((filenames , labels))
16 dataset = dataset.map(_parse_function)
```

### 3.6.10 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)

```

### 3.6.11 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])

```

### 3.6.12 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() 使你通过指定一个或者更多维（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,
9   3]]

```

```

9 print(sess.run(next_element)) # ==> [[4, 4, 4, 4, 0, 0, 0],
10          #      [5, 5, 5, 5, 5, 0, 0],
11          #      [6, 6, 6, 6, 6, 6, 0],
12          #      [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，它也许有变化的长度或者固定的长度，它可以覆盖 padding 的值（默认为 0）。

### 3.6.13 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() 变换连接它的参数每个 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.]

```

### 3.6.14 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 (查看完整例子):

```

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()

```

### 3.6.15 用高级 APIs

tf.train.MonitoredTrainingSession API 简化了一些在分布式方面运行方面的设置。MonitoredTrainingSession 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

```

## 3.7 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 sess = tf.Session()
2 q = tf.FIFOQueue(3, "float")
3 init = q.enqueue_many(([0., 0., 0.],))
4 x = q.dequeue()
5 y = x+1
6 q_inc = q.enqueue([y])
7 init.run(session=sess)

```

```

8 q_inc.run(session=sess)
9 q_inc.run(session=sess)
10 q_inc.run(session=sess)
11 q_inc.run(session=sess)

```

Enqueue, EnqueueMany 和 Dequeue 是一个特别的节点。它们是指向队列真实值的指针，允许它们改变状态。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

### 3.7.1 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 pipeline 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。

### 3.7.2 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它们停止, `QueueRunner` 类被用于创建一个线程协作同一队列中的入队 tensor。

### 3.7.3 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。它们不是必须和上面的例子一样。`coordinator` 也支持捕获和报告异常, 查看[tf.train.Coordinator](#)文档查看更多信息。

#### 3.7.4 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 enqueue 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)

```

### 3.7.5 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。它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)

```

## 3.8 embeddings

一个 embedding 是一个从离散对象，像字映射为一个真实值的向量，例如一个英文字符的 300 维的 embedding 可能包括：

```

1 blue:  (0.01359, 0.00075997, 0.24608, ..., -0.2524, 1.0048, 0.06259)
2 blues: (0.01396, 0.11887, -0.48963, ..., 0.033483, -0.10007, 0.1158)
3 orange: (-0.24776, -0.12359, 0.20986, ..., 0.079717, 0.23865, -0.014213)
4 oranges: (-0.35609, 0.21854, 0.080944, ..., -0.35413, 0.38511, -0.070976)

```

embedding 让你能在离散的数据上应用机器学习，分类器，更常用的神经网络都被设计为一个高密度的连续向量（所有的值都用来定义一个对象）如果离散对象被编码为一个离散的原子，有独一无二的 id 号，它们阻止学习和泛化，一种考虑 embeddings 的方法是转化非向量的对象为有用的机器学习输入。, embedding 作为机器学习的输出也是有用的，因为 embedding 映射对象为向量，在向量空间中的应用是类似的。一个通常的用法是找到一个最接近的邻居。用和相面相同的 word embedding，例如对于每个字和相关角度这里有三个接近的邻居：

```

1 blue: (red, 47.6°), (yellow, 51.9°), (purple, 52.4°)
2 blues: (jazz, 53.3°), (folk, 59.1°), (bluegrass, 60.6°)
3 orange: (yellow, 53.5°), (colored, 58.0°), (bright, 59.9°)
4 oranges: (apples, 45.3°), (lemons, 48.3°), (mangoes, 50.4°)

```

这将告诉程序苹果和橙子类似度 ( $45.3^\circ$ ) 比柠檬和橙子 ( $48.3^\circ$ ) 更类似。

### 3.8.1 训练一个 embedding

为了在 TensorFlow 中训练 word embedding，首先你需要分隔文字成单词赋值给词汇表中的每一个单词一个整数。假设我们已经做了这步，word\_ids 是一个整数向量。例如句子 “I have a cat.” 可能被分割成 [“I”, “have”, “a”, “cat”, “.”] 然后相关的 word\_ids tensor 形状为 [5] 由 5 个整数组成。为了得到这些单词 ids embed，我们需要用 tf.gather 函数按照下面创建 embedding 变量。

```

1 word_embeddings = tf.get_variable("word_embeddings",
2     [vocabulary_size, embedding_size])
3 embedded_word_ids = tf.gather(word_embeddings, word_ids)

```

在这之后 tensor embedded\_word\_ids 在我们的例子中将有形状 [5,embedding\_size] 同时包含 5 个单词的 embeddings(dense vector)。变量 word\_embeddings 将被学习, 在训练结束后它将包含所有在词汇表中的 embeddings。embeddings 可能被多种方式训练, 依靠数据变量。例如可以用循环神经网络从语料中的句子的前一个单词预测下一个单词或者用两个网络做多种语言的翻译。这个方法在**字词的向量表示**中有描述, 但是上面的所有情况下有一个像上面的 embedding 变量和用 tf.gather 的 embedded word。

### 3.8.2 可视化 Embeddings

TensorBoard 有一个内建的可视化器, 称为 EmbeddingProjector, 用于交互式的 embedding 可视化。embedding projector 将从你的 checkpoint 文件读取 embeddings 用 PCA 映射它们到 3 维空间。对于 PCA 的可视化查看[这里](#), 另一个有用的映射是 t-SNE。如果你正在 embedding 上工作, 你可能像添加 labels/images 到数据点。你可以通过生成一个包含每个点的 labels 和用我们的 Python API 映射器的配置[metadata file](#), 或者在和你的 checkpoint 文件相同多个目录手动构造和保存一个 projector\_config.pbtxt。

### 3.8.3 创建

1. 设置一个 2 维 tensor 保存你的 embedding。

```
1     embedding_var = tf.get_variable(...)
```

2. 定期保存你模型的变量到 LOG\_DIR 目录中

```
1     saver = tf.train.Saver()
2     saver.save(session, os.path.join(LOG_DIR, "model.ckpt"), step)
3
```

3. 结合 meta data 和你的 embedding(可选)

如果你有任何的 metadata(labels,images) 结合你的 embedding, 你可以调用 TensorBoard 通过指定存储在 LOG\_DIR 中的一个 projector\_config.pbtxt 或者用你自己的 PythonAPI。例如下面的 projector\_config.pbtxt 结合 word\_embedding tensor 和存储在 \$LOG\_DIR/metadata.tsv 的 metadata 文件。

```
1 embeddings {
2     tensor_name: 'word_embedding'
3     metadata_path: '$LOG_DIR/metadata.tsv'
4 }
```

相同的配置可以通过下面的代码段一程序产生:

```

1 from tensorflow.contrib.tensorboard.plugins import projector
2 # Create randomly initialized embedding weights which will be trained.
3 vocabulary_size = 10000
4 embedding_size = 200
5 embedding_var = tf.get_variable('word_embedding', [vocabulary_size,
6                                         embedding_size])
# Format: tensorflow/tensorboard/plugins/projector/projector_config.proto
7 config = projector.ProjectorConfig()
8 # You can add multiple embeddings. Here we add only one.
9 embedding = config.embeddings.add()
10 embedding.tensor_name = embedding_var.name
11 # Link this tensor to its metadata file (e.g. labels).
12 embedding.metadata_path = os.path.join(LOG_DIR, 'metadata.tsv')
13 # Use the same LOG_DIR where you stored your checkpoint.
14 summary_writer = tf.summary.FileWriter(LOG_DIR)
15 # The next line writes a projector_config.pbtxt in the LOG_DIR. TensorBoard will
16 # read this file during startup.
17 projector.visualize_embeddings(summary_writer, config)

```

在你运行你的模型和训练你的 embedding 后，运行 TensorBoard 和指定 job 的 LOG\_DIR

```
1 tensorboard --logdir=LOG_DIR
```

点击顶端面板的 Embedding 选择合适的运行。

#### 3.8.4 metdadata

通常 embeddings 有 metedata 结合。metadata 应该被存储在和模型的 checkpoint 分隔的文件中，因此 metadata 不是可训练的模型的参数。这是应该为[TSV file](#)（显示红色标签），第一行十粗体显示的表头和包含 metadata 值得子序列行。

```

1 Word\tFrequency
2 Airplane\t345
3 Car\t241
4 ...

```

有不明确的和主要数据文件的关键共享，而不是在 madatada 文件中的顺序和 embedding tensor 里面的顺序相匹配，换句话说第一行是头信息，metadata 文件中 (i+1) 行和存储在 checkpoint 文件中的 embedding tensor 的第 i 行相关。

如果 TSV metadata 文件仅仅有一列，我们不需要第一行假设每一行是 embedding 的 label，我们包括这个例外，因为它匹配通常用的”vocab file” 格式。

### 3.8.5 图像

如果你有图像和你的 embedding 结合，你将需要生成一个每个数据点缩略图组成的图像。它被称为[sprite image](#)。sprite 应该有和存储进行优先顺序的缩略图相同的行和列：第一个数据点放在是左上角最后一个数据放在右下角。

0	1	2
3	4	5
6	7	

注意上面例子的最后一个元素不是必须填，对于具体的 sprite 例子查看 10000 张手写体数据[sprite image](#)

注意我们当前支持  $8192px \times 8192px$

在够早了 sprite 后你需要告诉 Embedding Projector 到哪里寻找它：

```
1 embedding.sprite.image_path = PATH_TO_SPRITE_IMAGE
2 # Specify the width and height of a single thumbnail.
3 embedding.sprite.single_image_dim.extend([w, h])
```

### 3.8.6 交互

Embedding Projector 有三个面板：

1. 左上角的 Data 面板，你也已选择运行 embedding tensor 和数据列标上颜色和 label points。
2. 左下角的 Projections 面板选择 projection 的类型 (PCA,tSNE)
3. 右边的 Inspector 面板，这里你能搜索类似的点查看最近邻居的列表

### 3.8.7 Projections

Embedding Projector 有三种方法减少数据集的维度：两个线性一个非线性。每二个方法可以被用来创建一个二维或者三维视图。

PCA：一个直接的降维技术，Embedding projectorj 计算 10 个主成分。菜单让你映射这些成为任何二维或者三维。PCA 是一个线性 projection，检查全局结构时很有效。

t-SNE：一个流行的非线性降维 Embedding Projector 提供二维和三维视图。客户端执行算法的每一步都被更新动画。因为 t-SNE 经常保留一些本地结构，对于本利邻居和发现集群是很有用的。尽管对于高维数据可视化很有用,t-SNE 画图可能会有一些神秘或者难以理解，查看[great artical](#)查看 t-SNE 如何高效使用。

Custom: 你也可以构造基于文本搜索查找在空间中有用的方向的特别的线性 projections。为了定义一个 projection 轴，输入两个搜索字符串或者正则表达式。程序计算标签和搜索结果配的标签的中心，在不同的向量中心作为一个 projection 轴。

### 3.8.8 导航

为了探索数据集，用 2 维或者 3 维视图查看，缩放，旋转和用自然的手势点击和拖动面板。点击一个点引起右边面板显示一个确定的最相邻的文本列表。最相邻的点本身在 projection 被强调。

缩放进入集群给定一些信息，但是有时候更有用的是限制自己点的视角和在这些点上执行 projection.，你可以用多种方式选择点：

1. 在点击点后，相邻的点也被选中。
2. 在搜索后匹配的查询被选中。
3. Embedding 选择，点击一个点拖动定义一个选择范围。

在选择一个数据集点后，你可以用右边的 inspector 面板隔离这些点用 isolate Pointes 进一步分析，

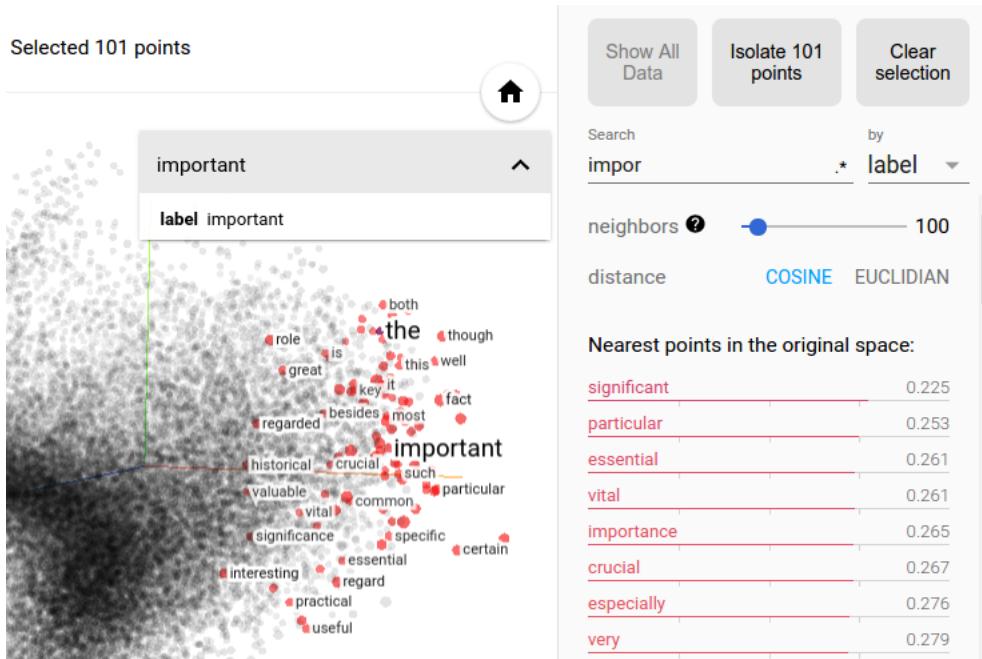
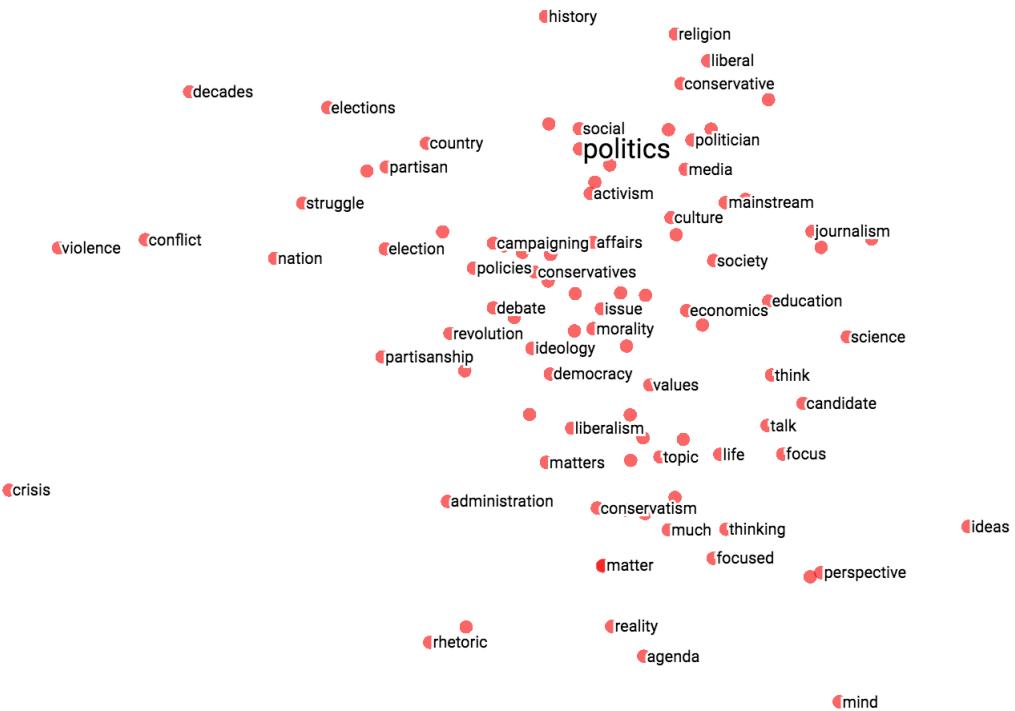


图 3.1: important 最相邻的 embedding 数据集

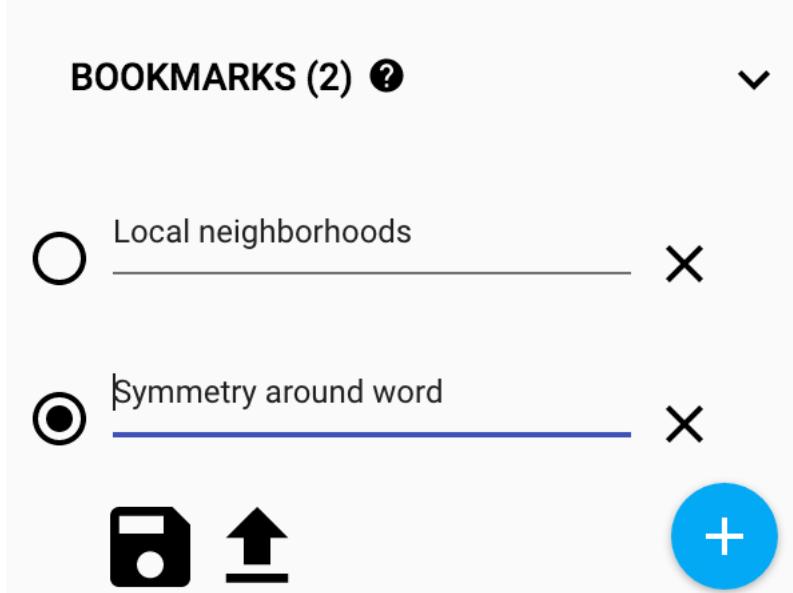
custom projection 结合过滤会很有用，下面我们过滤和”politics”100 个最相邻的点,project



它们在 x 轴上从好到坏, 向量表示, y 轴水机。你可以看右边的我们有”ideas”, ”science”, ”perspective”, ”journalism” 左边有”crisis”, ”violence” 和”conflict”。

### 3.8.9 合作的特性

为了分享你的发现你可以用右上角的 bookmark 面板保存当前状态 (包括计算任何 projection 的坐标) 为一个小文件。Projector 可能被指定到一个或者更多文件, 产生下面的面板, 其它人可以通过标签序列查看。



### 3.8.10 简单的问答

embedding 是一个动作还是一个事物？两者都是，人们谈论向量空间的 embedding word 形成 embedding(事物)。通常两个是一个从离散对象到向量映射 embedding 概念，创建应用映射是一个行为，但是映射是一个事物。

是高维还是低维 embedding？300 维的向量字词空间，例如当和上百万的字词空间相比是低维。但是数学上它是高维，显示了大量和人们直觉上的 2 维或者三维空间不一致的特性。

embedding 和 embedding layer 是一样的吗？不是，一个 embedding layer 是神经网络的一部分，不是一个 embedding 是一个更常用的概念。

# Chapter 4

## 使用向导

### 4.1 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]]
```

### 4.1.1 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`)

### 4.1.2 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('/gpu:2'):
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，你将得到一个 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()
```

## 4.2 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。

### 4.2.1 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。接下来的章节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
```

## 4.2. 如何利用 INCEPTION 的最后一层重新训练新的分类

```
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

```
4 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

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

训练器可以是这样：

```
1 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。

### 4.2.2 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它们的瓶颈，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，它的速度达到缓存起的瓶颈因此不能被重复计算。默认它们存储在/tmp/bottleneck 陌路，如果你仍然会脚本它们将被重用，因此你不是必须再次等待这部分。

### 4.2.3 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

## 4.2. 如何利用 INCEPTION 的最后一层重新训练新的分类

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。

### 4.2.4 用 TensorBoard 可视化

包含 TensorBoard 总结的脚本吗很容易理解，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：

```
1 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。

### 4.2.5 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, text 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：

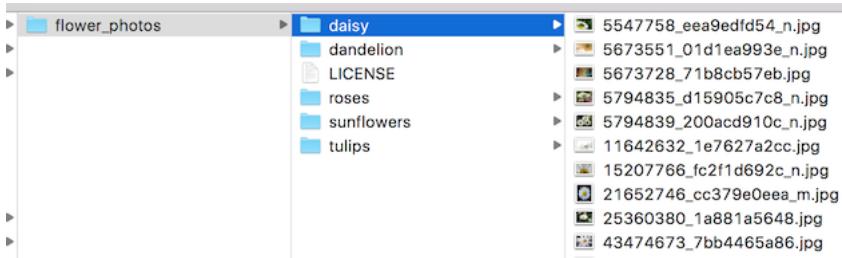
```
1 bazel build tensorflow/examples/image\_retraining:label\_image && \
2 bazel-bin/tensorflow/examples/image_retraining/label_image \
3 --graph=/tmp/output_graph.pb --labels=/tmp/output\_labels.txt \
4 --image=$Home/flower_photos/daisy/21625746_cc379eea_m.jpg
```

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](#)

### 4.2.6 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，你可以教它认识其它你想让它认识的东西。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。



### 4.2.7 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它看到的，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。

### 4.2.8 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，它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。

### 4.2.9 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。

### 4.2.10 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, 但是它可能对精度有帮助, 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。

### 4.2.11 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。最大的数据及是训练集, 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它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, 因此模型不能记住它们, 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它们。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, 因为它们仅仅反映在训练集中更多的问题。

### 4.2.12 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它的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  $128 \times 128$  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，它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。

## 4.3 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  $28 \times 28$

### 4.3.1 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
```

```

9 tf.logging.set_verbosity(tf.logging.INFO)
10
11 # Our application logic will be added here
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](#)

### 4.3.2 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。

### 4.3.3 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  $5 \times 5$  卷积核 (提取  $5 \times 5$  像素的区域)，用 relu 激活函数。
2. 池化层 1: 执行最大池化  $2 \times 2$  stride=2(指定的池化区域不重叠)
3. 卷积层 2: 应用 64 个  $5 \times 5$  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  $2 \times 2$ ) stride=2。
5. Dense1:1024 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。aughories 你已经体验过 CNN 设 TensorFlow Estimators，你可以跳到[Training and Evaluating the CNN MNIST Classifier](#)

#### 4.3.4 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  $28 \times 28$  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]`。

#### 4.3.5 第一层卷积层

在我们的卷积层我想用 32 个  $5 \times 5$  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_layers,
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 参数指定卷积核的个数, 这里卷积核为 32 个。kernel\_size 指定卷积核的维度为 [width,height] (这里 [5,5]) padding 参数指定两个值:valid(默认), 和 same。指定输出 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, 宽度和高度为 28(没有 padding $5 \times 5$  卷积  $28 \times 28$  将生成  $24 \times 2$  的 4 维 tensor, 在  $28 \times 28$  用  $5 \times 5$  提取出  $24 \times 24$  个位置)。activation 参数指定应用到输出的激活函数, 这里我们只用 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。

#### 4.3.6 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  $2 \times 2$ , stride 为 2。

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

再次, inputs 指定输入 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 参数指定 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, 你可以指定一个元组或者列表。

我们的输出特征是偶然和 max\_pooling2d(pool1, 形状为 [batch\_size,14,14,32]) 相乘:  $2 \times 2$  减少宽度和高度到 50%。

#### 4.3.7 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  $5 \times 5$  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）

#### 4.3.8 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 参数指定输入 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 参数指定 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 行值指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]

#### 4.3.9 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]。

#### 4.3.10 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 参数指定需要提取最大值的 tensor, axis 参数指定输入 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)
```

#### 4.3.11 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)
```

### 4.3.12 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)
```

### 4.3.13 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)
```

## 4.4 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。

### 4.4.1 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。

#### 4.4.2 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 参数指定模型数据的保存目录为 /tmp/mnist\_convnet\_model。

#### 4.4.3 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, 指定训练的时候每 50 步采集概率。

#### 4.4.4 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 logging\_hook 去 hooks 参数，以至于它将在训练期间被触发。

#### 4.4.5 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。

#### 4.4.6 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,
    loss = 0.0902271
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。

## 4.5 卷积神经网络

### 4.5.1 概览

CIFAR-10 分类是一个机器学习常见的测试标准。即分类  $32 \times 32$  的 RGB 图像为 10 类: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck。更多细节查看 [CIFAR-10 Page](#)和[Tech Report](#)。

### 4.5.2 目标

这个导航的目标是构建一个相当小的 CNN 识别图像。在这个导航中:

1. 着重于建立一个规范的网络组织结构，训练并进行评估
2. 为建立更大规模更加复杂的模型提供一个范例

选择 CIFAR-10 是因为它的复杂程度足以用来检验 TensorFlow 中的大部分功能，并可将其扩展为更大的模型。与此同时由于模型较小所以训练速度很快，比较适合用来测试新的想法，检验新的技术

### 4.5.3 本教程重点

CIFAR-10 教程演示了在 TensorFlow 上构建更大更复杂模型的几个种重要内容:

- 相关核心数学对象，如[卷积](#)、[修正线性激活](#)、[最大池化](#)以及[局部响应归一化](#)；

文件	作用
cifar10_input.py	读取本地 CIFAR-10 的二进制文件格式的内容。
cifar10.py	建立 CIFAR-10 的模型。
cifar10_train.py	在 CPU 或 GPU 上训练 CIFAR-10 的模型。
cifar10_multi_gpu_train.py	在多 GPU 上训练 CIFAR-10 的模型。
cifar10_eval.py	评估 CIFAR-10 模型的预测性能。

- 训练过程中一些网络行为的可视化，这些行为包括输入图像、损失情况、网络行为的分布情况以及梯度；
- 算法学习参数的移动平均值的计算函数，以及在评估阶段使用这些平均值提高预测性能；
- 实现了一种机制，使得学习率随着时间的推移而递减；
- 为输入数据设计预存取队列，将磁盘延迟和高开销的图像预处理操作与模型分离开来处理；

我们也提供了模型的多 GUP 版本，用以表明：

- 可以配置模型后使其在多个 GPU 上并行的训练
- 可以在多个 GPU 之间共享和更新变量值

我们希望本教程给大家开了个头，使得在 Tensorflow 上可以为视觉相关工作建立更大的 Cnns 模型

#### 4.5.4 模型架构

本教程中的模型是一个多层架构，由卷积层和非线性层 (nonlinearities) 交替多次排列后构成。这些层最终通过全连通层对接到 softmax 分类器上。这一模型除了最顶部的几层外，基本跟 Alex Krizhevsky 提出的模型一致。在一个 GPU 上经过几个小时的训练后，该模型达到了最高 86% 的精度。细节请查看下面的描述以及代码。模型中包含了 1,068,298 个学习参数，分类一副图像需要大概 19.5M 个乘加操作。

#### 4.5.5 代码组织

本教程的代码位于[tensorflow/models/image/cifar10/](https://github.com/tensorflow/models/tree/master/image/cifar10/)

### 4.5.6 CIFAR-10 模型

CIFAR-10 网络模型部分的代码位于[cifar10.py](#), 完整的训练图中包含约 765 个操作。但是我们发现通过下面的模块来构造训练图可以最大限度的提高代码复用率:

1. 模型输入: 包括 `inputs()`、`distorted_inputs()` 等一些操作, 分别用于读取 CIFAR 的图像并进行预处理, 做为后续评估和训练的输入;
2. 模型预测: 包括 `inference()` 等一些操作, 用于进行统计计算, 比如在提供的图像进行分类; adds operations that perform inference, i.e. classification, on supplied images.
3. 模型训练: 包括 `loss()` and `train()` 等一些操作, 用于计算损失、计算梯度、进行变量更新以及呈现最终结果。

### 4.5.7 模型输入

输入模型是通过 `inputs()` 和 `distorted_inputs()` 函数建立起来的, 这 2 个函数会从 CIFAR-10 二进制文件中读取图片文件, 由于每个图片的存储字节数是固定的, 因此可以使用 `tf.FixedLengthRecordReader` 函数。更多的关于 Reader 类的功能可以查看[Reading Data](#)。图片文件的处理流程如下:

- 图片会被统一裁剪到 24x24 像素大小, 裁剪中央区域用于评估或[随机](#)裁剪用于训练;
- 图片会进行[近似的白化](#)处理, 使得模型对图片的动态范围变化不敏感。

对于训练, 我们另外采取了一系列随机变换的方法来人为的增加数据集的大小:

- 对图像进行[随机的左右翻转](#);
- 随机变换[图像的亮度](#);
- 随机变换[图像的对比度](#);

可以在[Images](#)页的列表中查看所有可用的变换, 对于每个原始图我们还附带了一个 `image_summary`, 以便于在 TensorBoard 中查看。这对于检查输入图像是否正确十分有用。



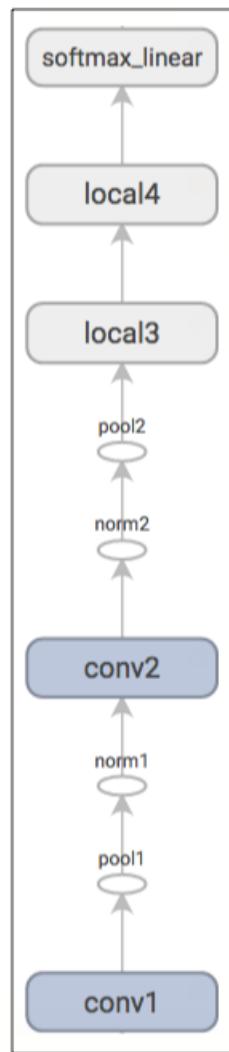
从磁盘上加载图像并进行变换需要花费不少的处理时间。为了避免这些操作减慢训练过程, 我们在 16 个独立的线程中并行进行这些操作, 这 16 个线程被连续的安排在一个 TensorFlow [队列](#)中。

#### 4.5.8 模型预测

模型的预测流程由 `inference()` 构造，该函数会添加必要的操作步骤用于计算预测值的 logits，其对应的模型组织方式如下所示：

Layer 名称	描述
conv1	实现卷积以及 rectified linear activation
pool1	max pooling
norm1	局部响应归一化
conv2	卷积 and rectified linear activation
norm2	局部响应归一化
pool2	max pooling
local3	基于修正线性激活的全连接层
local4	基于修正线性激活的全连接层
softmax_linear	进行线性变换以输出 logits.

这里有一个由 TensorBoard 绘制的图形，用于描述模型建立过程中经过的步骤：

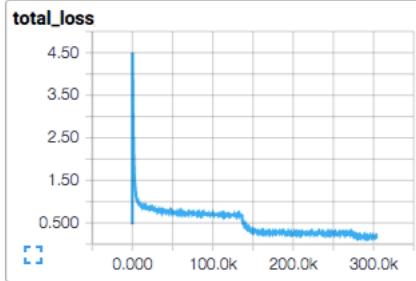


练习: inference 的输出是未归一化的 logits, 尝试使用 `tf.softmax()` 修改网络架构后返回归一化的预测值。`inputs()` 和 `inference()` 函数提供了评估模型时所需的所有构件, 现在我们把讲解的重点从构建一个模型转向训练一个模型。

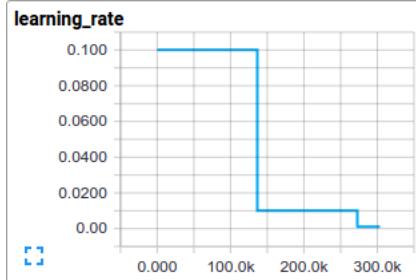
练习: `inference()` 中的模型跟[cuda-convnet](#)中描述的 CIFAR-10 模型有些许不同, 其差异主要在于其顶层不是全连接层而是局部连接层, 可以尝试修改网络架构来准确的复制全连接模型。

**模型训练** 训练一个可进行 N 维分类的网络的常用方法是使用多项式逻辑回归, 又被叫做 softmax 回归。Softmax 回归在网络的输出层上附加了一个 softmax nonlinearity, 并且计算归一化的预测值和 label 的 1-hot encoding 的交叉熵。在正则化过程中, 我们会对所有学习变量应用权重衰减损失。模型的目标函数是求交叉熵损失和所有权重衰减项的和, `loss()`

函数的返回值就是这个值。在 TensorBoard 中使用 scalar\_summary 来查看该值的变化情况：



我们使用标准的梯度下降算法来训练模型（也可以在 Training 中看看其他方法），其学习率随时间以指数形式衰减。



#### 4.5.9 开始执行并训练模型

我们已经把模型建立好了，现在通过执行脚本 cifar10\_train.py 来启动训练过程。`python cifar10_train.py` 注意：当第一次在 CIFAR-10 教程上启动任何任务时，会自动下载 CIFAR-10 数据集，该数据集大约有 160M 大小，因此第一次运行时泡杯咖啡小憩一会吧。你应该可以看到如下类似的输出：

```

1 Filling queue with 20000 CIFAR images before starting to train. This will take a
   few minutes.
2 2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/
   batch)
3 2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/
   batch)
4 2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/
   batch)
5 2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/
   batch)
6 2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/
   batch)

```

```
7 2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/batch)
```

脚本会在每 10 步训练过程后打印出总损失值，以及最后一批数据的处理速度。下面是几点注释：

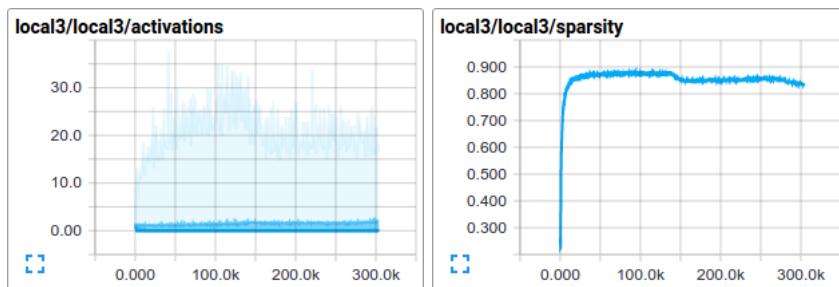
- 第一批数据会非常的慢（大概要几分钟时间），因为预处理线程要把 20,000 个待处理的 CIFAR 图像填充到重排队列中；
- 打印出来的损失值是最近一批数据的损失值的均值。请记住损失值是交叉熵和权重衰减项的和；
- 上面打印结果中关于一批数据的处理速度是在 Tesla K40C 上统计出来的，如果你运行在 CPU 上，性能会比此要低；

练习：当实验时，第一阶段的训练时间有时会非常的长，长到足以让人生厌。可以尝试减少初始化时初始填充到队列中图片数量来改变这种情况。在 cifar10.py 中搜索 NUM\_EXAMPLES\_PER\_EPOCH\_FOR\_TRAIN 并修改之。

cifar10\_train.py 会周期性的在检查点文件中保存模型中的所有参数，但是不会对模型进行评估。cifar10\_eval.py 会使用该检查点文件来测试预测性能（详见下面的描述：评估模型）。如果按照上面的步骤做下来，你应该已经开始训练一个 CIFAR-10 模型了。恭喜你！cifar10\_train.py 输出的终端信息中提供了关于模型如何训练的一些信息，但是我们可能希望了解更多关于模型训练时的信息，比如：

- 损失是真的在减小还是看到的只是噪声数据？
- 为模型提供的图片是否合适？
- 梯度、激活、权重的值是否合理？
- 当前的学习率是多少？

TensorBoard 提供了该功能，可以通过 cifar10\_train.py 中的 SummaryWriter 周期性的获取并显示这些数据。比如我们可以在训练过程中查看 local3 的激活情况，以及其特征维度的稀疏情况：



### 4.5.10 评估模型

现在可以在另一部分数据集上来评估训练模型的性能。脚本文件 `cifar10_eval.py` 对模型进行了评估，利用 `inference()` 函数重构模型，并使用了在评估数据集所有 10,000 张 CIFAR-10 图片进行测试。最终计算出的精度为 1:N，N= 预测值中置信度最高的一项与图片真实 label 匹配的频次。(It calculates the precision at 1: how often the top prediction matches the true label of the image)。为了监控模型在训练过程中的改进情况，评估用的脚本文件会周期性的在最新的检查点文件上运行，这些检查点文件是由 `cifar10_train.py` 产生。`python cifar10_eval.py`

注意：不要在同一块 GPU 上同时运行训练程序和评估程序，因为可能会导致内存耗尽。尽可能的在其它单独的 GPU 上运行评估程序，或者在同一块 GPU 上运行评估程序时先挂起训练程序。

你可能会看到如下输出

```
1 2015-11-06 08:30:44.391206: precision @ 1 = 0.860
2 ...
```

评估脚本只是周期性的返回 `precision@1` (The script merely returns the precision @ 1 periodically)–在该例中返回的准确率是 86%。`cifar10_eval.py` 同时也返回其它一些可以在 TensorBoard 中进行可视化的简要信息。可以通过这些简要信息在评估过程中进一步的了解模型。训练脚本会为所有学习变量计算其移动均值，评估脚本则直接将所有学习到的模型参数替换成对应的移动均值。这一替代方式可以在评估过程中提升模型的性能。

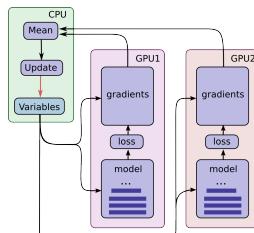
练习：通过 `precision @ 1` 测试发现，使用均值参数可以将预测性能提高约 3%，在 `cifar10_eval.py` 中尝试修改为不采用均值参数的方式，并确认由此带来的预测性能下降。

### 4.5.11 在多个 GPU 卡上训练模型

现代的工作站可能包含多个 GPU 进行科学计算。TensorFlow 可以利用这一环境在多个 GPU 卡上运行训练程序。在并行、分布式的环境中进行训练，需要对训练程序进行协调。对于接下来的描述，术语模型拷贝（model replica）特指在一个数据子集中训练出来的模型的一份拷贝。如果天真的对模型参数的采用异步方式更新将会导致次优的训练性能，这是因为我们可能会基于一个旧的模型参数的拷贝去训练一个模型。但与此相反采用完全同步更新的方式，其速度将会变得和最慢的模型一样慢 (Conversely, employing fully synchronous updates will be as slow as the slowest model replica.)。在具有多个 GPU 的工作站中，每个 GPU 的速度基本接近，并且都含有足够的内存来运行整个 CIFAR-10 模型。因此我们选择以下方式来设计我们的训练系统：

- 在每个 GPU 上放置单独的模型副本；
- 等所有 GPU 处理完一批数据后再同步更新模型的参数；

下图示意了该模型的结构：



可以看到，每一个 GPU 会用一批独立的数据计算梯度和估计值。这种设置可以非常有效的将一大批数据分割到各个 GPU 上。这一机制要求所有 GPU 能够共享模型参数。但是众所周知在 GPU 之间传输数据非常的慢，因此我们决定在 CPU 上存储和更新所有模型的参数（对应图中绿色矩形的位置）。这样一来，GPU 在处理一批新的数据之前会更新一遍的参数。图中所有的 GPU 是同步运行的。所有 GPU 中的梯度会累积并求平均值（绿色方框部分）。模型参数会利用所有模型副本梯度的均值来更新。

#### 4.5.12 在多设备中设置变量和操作

在多个设备中设置变量和操作时需要做一些特殊的抽象。我们首先需要把在单个模型拷贝中计算估计值和梯度的行为抽象到一个函数中。在代码中，我们称这个抽象对象为“tower”。对于每一个“tower”都需要设置它的两个属性：在一个 tower 中为所有操作设定一个唯一的名称。tf.name\_scope() 通过添加一个范围前缀来提供该唯一名称。比如，第一个 tower 中的所有操作都会附带一个前缀 tower\_0，示例：tower\_0/conv1/Conv2D；在一个 tower 中运行操作的优先硬件设备。tf.device() 提供该信息。比如，在第一个 tower 中的所有操作都位于 device('/gpu:0') 范围中，暗含的意思是这些操作应该运行在第一块 GPU 上；为了在多个 GPU 上共享变量，所有的变量都绑定在 CPU 上，并通过 tf.get\_variable() 访问。可以查看 Sharing Variables 以了解如何共享变量。

#### 4.5.13 启动并在多个 GPU 上训练模型

如果你的机器上安装有多块 GPU，你可以通过使用 cifar10\_multi\_gpu\_train.py 脚本来加速模型训练。该脚本是训练脚本的一个变种，使用多个 GPU 实现模型并行训练。  
python cifar10\_multi\_gpu\_train.py --num\_gpus=2 训练脚本的输出如下所示：

```
1 Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.
```

```

2 2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/
batch)
3 2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/
batch)
4 2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/
batch)
5 2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/
batch)
6 2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/
batch)
7 2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/
batch)
8 ...

```

需要注意的是默认的 GPU 使用数是 1，此外，如果你的机器上只有一个 GPU，那么所有的计算都只会在一个 GPU 上运行，即便你可能设置的是 N 个。

练习：cifar10\_train.py 中的批处理大小默认配置是 128。尝试在 2 个 GPU 上运行 cifar10\_multi\_gpu\_train.py 脚本，并且设定批处理大小为 64，然后比较 2 种方式的训练速度

#### 4.5.14 下一步

恭喜你！你已经完成了 CIFAR-10 教程。如果你对开发和训练自己的图像分类系统感兴趣，我们推荐你新建一个基于该教程的分支，并修改其中的内容以建立解决您问题的图像分类系统。

练习：下载 Street View House Numbers (SVHN) 数据集。新建一个 CIFAR-10 教程的分支，并将输入数据替换成 SVHN。尝试改变网络结构以提高预测性能。

## 4.6 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件了解新的事件。

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。

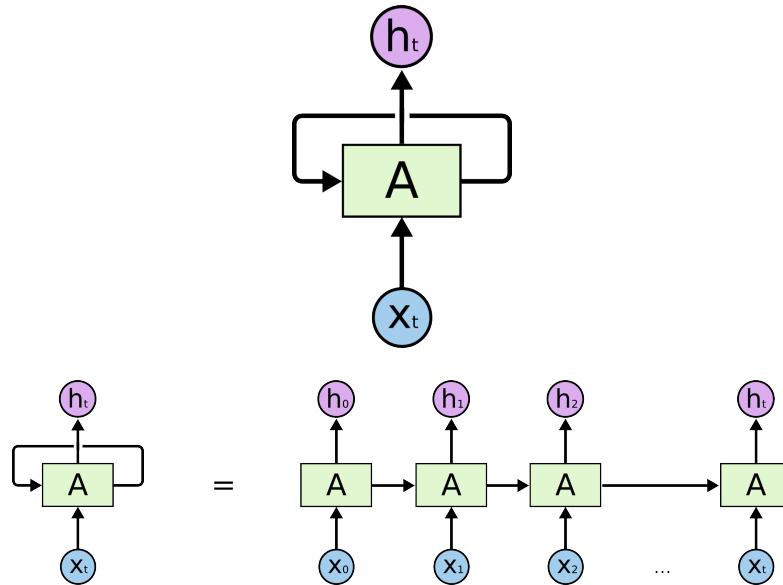
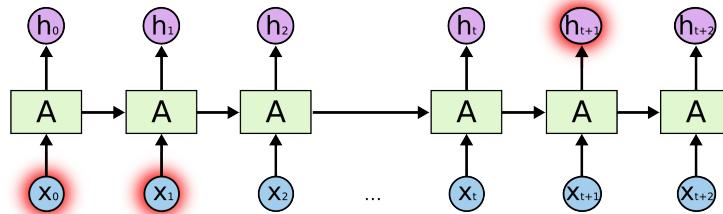


图 4.1: unrolled RNN

#### 4.6.1 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。

#### 4.6.2 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，它被 [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。

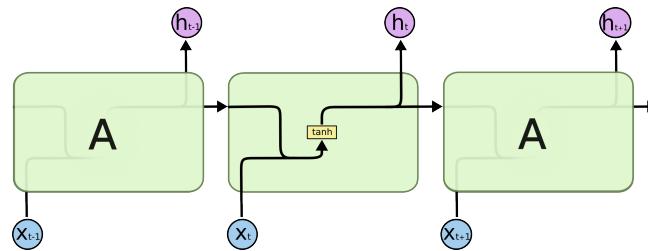


图 4.2: The repeating module in a standard RNN contains a single layer

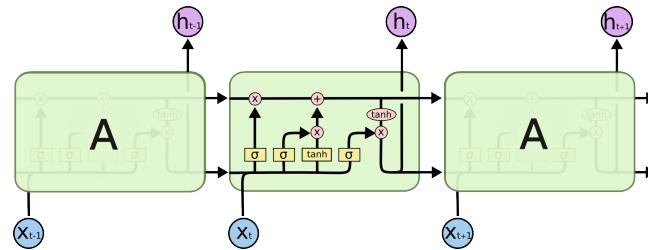


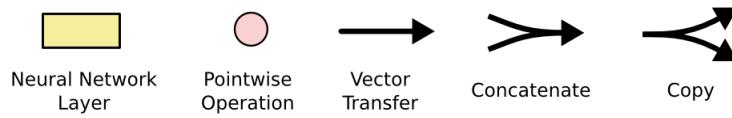
图 4.3: The repeating module in an LSTM contains four interacting layers

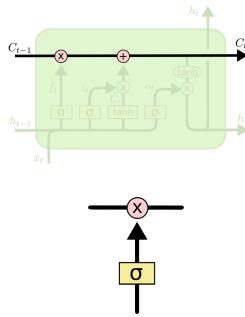
LSTMs 明确的设计去解决 long-term dependency problem。

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它输入，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。

### 4.6.3 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。





#### 4.6.4 一步步的设置

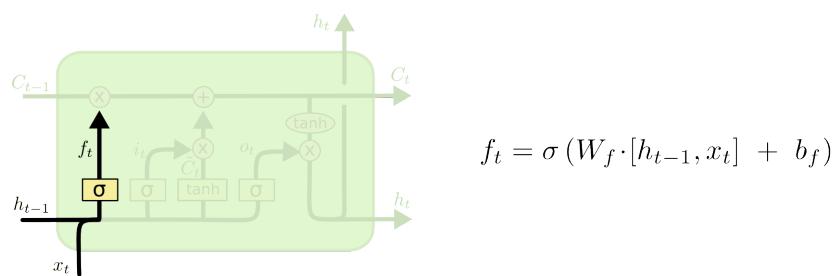
第一步是 LSTMs 决定什么信息应该被传送，这个决定每一个称为忘记门的 sigmoid layer 组成，通过  $h_{t-1}$  和  $x_t$  输出 0 到 1 之间的数给当前的  $C_{t-1,1}$  表示完全保持，0 表示丢弃。

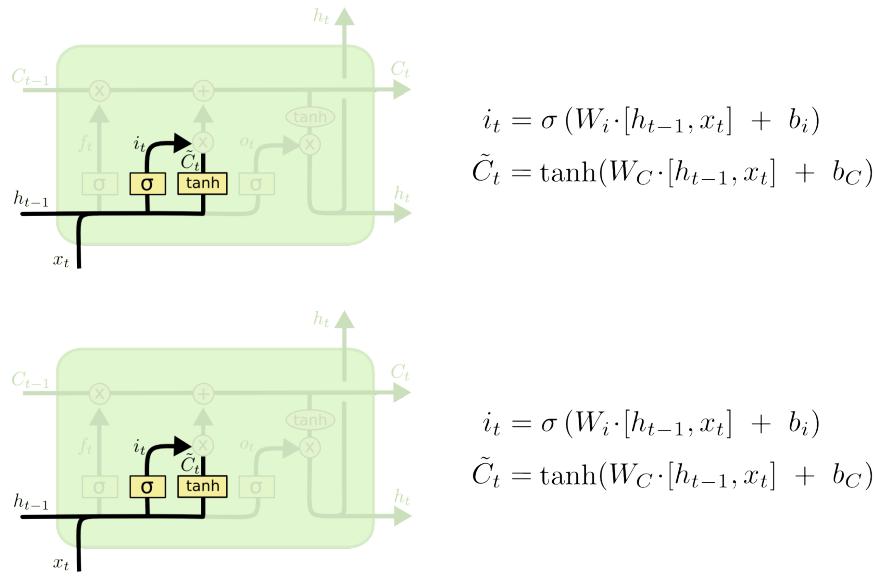
对于上面的语言模型，cell state 也许包含 the gender of the present subjects，以至于正确的带名字能被使用，当我们看一个新的 subject，我们想图忘记 the gender of the old subject。下一步是决定什么新的信息将被存储在 cell state 中，这分为两部分

1. Sigmod 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



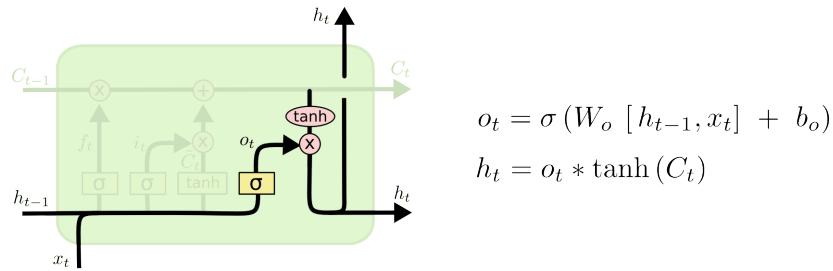


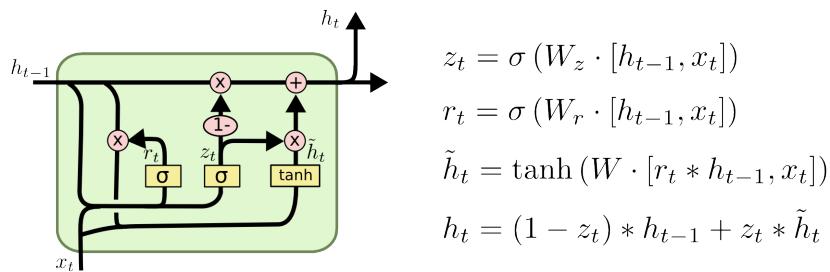
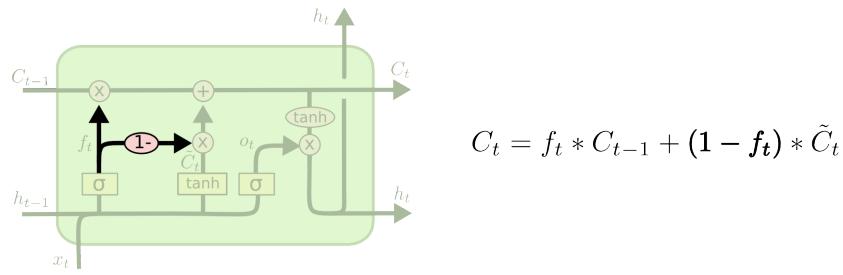
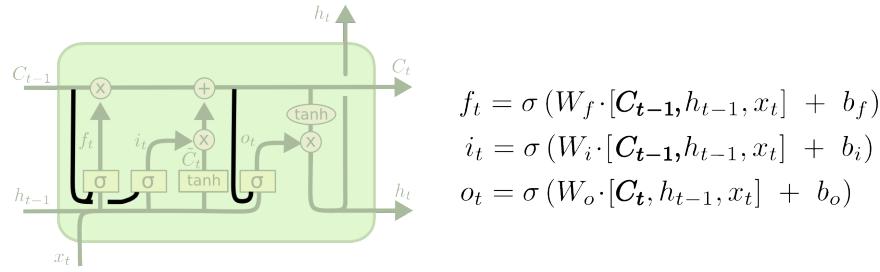
接下来应该是动词的什么形式。

#### 4.6.5 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? 它们的差别大吗? Greff, et al. (2015) 做了一些比较了一些流行的变体, 发现它们基本相同。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。





## 4.7 向量字表示

### 4.7.1 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。

### 4.7.2 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  $score(w_t, h)$  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：

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

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。

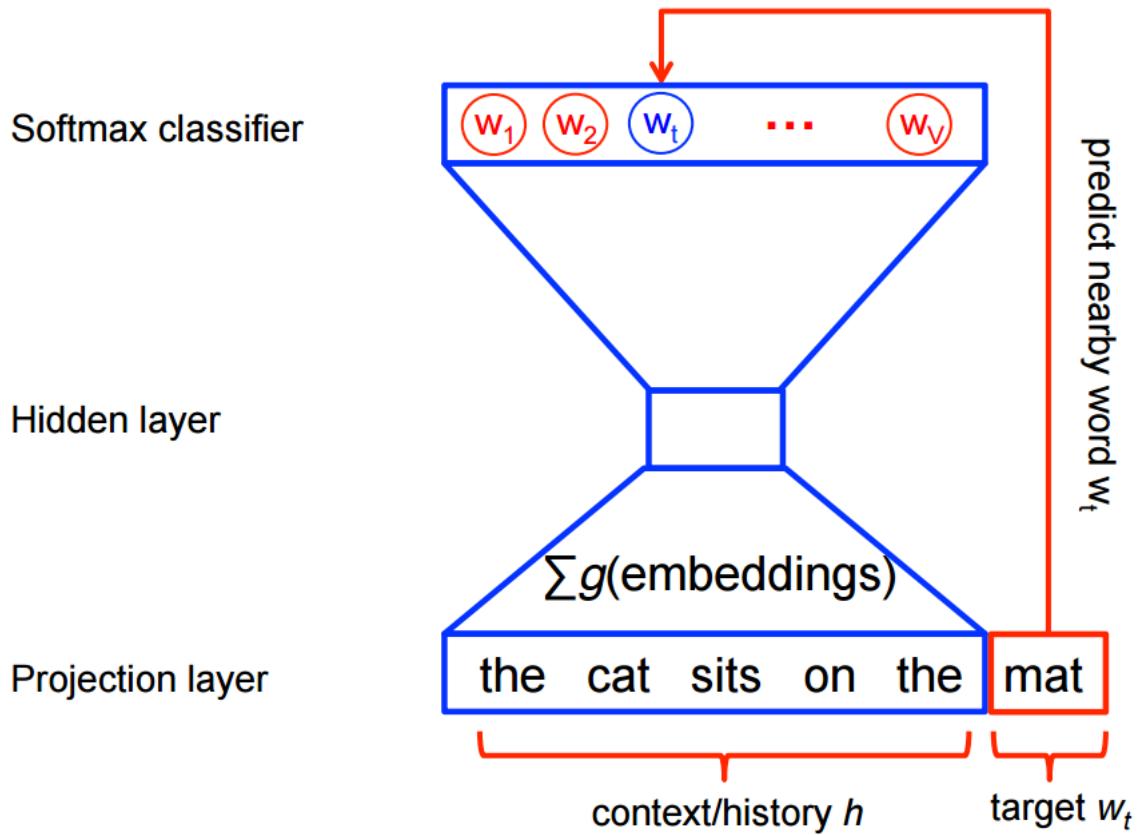


图 4.4: CBOW 方法

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。

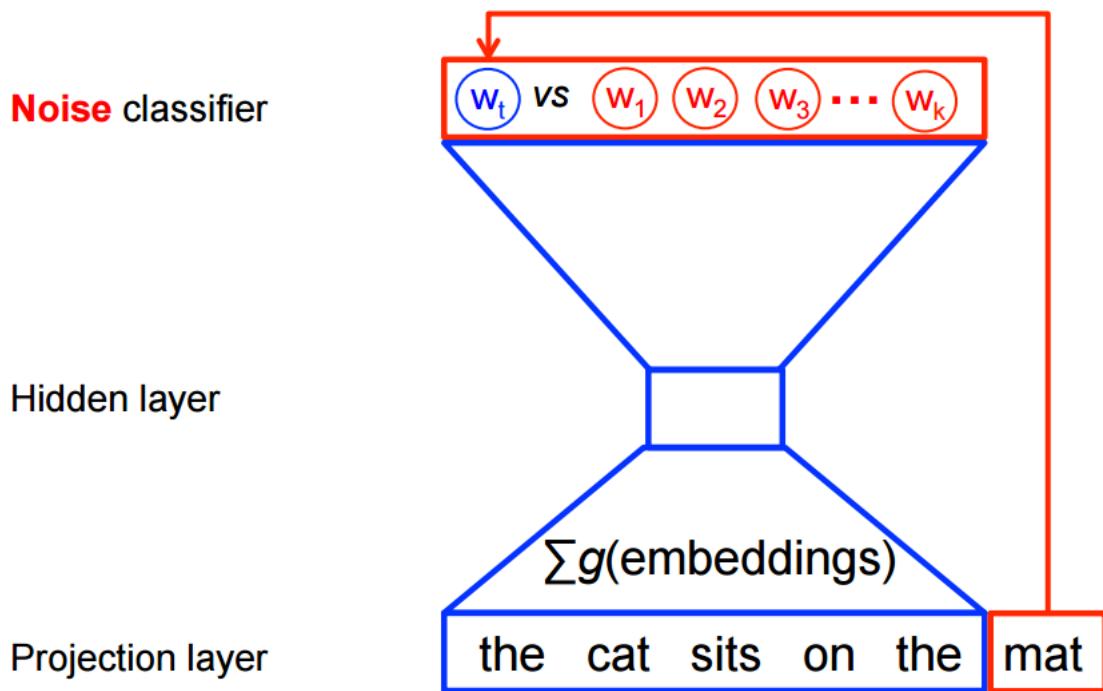


图 4.5: Skip-Gram

从数学的角度来说，我们的目标是对每个样本最大化：

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

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  $\theta$  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()`。

#### 4.7.3 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：

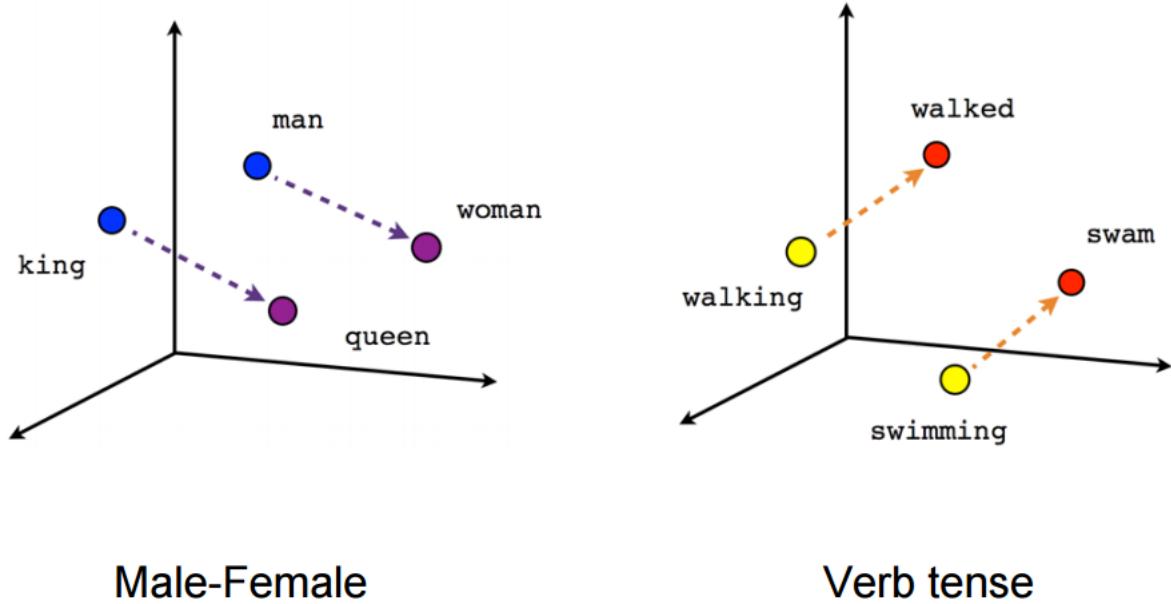
$$J_{NEG}^{(t)} = \log Q_\theta(D=1|the, quick) + \log(Q_\theta(D=0|sleep, quick))$$

整个计算过程的目标是通过更新嵌套参数  $\theta$  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  $\theta$  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)

```

#### 4.7.4 训练过程

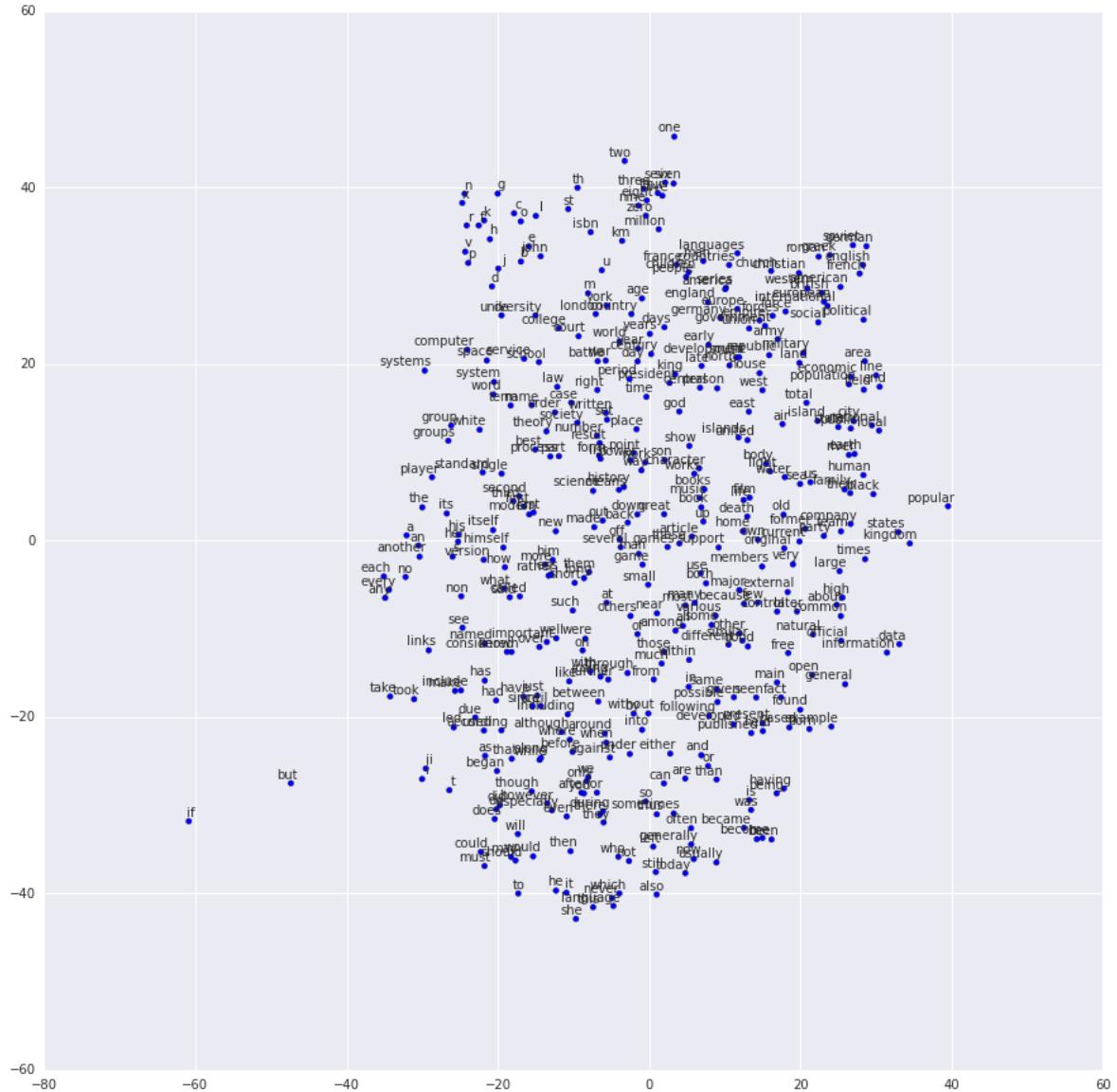
训练的过程很简单，只要在循环中使用 `feed_dict` 不断给占位符填充数据，同时调用 `session.run` 即可。

```

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)

```

### 4.7.5 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](#)

### 4.7.6 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。

#### 4.7.7 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。

#### 4.7.8 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，它们在 PTB 数据集上得到了很棒的结果。

### 4.7.9 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，让它们各自有唯一的整型标识符，便于神经网络处理。

### 4.7.10 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)
```

### 4.7.11 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
```

### 4.7.12 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。

### 4.7.13 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）

### 4.7.14 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
9     # 其余的代码.
10    #
11
12 final_state = state

```

#### 4.7.15 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

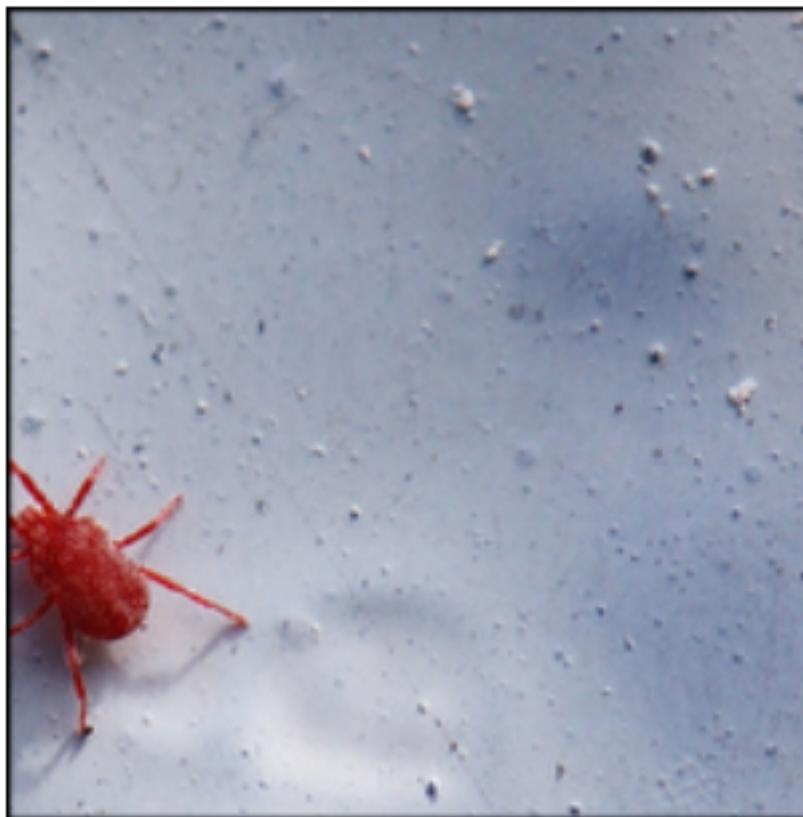
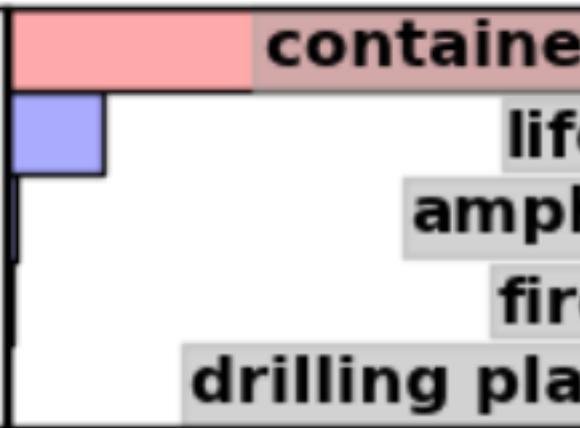
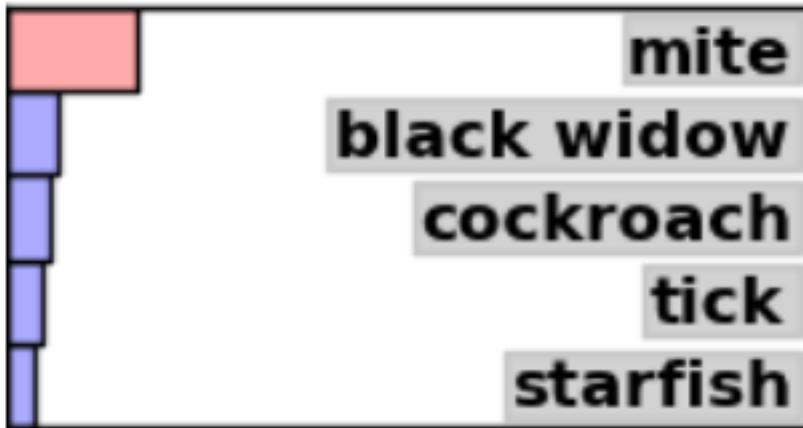
```

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。

## 4.8 图像识别

我们的大脑看东西很简单，它不需要得到任何人的告诉就能分别狮子，钱包，读一个符号或者识别人脸，但是计算机解决这个问题却很难。最近几年机器学习已经在这个问题上取得了很大的进步，事实上我们已经发现了一个称谓 CNN 的模型可以在很难的视觉认知领域上获得匹敌甚至超过人类的性能。研究人员在 ImageNet 上展示它们的计算机视觉的成果，模型性能持续提升从[QuocNet](#),[AlexNet](#),[Inception\(GoogleNet\)](#),[BN-inception-v2](#), Google 累不和外部的研究人员都已经出版了一些论文描述这些模型，但是结果任然河南再次提升，我们用下面的代码实现我们最新的模型[Inception-v3](#),[Inception-v3](#) 2012 年在 Imagenet 的 LVRG 上训练，这对计算机来说是一个基本的任务，这里模型尝试着分类整个图形为 1000 中分类像“斑马”，“大麦町犬”，“洗碗机”，下面是 AlexNet 分类的一些图像：

**mite****container s**

为了对比模型，我们检查迷行预测正确结果为”top-5 error rate”,AlexNet2012 年的验证数据及上得到 top-5 error rate 15.3%,Inception(GoogleLeNet) 得到 6.67%,BN-Inception-v2 得到 4.9%,Inception-v3 得到 3.46%。

人类在 ImageNet 中做的怎么样，由 Andrej Karpathy 的[博客](#)尝试测量人们的

性能，它得到了 5.1% 的 top-5 error rate

下面的导航将教你如何使用 Inception-v3. 你将学习用 Python 或者 C++ 学习分类图像成 1000 类，我们也将讨论如何从模型提取高级特征重用到其它的视觉任务。

### 4.8.1 用 Python API

第一次运行时 `classify_image.py` 将从 tensorflow 官网下载训练好的模型，你将需要 200M 左右的硬盘空间。开始从 github 上 clone[Google 模型](#)，然后运行命令：

```
5 cd models/tutorials/image/imagenet
6 python classify_image.py
```

上面的命令将分类下面的熊猫图片。



如果模型正确运行将生成下面的输出：

```
5 giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (score =
    0.88493)
6 indri, indris, Indri indri, Indri brevicaudatus (score = 0.00878)
7 lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score =
    0.00317)
8 custard apple (score = 0.00149)
9 earthstar (score = 0.00127)
```

如果你希望添加其它的 JPEG 图片，你也许需要编辑`-image_file`参数，如果你需要下载模型数据到不同的目录，你将需要指定`-model_dir`到使用的目录。

### 4.8.2 用 C++ API

你可以在生产环境上使用 C++ 运行相同的 Inception-v3 模型。你可以下载包含像这样定义的 GraphDef 的打包文件(从 TensorFlow 仓库的根目录运行)

```
7 curl -L "https://storage.googleapis.com/download.tensorflow.org/models/
    inception_v3_2016_08_28_frozen.pb.tar.gz" |
8 tar -C tensorflow/examples/label_image/data -xz
```

下一步我们编译包含 C++ 代码载入的二进制运行图，如果你已经按照[the instructions to download the source installation of TensorFlow](#) 配置了你的平台，你将能在 shell 终端通过下面的命令构建例子运行：

```
9  bazel build tensorflow/examples/label_image/...
```

你可以像这样创建一个二进制的执行文件：

```
10 bazel-bin/tensorflow/examples/label_image/label_image
```

用框架默认的例子图片，输出下面的结果：

```
11 I tensorflow/examples/label_image/main.cc:206] military uniform (653): 0.834306
12 I tensorflow/examples/label_image/main.cc:206] mortarboard (668): 0.0218692
13 I tensorflow/examples/label_image/main.cc:206] academic gown (401): 0.0103579
14 I tensorflow/examples/label_image/main.cc:206] pickelhaube (716): 0.00800814
15 I tensorflow/examples/label_image/main.cc:206] bulletproof vest (466):
    0.00535088
```

在这个例子中你用一张默认的图片[Admiral Grace Hopper](#)，你可以看到网络正确识别她的军官制服，得分 0.8。



下一步提供--image= 参数分辨你的图片

```
16 bazel-bin/tensorflow/examples/label_image/label_image --image=my_image.png
```

如果你查看[tensorflow/examples/label\\_image/main.cc](#)你可以明白它是如何工作的，我们希望代码能帮你整合 TensorFlow 到你的应用，因此我们将一步步的浏览主函数。命令行参数控制文件从哪里载入，即输入图像的内容。模型希望得到  $299 \times 299$  的 RGB 图像，分别是 input\_width,input\_height，我们需要缩放图像的像素从 0-255 到图能操作的一个浮点值。我们通过 input\_mean 和 input\_std 控制缩放：我们首先用像素值减去 input\_mean

然后除以 input\_std, 这些值看起来可能有点琢磨不透, 但是它们被原始作者定义, 她/它想训练的时候什么值被匹配, 你可以看看我们如何在[ReadTensorFromImageFile\(\)](#)函数中如何应用一张图像。

```
17 // Given an image file name, read in the data, try to decode it as an image,
18 // resize it to the requested size, and then scale the values as desired.
19 Status ReadTensorFromImageFile(string file_name, const int input_height,
20                                 const int input_width, const float input_mean,
21                                 const float input_std,
22                                 std::vector<Tensor>* out_tensors) {
23     tensorflow::GraphDefBuilder b;
```

我们开始撞见亿个 GraphDefBuilder 对象指定一个运行或者载入的模型。

```
24     string input_name = "file_reader";
25     string output_name = "normalized";
26     tensorflow::Node* file_reader =
27         tensorflow::ops::ReadFile(tensorflow::ops::Const(file_name, b.opts()),
28                                b.opts().WithName(input_name));
```

然后我们开始为我们想要运行或者载入的小模型创建节点, 变换大小, 缩放像素值得到主函数希望得到的结果, 我们创建的第一个节点是保持图像名字的 tensor 想要载入 Const 操作然后作为第一个输入传入 ReadFile 操作, 你也许注意到我们传递 b.opts() 操作给所有的创建操作, 参数确保节点被添加到 GraphDefBuilder 的模型定义。这给了一个名字给节点, 对于给一个名字节点不是严格需要的因为如果你不指定这个将自动命名, 但是它使得调试变得简单

```
29     // Now try to figure out what kind of file it is and decode it.
30     const int wanted_channels = 3;
31     tensorflow::Node* image_reader;
32     if (tensorflow::StringPiece(file_name).ends_with(".png")) {
33         image_reader = tensorflow::ops::DecodePng(
34             file_reader,
35             b.opts().WithAttr("channels", wanted_channels).WithName("png_reader"));
36     } else {
37         // Assume if it's not a PNG then it must be a JPEG.
38         image_reader = tensorflow::ops::DecodeJpeg(
39             file_reader,
40             b.opts().WithAttr("channels", wanted_channels).WithName("jpeg_reader"));
41     }
42     // Now cast the image data to float so we can do normal math on it.
43     tensorflow::Node* float_caster = tensorflow::ops::Cast(
44         image_reader, tensorflow::DT_FLOAT, b.opts().WithName("float_caster"));
45     // The convention for image ops in TensorFlow is that all images are expected
46     // to be in batches, so that they're four-dimensional arrays with indices of
```

```

47 // [batch, height, width, channel]. Because we only have a single image, we
48 // have to add a batch dimension of 1 to the start with ExpandDims().
49 tensorflow::Node* dims_expander = tensorflow::ops::ExpandDims(
50     float_caster, tensorflow::ops::Const(0, b.opts()), b.opts());
51 // Bilinearly resize the image to fit the required dimensions.
52 tensorflow::Node* resized = tensorflow::ops::ResizeBilinear(
53     dims_expander, tensorflow::ops::Const({input_height, input_width},
54                                         b.opts().WithName("size")),
55     b.opts());
56 // Subtract the mean and divide by the scale.
57 tensorflow::ops::Div(
58     tensorflow::ops::Sub(
59         resized, tensorflow::ops::Const({input_mean}, b.opts(), b.opts()),
60         tensorflow::ops::Const({input_std}, b.opts(),
61         b.opts().WithName(output_name)));

```

我们然后添加更多的节点解码文件数据为图片，转化整数为浮点值，变形然后最后运行减去和除法操作：

```

62 // This runs the GraphDef network definition that we've just constructed, and
63 // returns the results in the output tensor.
64 tensorflow::GraphDef graph;
65 TF_RETURN_IF_ERROR(b.ToGraphDef(&graph));

```

在这最后我们有一个存储在变量 b 中的模型定义我们用 ToGraphDef() 转化完整的图定义

```

66 std::unique_ptr<tensorflow::Session> session(
67     tensorflow::NewSession(tensorflow::SessionOptions()));
68 TF_RETURN_IF_ERROR(session->Create(graph));
69 TF_RETURN_IF_ERROR(session->Run({}, {output_name}, {}, out_tensors));
70 return Status::OK();

```

我们创建一个运行图的接口 tf.Session 对象，运行它，指定我们想从那个节点获得输出和放输出数据到哪里，这给我们一个 Tensor 对象的向量，这个对象将变成一个三个对量，你可以将 Tensor 认为是多维数组，它保存 299 像素宽，299 像素高。3 通道的图像作为浮点值，如果你有自己的图像处理框架，你应该能用它替代，只要你在输入图像到主图中时应用相同的转换。这是用 C++ 创建一个简单的 TensorFlow 图的例子，但是对于预先训练好的 Inception 模型我们想从文件载入一个大的定义，你可以在 LoadGraph() 函数中查看我们如何做的：

```

71 // Reads a model graph definition from disk, and creates a session object you
72 // can use to run it.
73 Status LoadGraph(string graph_file_name,
74                   std::unique_ptr<tensorflow::Session>* session) {
75     tensorflow::GraphDef graph_def;

```

```

76 Status load_graph_status =
77     ReadBinaryProto(tensorflow::Env::Default(), graph_file_name, &graph_def);
78 if (!load_graph_status.ok()) {
79     return tensorflow::errors::NotFound("Failed to load compute graph at '",
80                                         graph_file_name, "'");
81 }

```

如果你已经查看的图像载入代码，一些名词应该很熟悉。想必须用一个 GraphDefBuffer 残生一个 GraphDef 对象，我们载入一个包含 GraphDef 的 protobuf 文件。

```

82 session->reset(tensorflow::NewSession(tensorflow::SessionOptions()));
83 Status session_create_status = (*session)->Create(graph_def);
84 if (!session_create_status.ok()) {
85     return session_create_status;
86 }
87 return Status::OK();
88 }

```

我们从 GraphDef 创建一个 Session 对象传递它给调用器以至于我们稍后能使用。GetTopLabels() 函数有些像函数载入，但是我们希望运行主图得到结果，存储它进一个按照标签最高评分的结果，像图片载入，它创建一个的 GraphDefBuilder，添加一对节点上去然后运行短图割刀一对输出 tensor，在这个例子中它们代表排序的分数和最好可能性的结果的索引位置。

```

89 // Analyzes the output of the Inception graph to retrieve the highest scores and
90 // their positions in the tensor, which correspond to categories.
91 Status GetTopLabels(const std::vector<Tensor>& outputs, int how_many_labels,
92                      Tensor* indices, Tensor* scores) {
93     tensorflow::GraphDefBuilder b;
94     string output_name = "top_k";
95     tensorflow::ops::TopK(tensorflow::ops::Const(outputs[0], b.opts()),
96                           how_many_labels, b.opts().WithName(output_name));
97     // This runs the GraphDef network definition that we've just constructed, and
98     // returns the results in the output tensors.
99     tensorflow::GraphDef graph;
100    TF_RETURN_IF_ERROR(b.ToGraphDef(&graph));
101    std::unique_ptr<tensorflow::Session> session(
102        tensorflow::NewSession(tensorflow::SessionOptions()));
103    TF_RETURN_IF_ERROR(session->Create(graph));
104    // The TopK node returns two outputs, the scores and their original indices,
105    // so we have to append :0 and :1 to specify them both.
106    std::vector<Tensor> out_tensors;
107    TF_RETURN_IF_ERROR(session->Run({}, {output_name + ":0", output_name + ":1"},
108                        {}, &out_tensors));

```

```

109 *scores = out_tensors[0];
110 *indices = out_tensors[1];
111 return Status::OK();

```

PrintTopLabels() 函数得到排序接軌哦用一种友好的方式打印它们。CkeckTopLabel() 函数是非常累是，但是仅仅确保顶部的标签是我们想要的，用于调试。在最后 main() 结合所有的函数调用

```

112 int main(int argc, char* argv[]) {
113     // We need to call this to set up global state for TensorFlow.
114     tensorflow::port::InitMain(argv[0], &argc, &argv);
115     Status s = tensorflow::ParseCommandLineFlags(&argc, argv);
116     if (!s.ok()) {
117         LOG(ERROR) << "Error parsing command line flags: " << s.ToString();
118         return -1;
119     }
120
121     // First we load and initialize the model.
122     std::unique_ptr<tensorflow::Session> session;
123     string graph_path = tensorflow::io::JoinPath(FLAGS_root_dir, FLAGS_graph);
124     Status load_graph_status = LoadGraph(graph_path, &session);
125     if (!load_graph_status.ok()) {
126         LOG(ERROR) << load_graph_status;
127         return -1;
128     }

```

我们载入主图:

```

129 // Get the image from disk as a float array of numbers, resized and normalized
130 // to the specifications the main graph expects.
131 std::vector<Tensor> resized_tensors;
132 string image_path = tensorflow::io::JoinPath(FLAGS_root_dir, FLAGS_image);
133 Status read_tensor_status = ReadTensorFromFile(
134     image_path, FLAGS_input_height, FLAGS_input_width, FLAGS_input_mean,
135     FLAGS_input_std, &resized_tensors);
136 if (!read_tensor_status.ok()) {
137     LOG(ERROR) << read_tensor_status;
138     return -1;
139 }
140 const Tensor& resized_tensor = resized_tensors[0];

```

载入，变形，处理输入图像

```

141 // Actually run the image through the model.
142 std::vector<Tensor> outputs;
143 Status run_status = session->Run({ {FLAGS_input_layer, resized_tensor} },

```

```

144                     {FLAGS_output_layer}, {}, &outputs);
145     if (!run_status.ok()) {
146         LOG(ERROR) << "Running model failed: " << run_status;
147         return -1;
148     }

```

我们用图像作为输入运行载入的图:

```

149 // This is for automated testing to make sure we get the expected result with
150 // the default settings. We know that label 866 (military uniform) should be
151 // the top label for the Admiral Hopper image.
152 if (FLAGS_self_test) {
153     bool expected_matches;
154     Status check_status = CheckTopLabel(outputs, 866, &expected_matches);
155     if (!check_status.ok()) {
156         LOG(ERROR) << "Running check failed: " << check_status;
157         return -1;
158     }
159     if (!expected_matches) {
160         LOG(ERROR) << "Self-test failed!";
161         return -1;
162     }
163 }

```

出于测试的目的我们可以检查确保我们得到我们想要的结果:

```

164 // Do something interesting with the results we've generated.
165 Status print_status = PrintTopLabels(outputs, FLAGS_labels);

```

最后我们打印我们输出的标签:

```

166 if (!print_status.ok()) {
167     LOG(ERROR) << "Running print failed: " << print_status;
168     return -1;
169 }

```

我们用 TensorFlow 的 State 对象处理错误，它很方便因为它用 ok() 作为检查器检查是否出现的任何错误，然后打印出可读的错误消息，在这个例子中我们一站式对象识别，但是你应该能用类似的代码在你发现的其它模型上或者自己训练在不同的领域。我们希望这个小的例子给你一些灵感如何用 TensorFlow 和你的产品。

转换学习是想法，如果你知道如何很好的解决问题，你应该能转换一些解决相关问题的理解，一种执行转换学习的方法是移除网络最后的分类层提取[next-to-last layer of the CNN](#)，在这个例子中 2048 维向量，有一个关于这个如何做到的向导[in the how-to section](#)

### 4.8.3 更多学习资源

为了学习神经网络，Michael Nielsen 的[free online book](#), 类似对于卷积神经网络 Chris Olah 有一些[nice blog posts](#), Michael Nielsen 的书有一个关于它的[great chapter covering them](#)为了找出更多的卷积神经网络的实现你可以调到[deep convolutional networks tutorial](#), 最后如果你想加速在这一领域的研究，你可以读导航中列出的最近的相关工作的论文。

## 4.9 TensorFlow 实现大规模线性模型

tf.estimator API 提供了一些丰富的工具在 TensorFlow 中处理线性模型，这个章节提供了一个官员这些工具的概览，解释如下：

- 线性模型是什么
- 为什么你想用线性模型
- tf.estimator 如何使得在 TensorFlow 中建立一个线性模型变得简单
- 如何结合线性模型和深度学习的有点

读这个概览决定了是否 tf.estimator 对于你来说有用，然后做一下<https://www.tensorflow.org/tutorials/wide>这个概览的代码来自于这个章节，但是导航更详细的调通代码。为了明白为了理解这个概念熟悉一些基本的机器学习概念和 tf.estimator 是有帮助的。

### 4.9.1 什么是线性模型

一个线性模型用一个特征权重和作出预测，例如，如果你有关于年龄。教育年数，每周工作小时数的数据，你可以了解这些权重以至于它们的权重求和评估一个人的薪水。你可以用线性模型分类，一些线性模型转化权重为一个更方便的形式，例如逻辑回归转化权重和维逻辑函数输出 0-1 之间的数值。但是你依然有对于每个输入特征你依然有一个权重。

### 4.9.2 为什么你想用线性模型？

当最近的研究已经展示了有很多层的复杂神经网络的能力为什么我们想用一个简单的模型？

线性模型：

- 比深度神经网络训练快速
- 在大型数据集上也能工作的很好

- 训练的时候不要求微小的学习率
- 相比神经网络能更简单轻松地调试，你可以检查付给给个特征的权重找出那个对于预测结果的影响最大。
- 对于了解机器学习提供了很好的起点
- 工业上广泛的使用

### 4.9.3 tf.estimator 将如何构建线性模型

你可以通过 TensorFlow 建立线性模型不需要特殊的 API，但是 tf.estimator 提供了一些工具帮助你更轻松地构建搞笑的大规模线性模型。

#### 特征列和线性模型

设计的线性模型的大部分工作是转化原始数据为合适的输入特征，TensorFlow 用 FeatureColumn 概念启动这些转换，一个 FeatureColumn 代表你的数据的单个特征，一个 FeatureColumn 也许代表一些像”height”或者也许代表一个像”eye\_color”的种类这里的值是来自离散的可能性像’blue’, ’brown’, ’green’。连续的特征像’height’和绝对的特征’eye\_color’，在输入模型前一个单个的值可能被转化为数值序列。FeatureColumn 概念让你用但一个语义单元操作特征，你可以转化选择特征来包含没有用指定索引处理的处理。

#### 稀疏列

绝对的特征在线性模型中被转化为稀疏向量，在向量中每个可能值有一个相关的索引或者 id，例如如果仅有三个可能的颜色代表’eye\_color’作为长度为 3 的向量，’brown’将变为 [1,0,0], ’blue’将变成 [0,1,0], ’green’将变成 [0,0,1]，这里向量被称为稀疏因为当可能值很大的时候。它们可能很长但是有很多 0。尽管我们不需要用绝对列用 tf.estimator 线性模型，一个有力的线性模型能处理大型的稀疏向量，tf.estimator 线性模型工具的稀疏特征也正是一个主要的用途。

#### 编码稀疏列

FeatureColumn 用下面的代码自动处理传统的绝对的值称为向量：

```
170 eye_color = tf.feature_column.categorical_column_with_vocabulary_list(
171     "eye_color", vocabulary_list=["blue", "brown", "green"])
```

这里的 eye\_color 是你的源数据的列的名字，入股你不知道所有的可能值你可以为你的绝对特征以生成 FeatureColumn。对于这种情况，你将用 categorical\_column\_with\_hash\_bucket()，用一盒散列函数赋值索引给特征值。

```
172 education = tf.feature_column.categorical_column_with_hash_bucket(
173     "education", hash_bucket_size=1000)
```

#### 4.9.4 特征交叉

因为线性模型赋值独立的权重给分开的特征，它们不嫩改写东西相对重要的指定特征值的结合。如果你有一个特征'favorite\_sport' 和特征'home\_city' 然后你想尝试预测一个是否喜欢船红色的衣服，你的线性模型将不能从喜欢穿红色衣服的圣路易斯的棒球粉丝中学习。你可以通过创建一个新的特征'favorite\_sport\_x\_home\_city' 得到这个限制。这些特征的值对一个人仅仅链接两个源特征'baseball\_x\_stlouis'，例如这个结合的特征被称为特征交叉，crossed\_column() 方法使得建立特征交叉很容易

```
174 sport_x_city = tf.feature_column.crossed_column(
175     ["sport", "city"], hash_bucket_size=int(1e4))
```

连续的列，你可以像这样指定一个连续的特征：

```
176 age = tf.feature_column.numeric_column("age")
```

尽管作为一个简单的实数，连续特征经常能直接输入给模型，TensorFlow 对于排序这列提供了有用的转化。

#### 4.9.5 Bucketization

Bucketization 转化一个连续的列为绝对列。这个转换让你在交叉特征中用连续的特征，或者了解那里指定值的范围特别重要，Bucketization 分隔可能值的范围为字范围称为 buckets。

```
177 age_buckets = tf.feature_column.bucketized_column(
178     age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
```

bucket 掉入那个区域变成这个值的绝对标签。

### 输入函数

FeatureColumn 为你的模型提供了一个指定的输入数据。标志着着如何代表和转化数据。但是它们本身不提供数据，你通过输入函数提供数据，输入函数必须返回一个 tensor 字典，每个键是 FeatureColumn 的名字。每个键的值是一个对于所有数据实例是包含特征值的 tensor。查看[Building Input Functions with tf.estimator](#)获取更多的关于输入函数的见解，在[linear models tutorial code](#)中的 input\_fn 是实现输入函数的一个例子。在输入函数被传递给 train() 和 evaluate() 调用初始训练和测试正如下一章描述的。

### 4.9.6 线性 estimator

TensorFlow 的 estimator 给训练和分类模型提供了一个独一无二的训练评估工具。它们考虑详细的训练和评估训练允许用户集中注意力在模型输入和架构上。

为了建立一个线性 estimator，你可以用 `tf.estimator.LinearClassifier` estimator 或者 `tf.estimator.LinearRegressor` estimator 分别建立分类和回归模型。

创建 tensorflow estimator 和运行 estimator:

- estimator 实例，对于两个线性 estimator 类，你传递一个 FeatureColumn 列表给构造器。
- 调用 estimator 的 `train()` 方法训练它
- 调用 estimator 的 `evaluate()` 方法看看它如何工作

例如:

```

179 e = tf.estimator.LinearClassifier(
180     feature_columns=[
181         native_country, education, occupation, workclass, marital_status,
182         race, age_buckets, education_x_occupation,
183         age_buckets_x_race_x_occupation],
184         model_dir=YOUR_MODEL_DIRECTORY)
185 e.train(input_fn=input_fn_train, steps=200)
186 # Evaluate for one step (one pass through the test data).
187 results = e.evaluate(input_fn=input_fn_test)
188
189 # Print the stats for the evaluation.
190 for key in sorted(results):
191     print("%s: %s" % (key, results[key]))

```

### 4.9.7 广泛深入的学习

`tf.estimator` API 提供一个 estimator 类让你结合训练一个模型和深度神经网络。这个出色的方法结合线性模型的能力存储神经网络泛化的能力关键特征。用 `tf.estimator.DNNLinearCombinedClassifier` 创建一个广而深的模型:

```

192 e = tf.estimator.DNNLinearCombinedClassifier(
193     model_dir=YOUR_MODEL_DIR,
194     linear_feature_columns=wide_columns,
195     dnn_feature_columns=deep_columns,
196     dnn_hidden_units=[100, 50])

```

更多信息请查看[Wide and Deep Learning tutorial](#)。

## 4.10 tensorflow 线性模型导航

在这个导航中我们将用 TensorFlow 中的 tf.estimator API 解决 2 分类问题: 给关于一个人的年龄, 性别, 教育, 职位统计数据, 我们将预测这个人是够得到超过 5000 美元的年薪(目标标签)。我们用逻辑回归模型, 给一个信息我们的模型将输出一个 0 到 1 的数用来解释收入超过 50000 美元的概率。

### 建立

尝试下面代码:

- 如果你没有安装 TensorFlow 先<https://www.tensorflow.org/install/index>
- 下载[https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/examples/learn/wide\\_n\\_deep\\_tutorial.py](https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/examples/learn/wide_n_deep_tutorial.py)
- 安装 pandas 数据分析库。tf.estimator 不要求 pandas, 但是支持它, 导航用 pandas, 安装 pandas:
  - 安装 pip

```

197 # Ubuntu/Linux 64-bit
198 $ sudo apt-get install python-pip python-dev
199
200 # Mac OS X
201 $ sudo easy_install pip
202 $ sudo easy_install --upgrade six
203

```

  - 安装 pandas

```

204 sudo pip install pandas
205

```

- 指定导航带吧训练导航表述的线性模型:

```

206 python wide_n_deep_tutorial.py --model_type=wide
207

```

### 4.10.1 读取调查数据

我们将要使用的数据集是Census Income Dataset, 你讲需要手动下载训练集和测试集或者用下面的代码:

```

208 import tempfile
209 import urllib
210 train_file = tempfile.NamedTemporaryFile()
211 test_file = tempfile.NamedTemporaryFile()
212 urllib.urlretrieve("https://archive.ics.uci.edu/ml/machine-learning-databases/
    adult/adult.data", train_file.name)
213 urllib.urlretrieve("https://archive.ics.uci.edu/ml/machine-learning-databases/
    adult/adult.test", test_file.name)

```

当 csv 文件下载后我们读入 pandas DataFrame。

```

214 import pandas as pd
215 CSV_COLUMNS = [
216     "age", "workclass", "fnlwgt", "education", "education_num",
217     "marital_status", "occupation", "relationship", "race", "gender",
218     "capital_gain", "capital_loss", "hours_per_week", "native_country",
219     "income_bracket"]
220 df_train = pd.read_csv(train_file.name, names=CSV_COLUMNS, skipinitialspace=True
    )
221 df_test = pd.read_csv(test_file.name, names=CSV_COLUMNS, skipinitialspace=True,
    skiprows=1)

```

因为是 2 分类问题，我们构造一个标签 label，label 的值为 1 表示收入超过 50K，否则为 0。

```

222 train_labels = (df_train["income_bracket"].apply(lambda x: ">50K" in x)).astype(
    int)
223 test_labels = (df_test["income_bracket"].apply(lambda x: ">50K" in x)).astype(
    int)

```

下一步我们查看 dataframe 看那一列我们可以预测目标标签。这列可以有两种类型-绝对和连续列：

- 如果它的值仅仅是有限数据集中的一个类别这一列称为绝对的。例如人的本土(U.S,India,Jpan,etc) 或者教育层度（高中，大学等等）是绝对列。
- 如果它的值可以是任何连续的范围这一列称为连续的，例如一个人的资本所得（例如 \$14084）是一个连续的列。

下面是一个连续的收入数据的调查

列 的 名字	类型	描述
age	Continuous	年龄的名字
workclass	Categorical	雇主的类型 (政府, 军队, 私人等等)
fnlwgt	Continuous	人们相信观察表示 (相同的权重) 的数量, 这个变量将被使用
education	Categorical	对个人获得最高的学历
education\_num	Continuous	数字形式的学历
marital\_status	Categorical	婚姻状况
occupation	Categorical	个人职位
relationship	Categorical	妻子, 小孩, 丈夫, 没有家庭, 其它相关等等或者没有结婚
race	Categorical	白人, 亚洲人, 爱斯基摩人, 其它, 黑人
gender	Categorical	男, 女
capital\_gain	Continuous	资本收入记录
capital\_loss	Continuous	资本损失记录
hours\_per\_week	Continuous	每周工作小时数
native\_country	Categorical	来自的国家
income	Categorical	“>50K” 或者 “<=50K”, 意味着一个人是否能赚超过 50K\$

#### 4.10.2 转换数据为 Tensors

当建立一个 tf.estimator 模型的时候, 输入数据通过输入 Builder 函数指定。Builder 函数不被调用知道传入方法 train 或者 evaluate。这个函数的目的是构造代表 tf.Tensor 或者 tf.SparseTensor 的输入数据, 输入 Builder 函数返回下面对:

- feature\_cols: 特征列命名为 Tensors 或者 SparseTensors 的字典。
- label: 包含标签列的 Tensor

feature\_cols 的 keys 用来在下一章节构造列。因为我们想在不同的数据上调用 train 和 evaluate 方法, 我们定义基于给定数据返回一个输入函数。注意返回输入函数知道构建 TensorFlow 图的时候被调用, 而不是当图运行的时候。输入数据返回的是一个代表 TensorFlow 基本单元的 Tensor。

我们用 tf.estimator.inputs.pandas\_input\_fn 方法从 pandas DataFrame 创建一个输入函数, 每个 train dataframe 或者 test dataframe 连续列将转化为一个 Tensor (是代表稠密数据的好形式), 对于绝对数据我们必须使用 SparseTensor 表示。这种数据格式有利于

代表稀疏数据，另一个更高级的代表输入数据将是构建一个Inputs and Reader代表一个文件或者数据源，通过文件迭代在TensorFlow图中运行。

```

10 def input_fn(data_file, num_epochs, shuffle):
11     """Input builder function."""
12     df_data = pd.read_csv(
13         tf.gfile.Open(data_file),
14         names=CSV_COLUMNS,
15         skipinitialspace=True,
16         engine="python",
17         skiprows=1)
18     # remove NaN elements
19     df_data = df_data.dropna(how="any", axis=0)
20     labels = df_data["income_bracket"].apply(lambda x: >50K in x).astype(int)
21     return tf.estimator.inputs.pandas_input_fn(
22         x=df_data,
23         y=labels,
24         batch_size=100,
25         num_epochs=num_epochs,
26         shuffle=shuffle,
27         num_threads=5)

```

## 为模型选择工程特征

选择加工正确的特征列是一个高校学习模型的关键，一个 feature column 可以是原始 dataframe 中的原始列（我们称之为特征列）或者任何新的基于一列或者多列变形的 derived feature。基本的 feature column 是一个原始数据和衍生数据的抽象概念来预测目标标签。

### 基于绝对特征列

为了定义基于绝对特征的特征列，我们用 `tf.feature_column` API 创建一个 CategoricalColumn，如归哦你知道可能的列的特征有一些值，你可以用 `categorical_column_width_vocabulary_list`。每个列表中的 key 将得到赋值从 0 开始自动增加 ID。例如对于性别列我们可以复制特征字符串“Female”为整数 ID 0 “Male”赋值为 1。

```

28 gender = tf.feature_column.categorical_column_with_vocabulary_list(
29     "gender", ["Female", "Male"])

```

如果你预先不知道可能值做怎么办？没问题你可以用 `categorical_column_width_hash_bucket` 代替：

```

224 occupation = tf.feature_column.categorical_column_with_hash_bucket(
225     "occupation", hash_bucket_size=1000)

```

ID	feature
...	
9	"Machine-op-inspct"
...	
103	"Farming-fishing"
...	
375	"Protective-serv"
...	

当我们在训练中遇到它们时在列中的每个值每散列到一个整数 ID:

无论我们选择哪种方法定义一个 SparseTensor。每个特征字符串将通过查找一个固定的映射或者散列特征字符串将被映射为一个整数 ID。注意散列冲突可能会发生,但是对于模型的质量影响不大。在 hood 下, LinearModel 类掉膘管理映射和为每个特征 ID 创建 tf.Variable 存储模型参数。模型参数将通过模型训练过程被学习到。我们将在定义的其它绝对特征做类似的模型训练:

```

226 education = tf.feature_column.categorical_column_with_vocabulary_list(
227     "education", [
228         "Bachelors", "HS-grad", "11th", "Masters", "9th",
229         "Some-college", "Assoc-acdm", "Assoc-voc", "7th-8th",
230         "Doctorate", "Prof-school", "5th-6th", "10th", "1st-4th",
231         "Preschool", "12th"
232     ])
233 marital_status = tf.feature_column.categorical_column_with_vocabulary_list(
234     "marital_status", [
235         "Married-civ-spouse", "Divorced", "Married-spouse-absent",
236         "Never-married", "Separated", "Married-AF-spouse", "Widowed"
237     ])
238 relationship = tf.feature_column.categorical_column_with_vocabulary_list(
239     "relationship", [
240         "Husband", "Not-in-family", "Wife", "Own-child", "Unmarried",
241         "Other-relative"
242     ])
243 workclass = tf.feature_column.categorical_column_with_vocabulary_list(
244     "workclass", [
245         "Self-emp-not-inc", "Private", "State-gov", "Federal-gov",
246         "Local-gov", "?", "Self-emp-inc", "Without-pay", "Never-worked"
247     ])
248 native_country = tf.feature_column.categorical_column_with_hash_bucket(
249     "native_country", hash_bucket_size=1000)

```

## 基于连续特征列

如果你用在模型用使用类似的你可以为每个连续的特征列定义一个 NumericColumn:

```
250 age = tf.feature_column.numeric_column("age")
251 education_num = tf.feature_column.numeric_column("education_num")
252 capital_gain = tf.feature_column.numeric_column("capital_gain")
253 capital_loss = tf.feature_column.numeric_column("capital_loss")
254 hours_per_week = tf.feature_column.numeric_column("hours_per_week")
```

## 通过 Bucketization 创建连续特征

在连续特征和非线性标签之间有一些关系。正如假设的例子，一个人的收入也许在它事业的早期随着年龄增长，然后在这些点的增长也许会变慢，最后退休后收入下降。在这个场景下，用原始的 age 作为真实值特征列也需要不是一个好的选择因为模型仅仅能从下面三种情况下学习：

1. 收入随着年龄的增加而增加（正相关）
2. 收入随着年龄的增加减少（负相关）
3. 收入随着年龄增加不变（不相关）

如果我们想单独了解收入和年龄组适当的相关性，我们可以利用 bucketization。bucketization 是分割整个连续特征成一系列连续的 bins/buckets 的处理，依赖值调入那个 bucket 转化原始数值特征成一个 bucket ID（正如一个绝对特征）。因此我们可以在 age 上定义一个 bucketized\_column:

```
255 age_buckets = tf.feature_column.bucketized_column(
256     age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
```

这里的 boundaries 是一个 bucket 范围列表，这个例子中有 10 个范围导致 11 个年龄组 buckets（从 17 岁即以下，18-24,2-29,...,65 及以上）

### 4.10.3 多列交叉的交叉列

用每个基础特征列分隔也许不能完全解释数据。例如，教育和标签 (earning>50000\$) 相关也许不同的职位不同。因此，如果我们仅仅针对 education="Bachelors" 和 education="Masters" 学习三个的模型权重，我们将不能捕获单个的职业教育组合（区别 education="Bachelors" 和 occupation="Exec-managerial" 与 education="Bachelors" 和 occupation="Craft-repair"）。为了学习不同的特征组合我们可以增加 crossed feature columns 到模型上

```
257 education_x_occupation = tf.feature_column.crossed_column(
258     ["education", "occupation"], hash_bucket_size=1000)
```

我们也可以创建一个超过两列的 CrossedColumn。每个组成的列既可以是一个基础的特征列 (SparseColumn), 耦合 bucketized 真实值特征列 (BucketizedColumn) 或者说是 CrossedColumn。这里是例子:

```
259 age_buckets_x_education_x_occupation = tf.feature_column.crossed_column(
260     [age_buckets, "education", "occupation"], hash_bucket_size=1000)
```

## 定义一个路基回归模型

在处理输入数据和定义所有的特征列后，我们先将准备将它们放在一起建立一个逻辑回归模型。在之前的章节中我么已经看到一些基础和衍生特征列的类型，包括:

- CategoricalColumn
- NumericaColumn
- BucketizedColumn
- CrossedColumn

所有的这些抽象的 FeatureColumn 类的子类可以被添加到模型 feature\_columns 里:

```
261 base_columns = [
262     gender, native_country, education, occupation, workclass, relationship,
263     age_buckets,
264 ]
265 crossed_columns = [
266     tf.feature_column.crossed_column(
267         ["education", "occupation"], hash_bucket_size=1000),
268     tf.feature_column.crossed_column(
269         [age_buckets, "education", "occupation"], hash_bucket_size=1000),
270     tf.feature_column.crossed_column(
271         ["native_country", "occupation"], hash_bucket_size=1000)
272 ]
273
274 model_dir = tempfile.mkdtemp()
275 m = tf.estimator.LinearClassifier(
276     model_dir=model_dir, feature_columns=base_columns + crossed_columns)
```

模型也自动学习控制预测使得没有观察任何特征的偏移，学到的模型文件将被存储在 model\_dir。

## 训练评估你的模型

在添加一个特征到模型后，下周乃让我么你查看如何训练模型，用 tf.estimator API 训练一个模型仅仅是一个简单的事

```
277 # set num_epochs to None to get infinite stream of data.
278 m.train(
279     input_fn=input_fn(train_file_name, num_epochs=None, shuffle=True),
280     steps=train_steps)
```

在模型被训练后，你可以评估我们的模型在预测数据时候如何好

```
281 results = m.evaluate(
282     input_fn=input_fn(test_file_name, num_epochs=1, shuffle=False),
283     steps=None)
284 print("model directory = %s" % model_dir)
285 for key in sorted(results):
286     print("%s: %s" % (key, results[key]))
```

输出的第一行像 accuracy: 0.83557522，意味着精确度大约 83.6%，如果你想看更好的结构尝试更多特征和转换。如果你想看从头到尾的实现，你可以下载[示例代码](#)设置 model\_type flag 为 wide。

## 添加正则化防止过拟合

正则化是阻止过拟合的一种技术。当你的数据在训练集上表现很好但是在没有见过的测试机上表现很糟糕时郭力赫发生。股拟合通常当模型很复杂时发生，像有一些关于观测训练数据的参数。正则化允许你控制模型的复杂度使得模型在没有见过的数据上能更加的泛化。

在线性模型库，你可以添加 L1 和 L2 证这话到模型上：

```
287 m = tf.estimator.LinearClassifier(
288     model_dir=model_dir, feature_columns=base_columns + crossed_columns,
289     optimizer=tf.train.FtrlOptimizer(
290         learning_rate=0.1,
291         l1_regularization_strength=1.0,
292         l2_regularization_strength=1.0),
293     model_dir=model_dir)
```

L1 和 L2 正则化一个重要的区别是 L1 正则化尝试使得模型的权重保持在 0，创建系数模型，由于 L2 证这话也尝试使得模型权重更接近于 0 但是不需要 0. 因此如果如果你增强 L1 正则化的强度你讲有一个小的模型尺寸因为一些模型的权重变为了 0. 因此，特征空间和大却不稀疏时和有一些资源限制阻止你离开一个太大的模型时需要，事实上你讲尝试多种 L1, L2 正则化组合强度找到最好的参数最好的控制过拟合给你一个想要的模型尺寸。

#### 4.10.4 逻辑回归如何工作

最后让我们花几分钟了解一下实际的逻辑回归模型看起来像什么。我们用  $Y$  表示标签，观察值  $\mathbf{x} = [x_1, x_2, \dots, x_d]$  我们定义如果个人所得超过 50K\$,  $Y=1$ , 否则  $Y=0$ 。

$$P(Y=1|\mathbf{x}) = \frac{1}{1 + \exp(-(\mathbf{W}^T \mathbf{x} + b))}$$

这里的  $\mathbf{w} = [w_1, w_2, \dots, w_d]$  是模型特征  $\mathbf{x} = [x_1, x_2, \dots, x_d]$  的权重  $b$  是称为偏重的常数。方程由两部分组成，线性模型和回归模型。

- 线性模型：首先我们看到  $\mathbf{w}^T \mathbf{x} + b = b + w_1 x_1 + \dots + x_d$ , 这里输出是输入特征的一个线性函数，偏置  $b$  是没有任何特征输入时的预测。模型的权重  $w_i$  反映了特征  $x_i$  是如何和特征相关，如果  $x_i$  与正相关的标签，权重  $w_i$  增加  $P(Y=1|\mathbf{x})$  将接近于 1，对应负相关时权重  $w_i$  增加  $P(Y=1|\mathbf{x})$  将接近于 0.
- 逻辑函数：你可以看到一个逻辑函数  $S(t) = \frac{1}{1+\exp(-t)}$  应用的线性模型，逻辑函数用于转换输出线性模型  $\mathbf{w}^T \mathbf{x} + b$  从任意实数值到  $[0,1]$  之间，这个值可以被解释为概率。

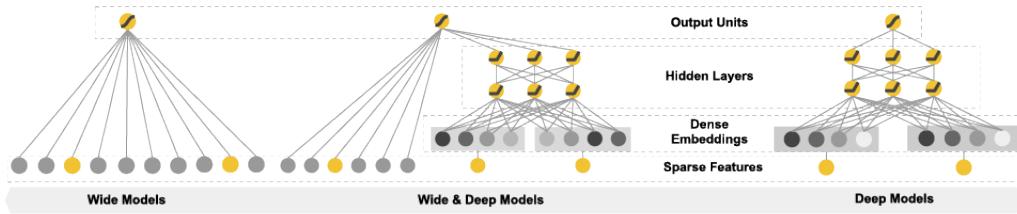
模型训练是一个优化问题：目标是找到模型的权重最小化训练数据上定义的损失函数，像逻辑回归的逻辑损失。损失函数描述了预测值和真实值的差别。如果预测接近指示标签，损失将很小。如果预测值离标签很远，损失将很大。

#### 深入学习

如果你想学习更多，查看下一章[Wide & Deep Learning Tutorial](#) 我们将教你如何用 `tf.estimator` 结合线性模型的力量和深度神经网络。

## 4.11 TensorFlow 广泛深入的学习

在先前的 TensorFlow 线性模型教程中，我们训练了一个逻辑回归模型，以使用[人口普查收入数据集](#)预测个人年收入超过 5 万美元的概率。TensorFlow 也适用于训练深层神经网络，您可能会考虑选择哪一个 - 那么为什么不这两个呢？是否可以将两者的优势结合在一个模型中？在本教程中，我们将介绍如何使用 `TF.Learn API` 联合训练广泛的线性模型和深入的前馈神经网络。这种方法结合了记忆和泛化的优势。它对于具有稀疏输入特征的通用大规模回归和分类问题（例如，具有大量可能特征值的分类特征）是有用的。如果您有兴趣了解 Wide & Deep Learning 如何运作，请查看我们的[研究论文](#)。



上图显示了宽模型（具有稀疏特征和变换的逻辑回归），深度模型（具有嵌入层和多个隐藏层的前馈神经网络）的比较，以及宽&深度模型（两者的联合训练）。在高层次上，只需 3 个步骤即可使用 TF.Learn API 来配置宽，深或宽和深的模型：

1. 选择广泛部分的功能：选择要使用的稀疏基础列和交叉列。
2. 选择深部分的要素：选择连续列，每个分类列的嵌入维度和隐藏的图层大小。
3. 把它们全部放在一个 Wide & Deep 模型 (DNNLinearCombinedClassifier) 中。就是这样！我们来看一个简单的例子。

#### 4.11.1 建立

要尝试本教程的代码：

1. 如果还没有[安装 TensorFlow](#)。
2. 下载教程[代码](#)。
3. 安装熊猫数据分析库。tf.learn 不需要大熊猫，但它支持它，本教程使用大熊猫。安装 pandas：

(a) 获取 pip:

```

1      # Ubuntu/Linux 64-bit
2          sudo apt-get install python-pip python-dev
3
4      # Mac OS X
5          sudo easy_install pip
6          sudo easy_install --upgrade six
7

```

(b) 使用 pip 安装 pandas: sudo pip install pandas

如果您在安装 pandas 麻烦, 请查阅[pandas 网站](#)。

4. 使用以下命令执行教程代码来训练本教程中描述的线性模型:python wide\_n\_deep\_tutorial.py --model\_type=wide\_n\_deep

请继续阅读, 了解此代码如何构建其线性模型。

### 4.11.2 定义基本特征列

首先, 我们来定义我们将使用的基本分类和连续特征列。这些基列将是模型的宽部分和深部使用的构件块。

```

1 import tensorflow as tf
2
3 gender = tf.feature_column.categorical_column_with_vocabulary_list(
4     "gender", ["Female", "Male"])
5 education = tf.feature_column.categorical_column_with_vocabulary_list(
6     "education", [
7         "Bachelors", "HS-grad", "11th", "Masters", "9th",
8         "Some-college", "Assoc-acdm", "Assoc-voc", "7th-8th",
9         "Doctorate", "Prof-school", "5th-6th", "10th", "1st-4th",
10        "Preschool", "12th"
11    ])
12 tf.feature_column.categorical_column_with_vocabulary_list(
13     "marital_status", [
14         "Married-civ-spouse", "Divorced", "Married-spouse-absent",
15         "Never-married", "Separated", "Married-AF-spouse", "Widowed"
16    ])
17 relationship = tf.feature_column.categorical_column_with_vocabulary_list(
18     "relationship", [
19         "Husband", "Not-in-family", "Wife", "Own-child", "Unmarried",
20         "Other-relative"
21    ])

```

```

22 workclass = tf.feature_column.categorical_column_with_vocabulary_list(
23     "workclass", [
24         "Self-emp-not-inc", "Private", "State-gov", "Federal-gov",
25         "Local-gov", "?", "Self-emp-inc", "Without-pay", "Never-worked"
26     ])
27
28 # To show an example of hashing:
29 occupation = tf.feature_column.categorical_column_with_hash_bucket(
30     "occupation", hash_bucket_size=1000)
31 native_country = tf.feature_column.categorical_column_with_hash_bucket(
32     "native_country", hash_bucket_size=1000)
33
34 # Continuous base columns.
35 age = tf.feature_column.numeric_column("age")
36 education_num = tf.feature_column.numeric_column("education_num")
37 capital_gain = tf.feature_column.numeric_column("capital_gain")
38 capital_loss = tf.feature_column.numeric_column("capital_loss")
39 hours_per_week = tf.feature_column.numeric_column("hours_per_week")
40
41 # Transformations.
42 age_buckets = tf.feature_column.bucketized_column(
43     age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])

```

### 4.11.3 宽模型：具有交叉特征列的线性模型

宽模型是一个线性模型，具有广泛的稀疏和交叉特征列：

```

1 base_columns = [
2     gender, native_country, education, occupation, workclass, relationship,
3     age_buckets,
4 ]
5
6 crossed_columns = [
7     tf.feature_column.crossed_column(
8         ["education", "occupation"], hash_bucket_size=1000),
9     tf.feature_column.crossed_column(
10        [age_buckets, "education", "occupation"], hash_bucket_size=1000),
11     tf.feature_column.crossed_column(
12        ["native_country", "occupation"], hash_bucket_size=1000)
13 ]

```

具有交叉特征列的宽模型可以有效地记住功能之间的稀疏交互。也就是说，交叉特征列的一个限制是它们不会推广到没有出现在训练数据中的特征组合。让我们用嵌入式添加一个深层次的模型来解决这个问题。

#### 4.11.4 深层模型：嵌入式神经网络

深层模型是前馈神经网络，如前图所示。每个稀疏的高维度分类特征首先被转换成低维和密集的实值向量，通常被称为嵌入向量。这些低维密集嵌入矢量与连续特征相连，然后在正向传递中嵌入神经网络的隐层。嵌入值被随机初始化，并与所有其它模型参数一起训练，以最小化训练损失。如果您有兴趣了解有关嵌入的更多信息，请参阅维基百科上的 TensorFlow 关于[单词向量表示](#)或[Word Embedding](#)的教程。另一个表示绝对列输入神经网络的方法是通过 mult-hot 表达。对于仅仅有一些可能的绝对列是合适的。”Male” 表示 [1,0], “Female” 表示 [0,1]. 这是一个固定的表达，embedding 能更灵活的计算。我们将使用 embedding\_column 并将它们连接到连续列来配置分类列的嵌入，我们用 indicator\_column 创建 mult-hot 代表一些绝对的列

```

1 deep_columns = [
2     tf.feature_column.indicator_column(workclass),
3     tf.feature_column.indicator_column(education),
4     tf.feature_column.indicator_column(gender),
5     tf.feature_column.indicator_column(relationship),
6     # To show an example of embedding
7     tf.feature_column.embedding_column(native_country, dimension=8),
8     tf.feature_column.embedding_column(occupation, dimension=8),
9     age,
10    education_num,
11    capital_gain,
12    capital_loss,
13    hours_per_week,
14 ]

```

较高的 dimension 嵌入的是，更多的自由度的模式将要学习的功能表示。为了简单起见，我们在这里为所有功能列设置尺寸为 8。在经验上，对维度数量的更明智的决定是从一个价值的顺序开始 日志 2 ( $\log_2$ ) 要么  $\sqrt{k}$ , 哪里  $k$  是特征列中的唯一特征的数量  $k$  是一个小常数（通常小于 10）。通过密集嵌入，深层模型可以更好地概括，并对以前在培训数据中看不到的特征对进行预测。然而，当两个特征列之间的底层交互矩阵稀疏和高等级时，很难学习特征列的有效低维表示。在这种情况下，大多数特征对之间的交互应该为零，除了少数几个，但密集的嵌入将导致所有特征对的非零预测，因此可能会过度泛化。另一方面，具有交叉特征的线性模型可以用更少的模型参数有效地记住这些“异常规则”。现在，我们来看看如何联合训练广泛而深入的模式，让它们相互补充优势和劣势。

#### 4.11.5 将宽和深度模型结合为一体

通过将其最终输出对数几率作为预测结合起来，将广泛的模型和深度模型相结合，然后将预测提供给物流损失函数。所有的图形定义和变量分配已经在你的引擎盖下处理，所

以你只需要创建一个 DNNLinearCombinedClassifier:

```

1 import tempfile
2 model_dir = tempfile.mkdtemp()
3 m = tf.contrib.learn.DNNLinearCombinedClassifier(
4     model_dir=model_dir,
5     linear_feature_columns=wide_columns,
6     dnn_feature_columns=deep_columns,
7     dnn_hidden_units=[100, 50])

```

#### 4.11.6 训练和评估模型

在我们训练模型之前，让我们一起阅读普查数据集，就像我们在 TensorFlow 线性模型教程中所做的一样。输入数据处理代码再次提供给您方便：

```

1 import pandas as pd
2 import urllib
3
4 # Define the column names for the data sets.
5 CSV_COLUMNS = [
6     "age", "workclass", "fnlwgt", "education", "education_num",
7     "marital_status", "occupation", "relationship", "race", "gender",
8     "capital_gain", "capital_loss", "hours_per_week", "native_country",
9     "income_bracket"
10]
11
12 def maybe_download(train_data, test_data):
13     """Maybe downloads training data and returns train and test file names."""
14     if train_data:
15         train_file_name = train_data
16     else:
17         train_file = tempfile.NamedTemporaryFile(delete=False)
18         urllib.request.urlretrieve(
19             "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.
20             data",
21             train_file.name) # pylint: disable=line-too-long
22         train_file_name = train_file.name
23         train_file.close()
24         print("Training data is downloaded to %s" % train_file_name)
25
26     if test_data:
27         test_file_name = test_data
28     else:
29         test_file = tempfile.NamedTemporaryFile(delete=False)

```

```

29     urllib.request.urlretrieve(
30         "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.
31             test",
32             test_file.name) # pylint: disable=line-too-long
33     test_file_name = test_file.name
34     test_file.close()
35     print("Test data is downloaded to %s" % test_file_name)
36
37
38 def input_fn(data_file, num_epochs, shuffle):
39     """Input builder function."""
40     df_data = pd.read_csv(
41         tf.gfile.Open(data_file),
42         names=CSV_COLUMNS,
43         skipinitialspace=True,
44         engine="python",
45         skiprows=1)
46     # remove NaN elements
47     df_data = df_data.dropna(how="any", axis=0)
48     labels = df_data[ "income_bracket" ].apply(lambda x: ">50K" in x).astype( int )
49     return tf.estimator.inputs.pandas_input_fn(
50         x=df_data,
51         y=labels,
52         batch_size=100,
53         num_epochs=num_epochs,
54         shuffle=shuffle,
55         num_threads=5)

```

阅读数据后，您可以对模型进行培训和评估：

```

1 m.fit(input_fn=train_input_fn, steps=200)
2 results = m.evaluate(input_fn=eval_input_fn, steps=1)
3 for key in sorted(results):
4     print("%s: %s" % (key, results[key]))

```

输出的第一行应该是这样的 accuracy: 0.84429705。我们可以看到，使用宽和深度模型，使用广泛线性模型，约 83.6% 的准确度提高到约 84.4%。如果您想看到一个有效的端对端示例，您可以下载我们的示例代码。请注意，本教程只是一个小型数据集的简单示例，可让您熟悉 API。如果您在具有许多可能的特征值的许多稀疏特征列的大型数据集上尝试，则 Wide & Deep Learning 将更加强大。再次，请随时查看我们的研究论文，了解更多关于如何在现实世界的大型机器学习问题中应用广泛和深度学习的想法。

## 4.12 移动平台

TensorFlow 对于移动平台被设计为一个好的深度学习解决方案。当前我们已经有两个解决方案用于部署机器学习应用到移动设备和嵌入式设备:[TensorFlow for Mobile](#)和[TensorFlow Lite](#)。

### 4.12.1 TensorFlow LiteVS TensorFlow Mobile

下面是两者的一些不同之处:

- TensorFlow Lite 是 TensorFlow Mobile 的一个进化版本。在多属性框下，用 TensorFlow Lite 开发的 app 二进制文件更小，依赖更少，性能更强。
- TensorFlow Lite 是在开发者预览中，因此不是所有情况都能覆盖。我们希望你用 TensorFlow Mobile 覆盖产品。
- TensorFlow Lite 仅仅支持有限的操作，因此不是所有的模型都能在上面运行的很好。TensorFlow Mobile 对函数有一个更完整的支持。

TensorFlow Lite 为移动平台提供了更好的性能和更小的文件尺寸如果在它们的平台上一些硬件加速功能可以使用的话。另外，它有更少的依赖以至于它可能被构建运行在更简单的，资源有限的设备上。TensorFlow Lite 可允许通过[Neural Networks API](#)加速。

TensorFlow Lite 当前已经覆盖了有限的一些操作。尽管默认 TensorFlow Mobile 仅仅支持有限的操作，原则上如果你用 TensorFlow 的任意操作，它能被自定义构建 kernel。这样用不支持 TensorFlow Lite 的情况将继续支持 TensorFlow Mobile。正如 TensorFlow Lite 设计的，它将添加额外的操作，将被很容易决定。

### 4.12.2 介绍 TensorFlow Lite

TensorFlow Lite 是针对移动设备和嵌入式设备的 TensorFlow 的轻量级的解决方案。它以很低的代价和小的二进制文件尺寸在设备上进行机器学习推理。TensorFlow Lite 也支持用[Android Neural Networks API](#)进行加速。

TensorFlow Lite 用一些想优化移动 app 核心，预先融合激活，量化内核的技术获取更低的消耗获得更小更快的模型。大多数的 TensorFlow Lite 文档在[Github](#)上。

### 4.12.3 TensorFlow Lite 包含什么？

TensorFlow Lite 包含一系列核心操作，包括针对移动平台调整的量化和浮点。它们通过预先融合激活和偏置加强性能和量化精度。另外，TensorFlow Lite 也支持在模型中使用自定义的操作。

## 4.13 介绍 TensorFlow Mobile

TensorFlow 被设计成为在像 Android 和 iOS 的移动平台上一个好的深度学习解决方案。移动向导应该帮助你理解机器学习可以在移动平台上工作以及如何高效整合 TensorFlow 到你的移动 app 中。

### 4.13.1 关于这个向导

这个向导针对已经有一个能在桌面环境下工作的模型想整合模型进入移动应用中，不能用 TensorFlow Lite，下面是你将面对的主要挑战：

- 明白如何使用 TensorFlow for mobile
- 为你的平台构建 TensorFlow
- 整合 TensorFlow 库进你的应用
- 为你的移动部署准备你的模型文件
- 优化时延，RAM 使用，模型文件大小，二进制文件大小

### 4.13.2 常用的机器学习情景

**为什么在移动设备上运行 TensorFlow？** 通常的深度学习结合数据中心和大型的高性能 GPU 集群。然而，通过网络连接发送设备数据这可能很昂贵耗费时间。运行在移动设备上是可能的当你继续不可能必须等待网络处理传送每个交互。下面是在设备上运行深度学习的常见情景。

### 4.13.3 语音识别

一些有趣的应用可能构建在一个语义驱动的接口上，这些请求在设备上处理。很多时候长时间不给任何指令，持续和远程服务器通信将浪费带宽，因为很多情况下很安静或者有背景噪声。为了解决这个问题一个常见的通常是运行一个小的神经网络在设备上[listening out for a particular keyword](#)当关键字被发现，一些剩下的场景可能是如果需要更多的计算力则发送给服务器进一步处理。

### 4.13.4 图像识别

对于移动 app 理解摄像头场景是很有用的。如果你的用户拍摄了照片，识别它们可能帮助你的相机 app 更好的过滤或者标记相片以至于它们能轻松地找到。这对于嵌入式应用来说也是很重要的，因此你可以用图像传感器检测一些感兴趣的条件，是否动物在野生环

境下或者[报告你的训练运行多晚](#) TensorFlow 结合一些预先训练好的目标识别的模型，它们可以运行在你的移动设备上。你可以尝试[Tensorflow for Poets](#) 和[Tensorflow for Poets 2: Optimize for Mobile](#) 代码实验室，查看如何获取已经训练好的运行非常快，轻量级的模型训练教它识别特殊物体，优化它在手机端的运行。

### 4.13.5 对象定位

有时候知道对象在图像中的位置和对象是什么一样重要。一些参数使用情景可能从一个移动 app 获益，比如说指导用户到正确的组件提供帮助修复它们的无线网络或者在一些特殊的场景提供信息覆盖其上。嵌入式应用经常需要计数传给它们的对象，是否宠物在庄稼地里或者人，汽车，自行车将通过街道信号灯。TensorFlow 提供一个预先训练好的模型描绘在图像中检测到的人一个框，结合追踪代码实时跟踪它们。最终对于你尝试统计多少对象被实时呈现是很重要的，因此放一个新的对象进入或者离开场景时它给你一个好的想法。对于 Android 设备我们给了一个可用的代码在[Github](#)，更多以[常用的对象检测模型](#)也可用

### 4.13.6 手势识别

不论是用手或者其它的姿势控制应用，从图像中识别或者分析加速度传感器数据是很有用的。创建这些模型超过了这个向导的内容，但是 TensorFlow 能高效的部署它们。

### 4.13.7 光学字符识别

Google 翻译的实时相机查看是一个设备检测文字的交互高效的好例子。在图像中识别文字有一些步骤，首先识别文字呈现的区域，就是目标定位，可以使用类似的技术解决。当你有文字区域后你将需要解释它为字符，然后用语言模型猜测它们表达的是什么意思。最简单的评估文字表达的意思是分割文本行为单个的文字然后使用简单的神经网络框住每个字符。你可以用 MNIST 模型（TensorFlow 导航）获得一个好的结果，你也可以像一个更高级的方案解决输入。一个高级的使用是用 LSTM 模型处理一行文本，模型自己处理片段为不同的字符。

### 4.13.8 翻译

即使在没有网络的情况下准确的翻译一种语言到另一种语言是很重要的使用场景。深度网络在这些任务上很高效，你能找到一些不同的文学模型描述。经常 seq2seq 循环模型能运行单个图处理整个翻译，结合需要运行分割的解析场景。

### 4.13.9 文本分类

如果你想在用户输入和阅读的基础上给出相关的建议，理解文本的意思将变得很有用。文本分类是一个包含了从语义分析到主题发现。你可能有自己想应用的策略或者标签，因此最好的地方是开始一个想[Skip-Thoughts](#)然后在你自己的例子上训练。

### 4.13.10 语音合成

语音合成可能是一个好的方法给予用户返回或者帮助，最近的像[WaveNet](#)显示了深度学习可以提供非常自然的声音。

### 4.13.11 移动机器学习和云

一些使用场景的例子在设备网络结合云服务上给出一个想法。云在控制的环境中有强大的计算力，但是运行在设备上可以提供更高效的交互。在这种情况下云是不可用的或者你的云能力有限，你可以提供一个离线的试验或者通过在设备上处理减少云负载。在设备上计算也是一个信号当它的时间交换了云上的工作。一个好的例子是语音中的关键词检测，因为设备能直接听关键词，一旦识别然后触发了一些通信和基于云的语音识别。没有在设备上的组件，整个应用将不可用，这些样本存在通过一些其它的应用识别一些 sensor 的输入对于更进一步的处理以创建一些有趣的产品是足够感兴趣的。

### 4.13.12 你应该拥有什么软件和硬件？

TensorFlow 运行在 Ubuntu, Windows10, OS X 上。详细的所有支持的操作系统和安装说明查看[Installing TensorFlow](#) 注意我们为移动 TensorFlow 提供多个示例代码要求你从源代码编译 TensorFlow，因此你将需要 pip install 在示例代码上工作。为了试验移动例子，你将需要一个设备用于开发，用[Android Studio](#)如果你开发 iOS 使用[XCode](#)

### 4.13.13 在开始之前你需要做什么？

首先思考如何获得移动机器学习解决方案。

1. 确定你的问题是否能通过移动机器学习解决
2. 创建标记的数据集定义你的问题
3. 为你的问题选择高效的模型

#### 4.13.14 你 的问题是否是移动机器学习能解决的？

关于你想解决的问题当你有一个想法时，你需要计划如何构建你的解。最重要的一步是确保你 的问题实际上是可解的，最好的方法是用人力循环测试。例如如果你想用语音驱动机器人玩具车。尝试从设备记录一些声音，监听如果你可以理解说了什么返回它。经常你将在捕获处理上找到一些问题，像电机淹没了声音或者因为距离不能被听到，你应该在模型处理调查前处理这些问题。另一个例子是给一张从你的 app 拍的照片给人看看是否人能分辨它们是什么。用这种方法寻找。如果它们不能做到（例如尝试从照片估计食物的热量也许是不可能的，因为所有的白色的汤都是一样的，然后你将重新设计你的试验处理它。）一个好的原则是如果一个人不能处理这个任务，希望训练机器人做到更好是很困难的。

#### 4.13.15 创建标记的数据集

在你解决了一些基本的问题后，你需要创建一个标记的数据集来定义你尝试解决的问题。这一步是极其重要的，甚至比选择使用的模型还要重要。你想它能作为你的实际场景下的表达，因此模型将在你教它的任务下变得高效。调查工具尽可能高效和精确地标记数据也是很有价值的。例如，如果你能转化在 web 接口的点击为键盘上的快捷键，你也许能加速生成过程。你应该自己初始化标记，因此你可以了解它的难度和可能的错误，可能改变你的标记和数据捕获处理来避免它。当你和你的团队能组合标记样本（对多数的样本生成相同的标签表示赞同），你可以尝试捕获你的只是在一个手册相互交流而不是如何运行相同的过程。

#### 4.13.16 选择一个高效的模型

下一步是选择一个高效的模型使用。如果有人已经实现了和你的模型类似的你需要的模型你也许能避免训练一个模型；我们在 Github 上有一个[模型仓库](#)，学习你能找到的最简单的模型，尝试在当你有一个小的标记数据是开始，因此你能在快速迭代时将获得最好的结果。很多的时间花费在训练一个模型和在真实应用上运行上，更好的结果你将在最终看到。通常一个算法获得很高的训练精度但是对于实际应用却没有用，因为数据集和真实使用上不匹配。端到端样本使用尽可能创建一个始终如一的用户体验。

#### 4.13.17 下一步

我么你建议你从我们给的[Android](#)和[iOS](#) demo 中开始。TensorFlow Lite 定义了一个新的基于[FlatBuffers](#)模型文件格式。FlatBuffers 是一个开源，高效，跨平台的序列化库。类似[Protocol buffers](#)，但是主要的不同是在访问数据时 FlatBuffers 不需要解析/解包步骤为一个二次表达，经常和对象内存分配成对出现。FlatBuffers 代码尺寸比 protocol buffers 更小。

TensorFlow Lite 是一个新的移动量化平台，目标是跟踪 app 的负载和速度。解释器用一个静态图和一个自定义的（少量动态）内存分配器期待最小的载入，初始化，和高效执行。

TensorFlow Lite 提供一个接口用于硬件加速，如果硬件加速在设备上可用。它通过 Android Neural Network 库（Android O-MR1）加速。

### 4.13.18 为什么需要一个新的专为移动平台设计的库？

机器学习正在改变计算范式，我们看到了嵌入式平台和移动平台融合的趋势。用户期望和它们的设备通过摄像头，声音交互模型以自然，人类喜欢的方式交互。下面是融合的一些因素：

- 在半导体硅上的创新是硬件加速的新的可能，像 Android Neural Network API 框架使得利用硬件加速成为可能
- 现在先进的实时计算机视觉和语义理解导致移动优化基准模型正在被开源
- 在设备智能上为广泛的智能应用创建了新的可能
- 对用户数据隐私不需要离开移动设备的兴趣
- 服务离线情况，这里的设备不需要连接网络

我们相信下一波机器学习应用浪潮将来自于移动平台和嵌入式设备。

### 4.13.19 TensorFlow Lite 开发者预览重点

作为开发者预览的 TensorFlow Lite 包含如下内容：

- 包括量化的和浮点的核心操作已经被转化用于移动平台。这可能用于创建和运行自定义的模型。开发者可以在它们的模型中写自己的操作
- 一个新的[FlatBuffers](#)模型文件格式
- 结合核心优化在移动设备上更快的执行
- TensorFlow 转化器转化 TensorFlow-trained 的模型为 TensorFlow Lite 格式
- 更小的尺寸：TensorFlow Lite 当所有的操作被连接小于 300KB，当用操作需要支持 Inception V3 和 MobileNet 时小于 200KB
- 预先测试的模型：所有的模型被确保效果
  - Inception V3，一个流行在图像上的侦测对象的模型

- [MobileNets](#)一个设计用来高效的在最大化在资源受限或者嵌入式应用中的熟悉的移动端优先的计算机视觉模型。它们能被构建用于分类，检测，嵌入和分割。MobileNet 模型很小但是相比 Inception V3[精度低](#)
- 在设备智能回复，在设备模型上对输入文本信息给出相关的建议信息回复。模型被构建用于内存有限的设备像手表和手机等设备上并且它对于第一方和第三方 app 已经被成功的用于[Smart Replies onAndroid Wear](#)
- 量化的 MobileNet 模型版本，运行比没有量化的版本在 CPU 上运行更快
- 新的 Android 示例程序解释 TensorFlow Lite 结合量化的 MobileNet 模型用于目标检测是如何使用的
- Java 和 C++ API 支持

这是一个开发者版本，可能在将来的 API 中修改，我们不能保证这个版本向后兼容

#### 4.13.20 开始

我们推荐你结合上面的预先测试好的模型进行尝试。如果你有一个存在的模型，你将需要测试是否你的模型和转化器和支持的操作集合兼容。为了测试模型，查看[documentation on GitHub](#)

#### 4.13.21 重新训练 Inception V3 或者 MobileNet 用于用户自定义的数据集

上面提到的模型是基于 ImageNet 数据集 1000 个分类的数据进行训练的。如果这些类对你的使用不相关或者没用，你将需要重新训练模型。这个技术称为迁移学习，在一个已经训练好的模型上重新训练一个类似的问题。深度学习训练可能花费很长时间，但是迁移学习能以被很快的使用。为了做这个事，你将需要生成你的自定义的数据集和相关的类标记。

[TensorFlow for Poets](#)代码实验室告诉你如何一步步做到。重新训练代码支持重新训练用于浮点和量化接口。

#### 4.13.22 TensorFlow Lite 架构

下面的图显示了 TensorFlow Lite 的架构设计

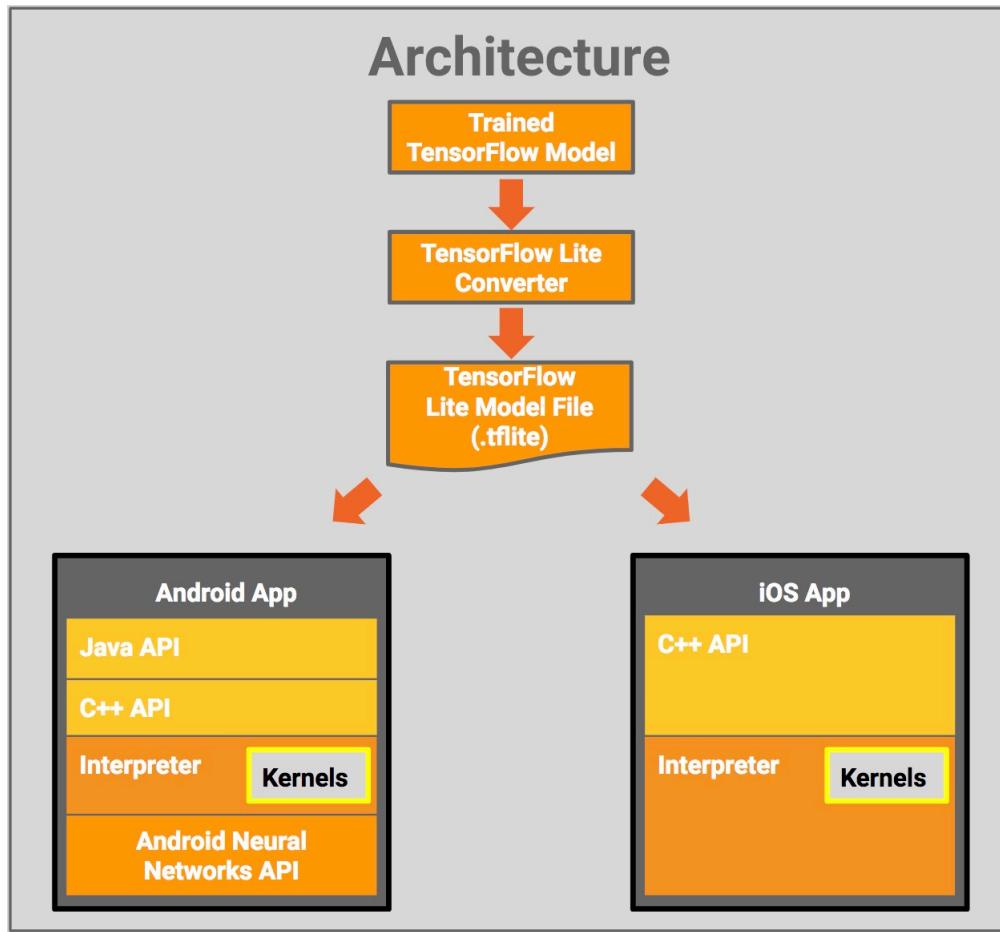


图 4.6: TensorFlow Lite 架构

在磁盘上开始一个训练的模型，你将用 TensorFlow Lite 转化器转化模型为 TensorFlow Lite 文件格式 (.tflite)。然后你可以用转化的文件在你的移动应用上。部署 TensorFlow Lite 模型文件：

- Java API: 一个方便的在 Android 上围绕 C++ API 的包装器
- C++ API: 载入 TensorFlow Lite 模型文件，调用解释器，相同的库在 Android 和 IOS 上都可用
- 解释器：用一些可执行模型。解释器支持选择核心载入；没有核心仅仅 100KB，全部载入核心 300KB。这是从 TensorFlow Mobile 要求的 1.5M 重要的减小
- 在选择的 Android 设备上，解释器将用 Android Neural Network API 用于硬件加速，或者如果没有可用的默认使用 CPU 执行。

你可以用 C++ API 实现用于解释器的自定义核心。

### 4.13.23 将来的工作

在将来的版本中 TensorFlow Lite 将支持更多的模型和内建操作，包括用于对固定点和浮点模型的运行改进，和让开发者更容易的开发工作流和对其它更小的设备的支持等等。正如我们继续开发，我们希望 TensorFlow Lite 将简化开发者对于小型设备的开发者经历。

将来用指定的机器学习硬件获取更好的可能性能用于类似设备上的类似模型。

### 4.13.24 下一步

对于开发者预览，多数文档在 GitHub 上。请查看[TensorFlow Lite repository](#)获取更多信息和代码样例，实例应用和更多。

## 4.14 在 Android 上构建 TensorFlow

为了在 Android 设备上开始 TensorFlow，我们通过两种方法构建我们的 TensorFlow 移动 demo 部署在 Android 设备上。第一种是 Android Studio，让你在 IDE 下构建和部署。第二个是在 Bazel 上构建通过 ADB 命令行部署。

为什么选择一个或者其它的方法？一个简单的在 Android 设备上使用 TensorFlow 的方法是用 Android Studio。如果你没有计划自定义你的 TensorFlow 构建或者你想用 Android Studio 的编辑器或者其它的特性去构建一个 app 仅仅想添加 TensorFlow 在其之上，我们推荐你使用 Android Studio。

如果你正在使用自定义的操作或者有一些其它的原因构建 TensorFlow，向下滚动鼠标查看[使用 Bazel 构建 demo](#)

### 4.14.1 使用 Android Studio 构建 Demo

#### 要求

如果你没有下面要求的软件：

- 按照网站上的说明安装[Android Studio](#)
- 从 Github 上克隆 TensorFlow 仓库`git clone https://github.com/tensorflow/tensorflow`

#### 构建

1. 打开 Android Studio，从欢迎窗口选择 Open an existing Android Studio project
2. 从出现的 Open File or Project 导航选择你克隆的 TensorFlow Github 仓库的目录，点击 OK。如果你出现”Failed to find target with hash string ‘android-23’”，你也许需要安装多平台和工具

3. 打开 build.gradle 文件 (你能用一边面板上的:Project 找到 Android 下的 Gradle Scripts zippy) 如果你没有, 查找 nativeBuildSystem 变量设置它为 none。

```
1 // set to 'bazel', 'cmake', 'makefile', 'none'  
2 def nativeBuildSystem = 'none'
```

4. 点击运行按钮 (绿色箭头) 或者用顶部菜单的 Run -> Run 'android'。如果它要你用实例运行, 点击 Proceed Without Instant Run。你也需要有一个 Android 设开发者选项打开。查看[这里](#)获取设置开发者设备的详细信息。

这安装 TensorFlow Demo 的三个 app 到你的手机上, 查看[Android Sample Apps](#)获取关于它们的更多详细信息。

#### 4.14.2 用 Android Studio 添加 TensorFlow 到你的 app 上

为了添加 TensorFlow 到你自己 Android 设备上的 app, 最简单的方法是添加下面的行到你的 Gradle build file:

```
1 allprojects {  
2     repositories {  
3         jcenter()  
4     }  
5 }  
6  
7 dependencies {  
8     compile 'org.tensorflow:tensorflow-android:+'  
9 }
```

自动下载最新版本的 TensorFlow 作为一个 AAR 安装它到你的工程上。

#### 4.14.3 使用 Bazel 构建 demo

另一个在 Android 设备上使用 TensorFlow 构建一个 APK 的方法是用[Bazel](#)然后使用[ADB](#)载入到你的设备。这要求了解一些构建系统和 Android 开发者工具, 但是我们将通过如下引导你:

- 首先, 跟着我们的说明[installing from sources](#), 这将引导你通过安装 Bazel 和克隆 TensorFlow 代码
- 下载 Android[SDK](#)和[NDK](#), 如果你没有这两个工具的话。你需要至少 12b 的 NDK 和 23 的 SDK
- 在你的复制的 TensorFlow 源代码, 更新[WORKSPACE](#)文件和你的 SDK 和 NSK 位置, 这里的 <PATH\_TO\_NDK> and <PATH\_TO\_SDK>.

## 4.14. 在 ANDROID 上构建 TENSORFLOW

---

- 运行 Bazel 构建 APK demo `bazel build -c opt //tensorflow/examples/android:tensorflow_demo`
- 使用ADB安装到你的设备上 `adb install -r bazel-bin/tensorflow/examples/android/tensorflow_demo.apk`

注意: 当使用 Bazel 为你的 Android 编译你需要在 Bazel 命令行添加`-config=android`, 然而在这个例子是 Android-only, 所以你不需要它

安装三个应用到你的手机上是 TensorFlow Demo 的一部分。查看[Android Sample Apps](#)获取更多关于它们的信息。

### 4.14.4 Android 样例 App

[Android example code](#)是一个简单的工程用相同的代码构建和安装三个 app。样例 app 通过手机的摄像头获取视频。

- TF Classify 使用 Inception V3 模型标记对象指出 Imagenet 的分类。这里仅有 1000 个种类, 失去了大多数常见的目标包含一些你在真实生活中不太可能碰到的目标, 因此结果可能仅供娱乐。例如没有人的分类, 因此它将不知到如何结合人的图片猜测, 像一个座椅或者氧气罩。如果你想自定义这个例子识别你关心的对象, 你可以使用[TensorFlow for Poets codelab](#) 作为一个例子如何在你自己的数据上训练一个模型。
- TF Detect 用多个盒子尝试绘制摄像头中人的位置的边界盒子。这个盒子给出每个侦测结果的置信度。这个 demo 也包含追踪两帧间物体的移动, 因此运行比 TensorFlow 推理更频繁。显然帧率越高用户体验越好, 但是它估计哪个盒子两帧之间涉及相同的对象, 对于实时统计对象很重要。
- TF Stylize 通过摄像头实现一个实时风格迁移算法。你可以选择屏幕中那个风格 混合它们, 因此转化分辨率到更高或者更低

当你构建安装 demo, 你将在你的手机上看到三个 app 图标。触摸它们应该打开 app 让你体验它们做了什么。你能当它们运行的时候通过触摸音量键上激活侧边键。

### 4.14.5 Android Inference Library

因为 Android app 已经被用 Java 写, TensorFlow 核心为 C++, TensorFlow 有 JNI 库连接它们。它的接口针对推理, 因此, 它提供载入图, 建立输入, 运行模型计算类似输出的能力, 你可以查看[TensorFlowInferenceInterface.java](#)的方法的完整的文档。

这个实例程序使用这个接口, 因此它们是一个查找例子使用的好地方。你可以在[ci.tensorflow.org](#)下载预先构建的二进制 jars。

## 4.15 在 iOS 上构建 TensorFlow

### 4.15.1 使用 CocoaPods

最简单的获取在 iOS 上可用的 TensorFlow 是通过 CocoaPods 包管理系统。你可以添加 TensorFlow-experimental pod 到你的 Podfile (安装了一些二进制框架)，这使得开始很容易但是自定义很难 (你想缩减你的二进制文件尺寸很重要)，如果你需要能自定义你的库，查看下面的章节。

### 4.15.2 创建你自己的 app

如果你想添加 TensorFlow 能力到你的 app，做下面的步骤：

- 创建自己的 app 或者载入 XCode 已经创建的 app
- 按照下面添加文件 Podfile 在工程的根目录：

```
1 target 'YourProjectName'
2 pod 'TensorFlow-experimental'
```

- 运行 pod install 下载安装 TensorFlow-experimental pod
- 打开 YourProjectName.xcworkspace 添加你的代码
- 在你的 app 的 Build Settings, 确保添加 \$(inherited) 到 Other Linker Flags 和 Header Search Paths 部分

### 4.15.3 运行样例

你将需要 XCode7.3 或者更新的版本运行我们的 iOS 样例。当前有三个例子：simple, benchmark, 和 camera。到现在你能通过克隆主要的 tensorflow 仓库下载样本代码（之后我们计划分割仓库为可用样例）

从 TensorFlow 文件夹根目录，下载[Inception v1](#)，用下面的步骤提取简单的和相机例子标签和图文件到数据文件夹：

```
1 mkdir -p ~/graphs
2 curl -o ~/graphs/inception5h.zip \
3   https://storage.googleapis.com/download.tensorflow.org/models/inception5h.zip \
4   && unzip ~/graphs/inception5h.zip -d ~/graphs/inception5h
5 cp ~/graphs/inception5h/* tensorflow/examples/ios/benchmark/data/
6 cp ~/graphs/inception5h/* tensorflow/examples/ios/camera/data/
7 cp ~/graphs/inception5h/* tensorflow/examples/ios/simple/data/
```

改变进入样例目录，下载Tensorflow-experimental pod, 打开 Xcode 工作空间，注意安装 pod 可能花费很长时间，因为它很大 ( 450M). 如果你想运行简单的例子，然后：

```
1 cd tensorflow/examples/ios/simple  
2 pod install  
3 open tf_simple_example.xcworkspace  # note .xcworkspace, not .xcodeproj  
4                                     # this is created by pod install
```

在 XCode 仿真器运行简单的 app。你应该看到一个有 Run Model 按钮的屏幕。按下它，你应该看到一些调试输出表明样例 Grace Hoper 图像在目录中的数据已经被分析，结合多个独一无二的识别。

用同样的过程运行其它的例子。相机样例要求是个真正的设备被连接上。当你构建和运行时你应该得到实时相机的场景，你能指出目标得到实时识别结果。

### 4.15.4 iOS 样本详情

针对 iOS 有三个实例应用，所有都定义在[tensorflow/examples/ios](#) Xcode 工程下

- Simple: 一个小的例子显示如何用尽量少的行，载入运行 TensorFlow 模型。它仅仅结合一个按钮组合一个简单的场景，按下就运行模型载入和推理
- Camera 一个和 Android TF Classify Demo 十分类似的 demo。它载入 Inception V3 为实时相机场景输出它的最好的标签估计。正如 Android 版本，你能用 TensorFlow for Poets 训练你自己的自定义的模型用最小的代码改动进入这个例子
- Benchmark: 和 Android 设备上的基准测试工具类似，很接近 Simple 但是重复的运行图输出类似的统计

### 4.15.5 Trubleshooting

- 确保你用 TensorFlow-experimental pod(不是 TensorFlow)
- TensorFlow-experimental pod 当前大约 450M。这么大是因为构建多个平台，pod 包含所有的 TensorFlow 函数。最终 app 的大小在构建之后减小到 ( 25M)。在开发的过程中结合完整的 pod 是很方便的，查看下面的章节了解你如何构建你自己的 TensorFlow 库，减小尺寸。

### 4.15.6 从源代码构建 TensorFlow iOS 库

尽管 Cocapods 是最简单和最容易的开始方式，你有时候需要更灵活的决定 TensorFlow 在你的 app 中应该被传递。例如，你可以从源代码构建 iOS 库。[导航](#)包含了如何去做的。

## 4.16 整合 TensorFlow 库

当你在模型上做了一些进步处理了一些你正尝试解决的问题，在你的应用上立即测试是很重要的。经常在你的训练数据和实际上真实世界碰到的问题有些不同，得到一个干净的图像提高产品的体验。

这页讨论当你已经成功的构建和部署 TensorFlow 移动 demo app 后整个 TensorFlow 库进入你自己的移动应用模型。

### 4.16.1 链接库

在你成功的构建了样例后，你将可能想从你的存在的应用上调用 TensorFlow。最简单的方法是使用[这里](#)描述的 Pod 安装，但是如果你想从源代码构建 TensorFlow(自定义包含的操作) 你将需要跳出 TensorFlow 作为一个框架，包含正确的头文件，连接针对构件库和依赖。

### 4.16.2 Android

对于 Android, 你需要链接包含 JAR 文件的 libandroid\_tensorflow\_inference\_java.jar 的 Java 库。有三种方法包含这个功能到你的程序中:

- 包含 jcenter AAR, 正如[example app](#)
- 从[ci.tensorflow.org](https://ci.tensorflow.org)下载 nightly 版本的预编译
- 用[Github repo](#)中的说明构建 JAR 文件。

### 4.16.3 iOS

获取一个在 iOS 上的 TensorFlow 库有一点复杂。这里是一个你的 iOS app 需要的列表:

- 通常添加-L /your/path/tensorflow/contrib/makefile/gen/lib/ and -ltensorflow-core 到你的链接器标记，链接 tensorflow/contrib/makefile/gen/lib/libtensorflow-core.a
- 添加-L /your/path/tensorflow/contrib/makefile/gen/protobuf\_ios/lib and -lprotobuf 和 -lprotobuf-lite 到你的命令行生成 protobuf 库
- 对于包含的路径，你需要你的 TensorFlow 元文件的根目录作为入口，如下 tensorflow/contrib/makefile/downloads/protobuf/src, tensorflow/contrib/makefile/downloads, tensorflow/contrib/makefile/downloads/eigen, and tensorflow/contrib/makefile/gen/proto

- 确保你的二进制用-force\_load (或者对应的平台) 针对 TensorFlow 库确保链接正确。更多的为什么这需要的细节在下面章节说明, [Global constructor magic](#) 在类 linux 平台, 你将需要不同的 flags, 更多像-Wl,-allow-multiple-definition -Wl,-whole-archive

你将需要链接加速器框架, 因此这被用于加速一些操作。

#### 4.16.4 全局结构体 magic

当你尝试从你自己的应用调用 TensorFlow 一个不太明显的错误是 No session factory registered for the given session options。你将需要了解 TensorFlow 架构才能明白为什么错误发生和如何修复。

这个框架被设计的十分模块化, 接了很小的核心和大量的指定的独立对象可能按照需要混合匹配。为了使这工作, C++ 的代码模型必须让模块容易通知框架关于它们提供的服务, 没有要求一个中心列表必须从每个实现被单独更新。很难允许单独的库没有需要的预编译的核被添加它们自己的实现。

为了获得能力, TensorFlow 在一些地方使用用一个注册模板。代码看起来如下:

```
1 class MulKernel : OpKernel {
2     Status Compute(OpKernelContext* context) { ... }
3 };
4 REGISTER_KERNEL(MulKernel, "Mul");
```

这将标准的.cc 文件链接进你的应用, 无论是作为主核心的一部分还是作为分割的自定义库的一部分。magic 部分是 REGISTER\_KERNEL() 宏能通知 TensorFlow 核心必须实现乘法操作, 因此它能在图上任何请求它的地方调用。

从编程的观点, 这建立是非常方便的。在相同的文件实现和注册代码, 添加新的实现作为一个简单的编译和链接。困难的部分是 REGISTER\_KERNEL() 宏用 C++ 实现。C++ 没有提供一个好的机制处理注册, 因此我们必须重排一些代码。如下, 宏被如下实现:

```
1 class RegisterMul {
2 public:
3     RegisterMul() {
4         global_kernel_registry()->Register("Mul", []() {
5             return new MulKernel();
6         });
7     }
8 };
9 RegisterMul g_register_mul;
```

这结合构造体建立一个 RegisterMul 类告诉全局内核当有人询问如何创建一个 Mul 核心的时候什么函数被调用。当有一个全局对象在类中, 构造体应该在任何程序的开始以调用。尽管这容易理解, 不幸的是全局对象定义没有被任何其它的代码使用, 因此链接器没有被

结合这个想法设计，决定可能被删除。因此，构造体没有被调用，类没有注册。所有的在 TensorFlow 中的排序的模型，Session 实现首先当代码运行的时候被查找，这也是为什么当问题出现时显示属性错误的原因。

解强制链接器不从库中删除任何代码，即使它相信它没有使用。在 iOS，这一步可能结合-force\_load flag 实现，指定库的路径，在 Linux 你需要--whole-archive。这些告诉链接器不被强制的删掉，应该保留全局。

实际的多种 REGISTER\_\* 实现是一个复杂的实践，但是他们遭遇相同的问题。如果你感兴趣它们是如何工作的，[ok\\_kernel.h](#)是一个开始学习的好地方。

#### 4.16.5 Protobuf 问题

TensorFlow 依赖于[Protocol Buffer](#)通常称为 protobuf。这个库定义数据结构和产品序列为不同的语言对它定义访问代码。技巧是生成的代码链接共享库为用于生成器的提取相同版本的框架。当 protoc 时可能是一个问题，这个工具用于从不同版本的 protobuf 而不是包含在路径中的库和标准链接 protoc 生成代码。例如，你可以复制构建在 /projects/protobuf-3.0.1.a 的 protoc，但是你需要库已经安装在/usr/local/lib 和 /usr/local/include 3.0.0 版本。

问题的症状是结合 protobufs 在编译或者链接时出现错误。通常构建工具考虑到了这个问题，但是如果你用 makefile，确保你正在本地使用它构建的 protobuf 库，正如[这个 Makefile](#)。

另一种情况是当 protobuf 的 touch 和源文件需要作为编译程序的一部分时导致错误。这个过程使得编译更复杂，因此，第一步必须被传送 protobuf 定义创建所有需要的代码文件，之后你可以继续和构建代码库。

#### 4.16.6 在同样的 app 中有多个版本的 protobufs

Protobufs 生成的头文件需要作为 C++ 接口的一部分到整个 TensorFlow 库，这复杂了作为标准框架使用库。

如果你的程序已经使用 protocol buffers 库版本 1，你整合 TensorFlow 也许会出现一些问题因为它要求版本 2。如果你尝试链接两个版本到相同的库，你将看到链接错误因为一些符号冲突。为了解决类似的问题，我们有一个试验脚本在[rename\\_protobuf.sh](#)。你需要运行下面目录作为 makefile 构建的一部分，之后你下载所有的依赖：

```
1 tensorflow/contrib/makefile/download_dependencies.sh  
2 tensorflow/contrib/makefile/rename_protobuf.sh
```

### 4.16.7 调用 TensorFlow API

当你有可用框架的时候你需要调用它。常见的模式是你先载入你的代表数值计算的模型，然后运行输入到模型（例如相机的图像）接收输出（例如，预测标签）

在 Android 上，我们提供 Java 推理库处理这种用法，然而在 iOS 和 Raspberry Pi 你可以直接调用 C++ API。

### 4.16.8 Android

这里是 Android 上典型的推理库序列：

```

1 // Load the model from disk.
2 TensorFlowInferenceInterface inferenceInterface =
3 new TensorFlowInferenceInterface(assetManager, modelFilename);
4
5 // Copy the input data into TensorFlow.
6 inferenceInterface.feed(inputName, floatValues, 1, inputSize, inputSize, 3);
7
8 // Run the inference call.
9 inferenceInterface.run(outputNames, logStats);
10
11 // Copy the output Tensor back into the output array.
12 inferenceInterface.fetch(outputName, outputs);

```

你可以在[Android examples](#)上找到源代码。

### 4.16.9 iOS 和 Raspbrry Pi

这里是对于 iOS 和 Raspbrry Pi 的对应代码：

```

1 // Load the model.
2 PortableReadFileToProto(file_path, &tensorflow_graph);
3
4 // Create a session from the model.
5 tensorflow::Status s = session->Create(tensorflow_graph);
6 if (!s.ok()) {
7   LOG(FATAL) << "Could not create TensorFlow Graph: " << s;
8 }
9
10 // Run the model.
11 std::string input_layer = "input";
12 std::string output_layer = "output";
13 std::vector<tensorflow::Tensor> outputs;
14 tensorflow::Status run_status = session->Run({ {input_layer, image_tensor} },

```

```

15         {output_layer}, {}, &outputs);
16 if (!run_status.ok()) {
17   LOG(FATAL) << "Running model failed: " << run_status;
18 }
19
20 // Access the output data.
21 tensorflow::Tensor* output = &outputs[0];

```

这是所有的基于[iOS sample code](#), 但是没有 iOS 指定; 同样代码应该在任何其它支持 C++ 的平台上可用。你可以找到 Raspberry Pi 的[例子](#)

## 4.17 为移动部署准备模型

这要求你在训练过程中储模型信息和你想释放它作为移动 app 的一部分非常不同, 这个章节包含从训练模型到产品释放的一些转化工具。

### 4.17.1 保存的文件格式有什么不同?

你可能发现 tensorflow 保存图的一些不同的方法变得非常的困惑。为了帮组理解, 下面是一些不同组件的关于用于何处的介绍。目标通常被定义和序列化为 protocol buffers:

- NodeDef: 在模型中定义一个操作。它有一个独一无二的名字, 一些其它节点的名字列表接收输入, 才做类型实现 (例如 Add 或者 Mul) 和任何需要控制操作的属性。这是 TensorFlow 基本的计算单元, 所有网络节点的工作通过迭代被做, 轮流使用每一个。这也许是一个单个的标量或者字符创, 但是也保存一整个多为 TensorFlow 数组。Constent 常熟的值被存在 NodeDef, 因此当序列化的时候大的常熟可能占据一些空间
- Checkpoint。另一个通过使用 variable 操作存储模型值的方法。不像 Const 操作, 这不存储它们的内容作为 NodeDef 的一部分, 因此它们在 GraphDef 中占据非常小的空间。当计算的时候它们的值在 RAM 中保持, 然后周期性的输出 checkpoint 文件到磁盘。这通常发生在神经网络被训练和权重被更新的时候, 因此它是一个时间敏感的操作, 也许发生在分布式的一些工作站上, 因此文件格式必须快和灵活。它们存储作为多个 checkpoint 文件的一部分, 结合 checkpoint 文件中的 metadata 文件描述 c。当你通过 API 访问 (例如传递一个文件名作为命令行参数) checkpoint 文件, 你见够用通常的前缀和文件关联。如果你有这些文件

```

1 /tmp/model/model-chkpt-1000.data-00000-of-00002
2 /tmp/model/model-chkpt-1000.data-00001-of-00002
3 /tmp/model/model-chkpt-1000.index
4 /tmp/model/model-chkpt-1000.meta

```

你将访问它们作为/tmp/model/chkpt-1000

- GraphDef: 有一个 NodeDefs 列表，结合定义计算图执行。在训练中一些节点将为 Variables，因此如果你想有一个完整的可运行的图，包含权重，你将需要调用恢复操作从 checkpoint 文件获取这些值。因为 checkpoint 载入必须灵活的处理所有的训练请求，这可能是一个在移动端和嵌入式设备实现的技巧，特别是像 iOS 没有合适的文件系统可用的情况。这里的手写脚本[freeze\\_graph.py](#)。正如上面提到的，常数操作存储它们的值作为 NodeDef 的一部分，因此如果所有的 Variable 权重被转化为常数节点。你可以在单次调用中载入结果文件，不需要从 checkpoint 中恢复变量值。GraphDef 文件有时候你已经存储在文本格式以方便查看。这个版本通常有.pbtxt 文件后缀，正如二进制文件的.pb 后缀一样
- FunctionDefLibrary: 在 GraphDef 出现，是高效的设置子图，关于输入输出的信息。每个子图可以用作主图的操作，运行简单的实例化不同的节点，用泪水的方式用其它语言包装代码
- MetaGraphDef: 一个空白的 GraphDef 仅仅有关于计算网络的信息，但是没有额外的关于模型的信息或者模型如何使用的信息。MetaGraphDef 包含一个 GraphDef 定义模型的部分计算，而且包含像 signatures 的信息（暗示你想要调用的的输入输出，数据和 Checkpoint 文件存储地），对于一组操作方便的标签
- SavedModel: 通常享有不同版本的图依赖一些 chechpoint 的变量。例如，你也许在一张图上需要需要 GPU 和 CPU 版本为两者保存权重。你也许需要额外的文件（像标记文件）作为你的模型的一部分。SavedModel 文件通过存放你保存的多版本的图不复制变量，也在同一个 bundle 存储 assert 文件处理这些需要。在 hood 下，用 MetaGraphDef 和 checkpoint 文件，伴随着额外的 metadata 文件。如果你用 TensorFlow Serving 部署 web API 它的格式你将想使用它。

### 4.17.2 如何获取一个能在移动端运行的模型？

多数情况下，结合 TensorFlow 训练一个模型给你一个包含 GraphDef 的文件（通常以.pb 或者.pbtxt 为扩展名）和一些 Checkpoint 文件。用于移动设备和嵌入设备部署的是被 frozen 的单个的 GraphDef 文件或者由一个变量转化为常数的所有文件，为了处理，你将需要在[tensorflow/python/tools/freeze\\_graph.py](#)脚本，你需要像这样运行：

```

1 bazel build tensorflow/tools:frozen_graph
2 bazel-bin/tensorflow/tools/freeze_graph \
3   --input_graph=/tmp/model/my_graph.pb \
4   --input_checkpoint=/tmp/model/model.ckpt-1000 \
5   --output_graph=/tmp/frozen_graph.pb \

```

---

```
6 --output_node_names=output_node \
```

输入 `input_graph` 参数应该指向保存你的模型架构的 `GraphDef` 文件。你的 `GraphDef` 必须以文本格式存储在磁盘，这种情况下以`.pbtxt` 结尾而不是`.pb`，你应该添加一个额外的`-input_binary=false` 标记到命令中。

`input_checkpoint` 应该是最近保存的 `checkpoint` 文件。正如 `checkpoint` 章节提到的，你需要给常规的前缀到 `checkpoint` 文件而不是一个完整的名字。

`output_graph` 定义 `frozen GraphDef` 将被存储在哪里。因为它可能包含一些占据较大磁盘空间的权重值，通常它总是保存为二进制的 `protobuf`。

`output_node_names` 是一个你想从图中提取的节点的名称列表。这时需要的是因为 `freezing` 进程需要明白图的哪一部分是实际需要的，哪一部分需要人工训练的，像一些总结操作一样。唯一的贡献计算给输出节点将被保存的操作。如果你知道你的图正在被使用。这些应该是节点的名字你传递进 `Session::run()` 作为你的获取目标。最简单的摘到节点名称的方法是在 Python 中构建你的图查看节点对象。在 `TensorBoard` 是另一个简单的方法。你可以通过[summarize\\_graph tool](#)获取可能的输出建议。

因为 `TensorFlow` 的输出格式已经随着时间改变，有一些通常比较少使用的的 flag 也可以使用，像 `input_saver`，但是希望你不要用现代的框架版本在图上训练。

### 4.17.3 用图转换工具

你需要在设备上高效运行图的一些事是通过[Graph Transform Tool](#)，这个命令行工具接收一个输入 `GraphDef` 作为输入应用一些你要求的重写规则，然后写出结果到 `GraphDef`。查看文档获取构建和运行这个工具的更多信息。

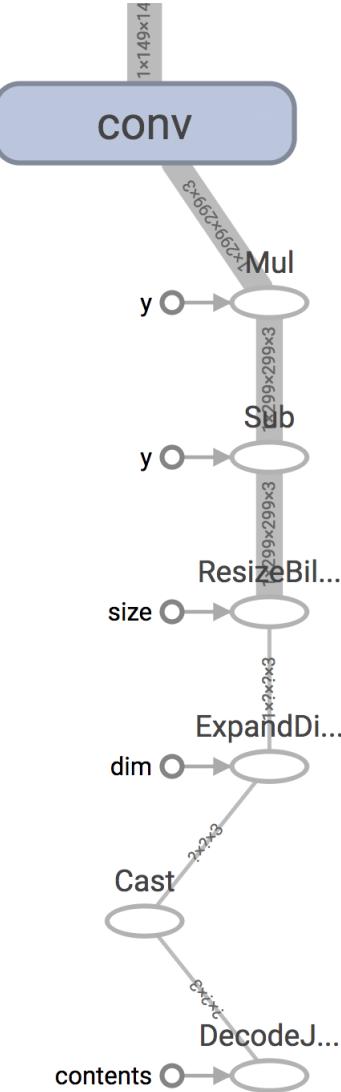
### 4.17.4 移除仅仅训练节点

`TensorFlow GraphDefs` 通过训练包含所有的需要反向计算和权重更新的计算，只要查询和解码输入，保存 `checkpoint`。所有的节点不再需要推理，一些像 `checkpoint` 保存的文件甚至不支持移动平台。为了创建一个模型文件你可能需要通过运行 `Graph Transformtool` 的 `strip_unused_node` 规则删除这些不需要的操作。

处理的最薄弱的部分是在推理过程中找出你想用作输入和输出的节点的名字。当你开始推理的时候需要这些方法，但是你也需要它们以至于转换可以计算哪一个节点在推理路径中不需要。这也许不同于训练代码。最简单的方法是用 `TensorBoard` 决定图中节点被访问。

记住，移动设备通常通过传感器收集数据以数组的形式保存，由于训练通常涉及载入和解码存储在磁盘上的数据表达。以 `Inception V3` 为例，有一个 `DecodeJpeg op` 开始图设计用于从磁盘接收 JPEG 编码的数据转换为任意大小的图像，之后有一个 `BilinearResize op` 缩

放它到希望的大小，接着一些操作转换字节数据为浮点数，用图希望的方式放大在里面值的幅度。一个典型的移动 app 将跳过这些步骤直接从相机开始输入，因此输入节点你将应用实际上成为输出 Mul 节点 你需要做类似查看找出正确的输出节点。如果你已经用一个 frozen



GraphDef 文件，不确保内容尝试使用 summarize\_graph 工具打印出从图结构中找到的关于输入和输出的信息。这里是一个原始的 Inception V3 文件的例子 `bazel run tensorflow/tools/graph_transforms:summarize_graph -- i --in_graph=tensorflow_inception_graph.pb` 当你明白输入输出节点是什么，你可以分别对应 `--input_names` 和 `--output_names` 参数输入它们到图转化工具，调用 `strip_unused_nodes` 变换，如下：

```

1 bazel run tensorflow/tools/graph_transforms:transform_graph --
2   --in_graph=tensorflow_inception_graph.pb
3   --out_graph=optimized_inception_graph.pb --inputs='Mul' --outputs='softmax'
  
```

```

4 --transforms=
5 strip_unused_nodes(type=float, shape="1,299,299,3")
6 fold_constants(ignore_errors=true)
7 fold_batch_norms
8 fold_old_batch_norms

```

一件需要留心的事是你需要指定你想的输入的大小和类型。这是因为一些你将传入作为输入用于推理的值需要输入到特殊的 Placeholder op 操作，转如果它们不存在换也许需要创建它们，用 Inception V3 的例子，一个 Placeholder 节点取代以前的 Mul 节点用于输出 resize 和 rescale 图像矩阵。因此我们在调用 TensorFlow 之前正在处理我们自己。为了保留原始的名字，这也是为什么我们总是 feed 输入到 Mul 后结合我们修改的 Inception 图然后运行一个 session。在你运行这个程序后，必将有一个仅仅包含需要运行预测程序的当前节点。在这点上在图上运行度量变得十分有用，因此它的值得再次运行 summary\_graph 明白在你的模型上有什么。

#### 4.17.5 什么操作应该被包含在移动端？

在 TensorFlow 中有上百个可用的操作，每个操作针对不同的数据类型有多种实现。在移动平台编译后的二进制文件的大小非常重要，因为 app 编译的东西越小用户体验越好。如果所有的 TensorFlow 操作和数据类型被编译进入 TensorFlow 库中编译库可能有上 10M，因此默认只有操作和数据类型的自己被包含。

这意味着如果你载入一个已经在桌面环境下训练过的模型文件，当你在移动平台载入它时你也许看到 No OpKernel was registered to support Op 错误。首先确保你删除了仅仅训练的节点，因为如果你的 op 从未执行，错误将在载入时发生。如果当你照做后依然出现同样的问题，你将需要查看添加操作到你的构建库。

包含操作和类型的标准有如下类别：

- 他们是不是仅仅在反向传播或者梯度计算中有用？因此移动集中在推理，我们不包含这些。
- 他们对于其他训练需要，像 checkpoint 保存是有用的？这里我们 leave out
- 是否依赖的框架是移动端可用的，如 libjpeg。避免额外的依赖，我们不包含 Decode-Jpeg 操作
- 有不常使用的类型？我们不包含操作的 boolean 变量，因此我们不能在通常的推理图上看到它们中大多数

操作默认被修剪以适应移动平台的推理，但是可以修改构建文件修改默认。在修改构建文件后，你将需要重新编译 TensorFlow。查看下面获取如何去做的更多细节，也可以查看[Optimizing](#)查看更多减小二进制文件尺寸的信息。

#### 4.17.6 定位实现

操作被分为两部分。第一部分是操作的定义，什么操作的 signature 声明，哪个输入，输出，拥有属性。这将占据非常小的空间，这些默认被包含。操作计算的实现在 kernel (在 tensorflow/core/kernels 文件夹) 中实现，你需要编译包含 C++ 文件的内核实现进入库。为了找出那个文件是，你可以在源文件中搜索操作的名字。[Here's an example search in github](#) 你将看到这个搜索正在寻找 Mul 操作实现，它在 tensorflow/core/kernels/cwise\_op\_mul\_1.cc 中。你需要用 REGISTER 开始寻找宏，你关心的操作的名字作为一个字符串参数。

在这种情况下，实现拆分分为多个.cc 文件，因此你需要在构建它们的过程中包含它们。如果你觉得用命令行搜索代码合适，这里是一个 grep 命令，如果你从 TensorFlow 仓库的根目录运行它能定位正确的文件 grep 'REGISTER.\*"Mul"' tensorflow/core/kernels/\*.cc

#### 4.17.7 添加实现构建

如果你用 Bazel, 为 Android 构建，你将需要添加文件[android\\_extended\\_ops\\_group1](#)或者[android\\_extended\\_ops\\_group2](#)目标。你也许需要包含任何的在这里依赖的.cc 文件。如果构建报出缺少头文件，添加.h 的文件到[android\\_extended\\_ops](#)。如果你用一个 makefile 在 iOS, Raspberry Pi 等等，去[tensorflow/contrib/makefile/tf\\_op\\_files.txt](#)添加正确的实现文件。

#### 4.17.8 为移动端优化

当你尝试部署在移动设备或者嵌入式设备上时，有一些特殊问题需要处理，当你部署你的模型的时候你将需要考虑它。

这些问题：

- 模型和执行文件的大小
- App 速度和模型载入速度
- 性能和线程

#### 4.17.9 TensorFlow 最低的设备要求是什么？

你需要至少一兆的程序存储空间和多兆 RAM 运行基本的 TensorFlow 环境，因此对于 DSP 或者微控制器来说是不合适的。相比于这些，最大的约束是设备的计算速度和是否你能用较低的代价为你的应用运行模型。你可以在[How to Profile your Model](#)使用基准测试工具了解你的模型需要多少 FLOPs，然后使用这个经验法则估计它们将在不同的设备上运行多快。例如，现代的智能手机也许有 10GFLOPs/s，因此你希望从一个 5GFLOP 模型是每秒两帧，尽管你也许做的比较差的依赖同计算样板提取什么。

模型依赖是的在比较旧的或者资源少的手机上运行 TensorFlow 成为了可能。只要你优化你的网络结合延迟 budget 在尽可能限制的 RAM，我们需要最大程度确保 TensorFlow 创建的中间缓存不要太大，你也可以检查基准输出。

#### 4.17.10 Speed

一个很高优先级的模型部署是找出如何更快的运行推理给出一个好的用户体验。首先是查看执行图要求的浮点操作。你可以使用 benchmark\_model 工具获取原始的估计。

```
1 bazel build -c opt tensorflow/tools/benchmark:benchmark_model && \
2 bazel-bin/tensorflow/tools/benchmark/benchmark_model \
3 --graph=/tmp/inception_graph.pb --input_layer="Mul:0" \
4 --input_layer_shape="1,299,299,3" --input_layer_type="float" \
5 --output_layer="softmax:0" --show_run_order=false --show_time=false \
6 --show_memory=false --show_summary=true --show_flops=true --logtostderr
```

这将给你显示运行图需要多少操作。你也可以用这信息找出你的模型在目标上如何可行。例如，2016 年出的高端手机能每秒到 200 亿 FLOPs。因此你可能希望的最佳速度要求在大约 500ms 范围达到 100 亿 Flops。在像 Raspberry Pi 3 上可能有 50 亿 Flops，你也许仅仅没每步得到一个推理。

有这些估计帮助你计划在设备上能获取真是的估计。如果模型有太多操作，然后有一些机会优化架构减少数量。高级的技术包含 SqueezeNet MobileNet (针对移动设备快速但是牺牲少量精度)。你也可以查看一些可用的模型，及时老的模型，也许也很小。例如，Inception V1 仅有 700 万参数，相比于 Inception V3 的 240 万参数，要求 3 亿 FLOPs 而不是 v3 的 9 亿参数。

#### 4.17.11 模型大小

运行在设备上的模型需要被存储在设备上的某个地方，非常大的神经网络可能有数百兆。多数用户不愿从 app store 下载非常大的捆绑包，因此你想保证你的模型尽可能小。更进一步，越小的神经网络能在移动设备上运行的越快。为了明白你磁盘上的网络多大，在上面（查看 Preparing models 了解更多关于这个工具的信息）运行 freeze\_graph 和 strip\_unused\_nodes 通过查看你的 GraphDef 的大小。因此当它仅仅包含推理相关的节点时。再次检查你的结果，运行 summarize\_graph 工具查看多少参数是常数：

```
1 bazel build tensorflow/tools/graph_transforms:summarize_graph && \
2 bazel-bin/tensorflow/tools/graph_transforms/summarize_graph \
3 --in_graph=/tmp/tensorflow_inception_graph.pb
```

这个命令应该给你类似下面的输出：

```
1 No inputs spotted.
```

```

2 Found 1 possible outputs: (name=softmax, op=Softmax)
3 Found 23885411 (23.89M) const parameters, 0 (0) variable parameters,
4 and 99 control_edges
5 Op types used: 489 Const, 99 CheckNumerics, 99 Identity, 94
6 BatchNormWithGlobalNormalization, 94 Conv2D, 94 Relu, 11 Concat, 9 AvgPool,
7 5 MaxPool, 1 Sub, 1 Softmax, 1 ResizeBilinear, 1 Reshape, 1 Mul, 1 MatMul,
8 1 ExpandDims, 1 DecodeJpeg, 1 Cast, 1 BiasAdd

```

我们当前意图的重要部分是常数参数的数量。在多数模型中将被存储作为 32 位浮点数，因此如果你将参数乘上 4，你应该得到接近磁盘上文件的大小。你可以用 8 位参数在最终的结果中损失很小的精度。因此如果你的文件太大你可以尝试使用`quantize_weights`减小参数：

```

1 bazel build tensorflow/tools/graph_transforms:transform_graph && \
2 bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
3 --in_graph=/tmp/tensorflow_inception_optimized.pb \
4 --out_graph=/tmp/tensorflow_inception_quantized.pb \
5 --inputs='Mul:0' --outputs='softmax:0' --transforms='quantize_weights'

```

如果你查看结果文件的大小，你应该看到它有原来 23M 文件的 1/4。

另一个变换是`round_weights`，它并不能使文件变得更小，但是它是文件能压缩`quantize_weights`使用时的相同大小。它的特殊用途是移动部署，利用 app 在用户下载前安装包被压缩。

原始文件用标准的算法不能压缩的很好，因为每个类似的数的位数可能非常不同。`round_weights`变换保持权重参数以浮点数存储，但是四舍五入它们到最近的值。这意味着存储的模型中有一些重复的字节模式，因此压缩可能带来尺寸的巨大减小，在一些情况下为了尽量小的存储他们被作为 8 位存储。`round_weights`的另一个好处是框架不用分配临时 buffer 给解包的参数，当我们必须的时候我们仅仅使用`quantize_weights`。这保存一些时延（尽管结果应该被缓存以至于仅仅在第一次运行时花费）保证它能用内存映射，正如后面描述的。

#### 4.17.12 二进制文件的大小

移动端和服务端最大不同是二进制文件的大小很重要。在桌面环境下它通常有上百兆，但是对于移动和嵌入式 app 保持二进制文件足够小以至于用户能轻松下载。正如上面提到的，TensorFlow 包含默认操作实现的子集，但是最后的可执行文件仍然有 12M。为了减小这，你可以建立你确实需要仅仅包含操作实现的库，基于自动分析你的模型。使用它。

- 在你的模型上运行`tools/print_required_ops/print_selective_registration_header.py`生成使 op 能用的头文件
- 放置`ops_to_register.h`文件到编译器能找到的路径。这可以是你的 TensorFlow 源文件的根目录。

- 用 SELECTIVE\_REGISTRATION 定义构建 TensorFlow，例如通过传递`--copts=-DSELECTIVE_REGISTRATION` 到你的 Bazel 构建命令。

这个处理重新编译库以至于仅仅你需要的操作和类型被包含，这可能极大地减小可执行文件的大小。例如，结合 Inception v3，新的大小仅仅 1.5M。

### 4.17.13 如何探测你的模型

当你了解你的设备的峰值性能后，值得查看当前性能。使用标准的 TensorFlow 基准，而不是在更大的 app 中运行它，帮助隔离 TensorFlow 与时延。[tensorflow/tools/benchmark](#) 工具设计用来帮助你做这个。为了在你的桌面机器中运行 Inception V3，构架基准模型：

```
1 bazel build -c opt tensorflow/tools/benchmark:benchmark_model && \
2 bazel-bin/tensorflow/tools/benchmark/benchmark_model \
3 --graph=/tmp/tensorflow_inception_graph.pb --input_layer="Mul" \
4 --input_layer_shape="1,299,299,3" --input_layer_type="float" \
5 --output_layer="softmax:0" --show_run_order=false --show_time=false \
6 --show_memory=false --show_summary=true --show_flops=true --logtostderr
```

你应该看到类似下面的输出：

```
1 ━━━━━━━━━━ Top by Computation Time ━━━━━━━━
2 [node
3   [type]  [start]  [first]  [avg ms]  [%]  [cdf%]  [mem KB]  [Name]
4 Conv2D    22.859   14.212   13.700  4.972%  4.972%  3871.488 conv_4/Conv2D
5 Conv2D     8.116    8.964   11.315  4.106%  9.078%  5531.904 conv_2/Conv2D
6 Conv2D    62.066   16.504    7.274  2.640% 11.717%  443.904 mixed_3/conv/Conv2D
7 Conv2D     2.530    6.226    4.939  1.792% 13.510%  2765.952 conv_1/Conv2D
8 Conv2D    55.585   4.605    4.665  1.693% 15.203%  313.600 mixed_2/tower/
      conv_1/Conv2D
9 Conv2D   127.114    5.469    4.630  1.680% 16.883%   81.920 mixed_10/conv/
      Conv2D
10 Conv2D   47.391    6.994    4.588  1.665% 18.548%  313.600 mixed_1/tower/
      conv_1/Conv2D
11 Conv2D   39.463    7.878    4.336  1.574% 20.122%  313.600 mixed/tower/conv_1/
      Conv2D
12 Conv2D   127.113    4.192    3.894  1.413% 21.535%  114.688 mixed_10/tower_1/
      conv/Conv2D
13 Conv2D   70.188    5.205    3.626  1.316% 22.850%  221.952 mixed_4/conv/Conv2D
14
15 ━━━━━━━━━━ Summary by node type ━━━━━━━━
16 [Node type]  [count]  [avg ms]  [avg %]  [cdf %]  [mem KB]
```

```

17 Conv2D           94   244.899    88.952%   88.952%  35869.953
18 BiasAdd          95    9.664     3.510%   92.462%  35873.984
19 AvgPool           9    7.990     2.902%   95.364%  7493.504
20 Relu              94    5.727     2.080%   97.444%  35869.953
21 MaxPool            5    3.485     1.266%   98.710%  3358.848
22 Const             192   1.727     0.627%   99.337%  0.000
23 Concat             11   1.081     0.393%   99.730%  9892.096
24 MatMul             1    0.665     0.242%   99.971%  4.032
25 Softmax             1    0.040     0.015%   99.986%  4.032
26 ◇                 1    0.032     0.012%   99.997%  0.000
27 Reshape             1    0.007     0.003%   100.000% 0.000
28
29 Timings (microseconds): count=50 first=330849 curr=274803 min=232354 max=415352
   avg=275563 std=44193
30 Memory (bytes): count=50 curr=128366400(all same)
31 514 nodes defined 504 nodes observed

```

这是总结的视图，它能通过 show\_summary 标记显示。为了解释它首先表格是一个节点花费时间的列表。花费时间的排序，从左到右，列是：

- 节点类型，操作的种类
- op 的开始时间，显示序列操作在哪里下降
- 首先是毫秒，是第一次运行基准测试花费了多少时间，因此通过默认 20 次运行得到真实的统计。第一次对于发现操作计算花费很有用，然后缓存数据
- 所有运行的操作的平均时间，毫秒
- 运行 op 占据的总时间的百分比。对于理解热点在哪里很有用。
- 之前所有表格中操作的时间。它对于理解层中什么分布式的工作，查看多少节点花费了大多数的时间很有用
- 节点的名字

第二个表格类似但是通过类似名称节点拆分时间，通过 op 的种类组合它们。这对于理解哪个操作实现想要优化或者从表中消除是很有用的。表格从开始排列多数费时的操作，仅仅显示前 10 个入口，对于其它的节点结合 placeholder，列从左往右是：

- 典型被分析的节点
- 这种类型所有节点的累计平均时间，毫秒
- 操作类型总共时间的多少比例被花费

- 比这个操作类型更高的累计时间花费，因此你可以理解 workload 的分布
- 这个输出的多少内存被占用

两个建立的表格使得你能很容易的复制和粘贴它们的结果到电子表格文件，因此它们被这个 tabs 作为列的分割输出。通过节点类型的总结当查找优化机会可能很有用，因为它是代码花费时间的关键点。在这种情况下，你可能看到 Conv2D 操作大约 90% 的执行时间。这是一个图优化的符号，因为卷积和矩阵乘作为神经网络的计算 workload 的 bulk。

正如经验指出，更多的考虑如果你查看一些其它的操作占据超过一个小时的比例。对于神经网络，通常不涉及大矩阵乘法通常应该比单个小，因此如果你的一些时间进入了这里，这意味着你的网络不是一个优化的结构，或者实现这些操作的代码没有优化到最优。如果你看到这些情况欢迎你查看[Performance bugs](#) 或者 patch，特别是如果你包含一个附件模型展示这个行为和命令行用于在基准测试工具上运行。

为了在桌面环境中运行下面，但是这个工具在 Android 上工作，这对于移动部署很有用，下面是在 arm 64bit 上运行示例的命令：

```

1 bazel build -c opt --config=android_arm64 \
2 tensorflow/tools/benchmark:benchmark_model
3 adb push bazel-bin/tensorflow/tools/benchmark/benchmark_model /data/local/tmp
4 adb push /tmp/tensorflow_inception_graph.pb /data/local/tmp/
5 adb shell '/data/local/tmp/benchmark_model \
6 --graph=/data/local/tmp/tensorflow_inception_graph.pb --input_layer="Mul" \
7 --input_layer_shape="1,299,299,3" --input_layer_type="float" \
8 --output_layer="softmax:0" --show_run_order=false --show_time=false \
9 --show_memory=false --show_summary=true'
```

你可以用和桌面版相同的方法解释上面的命令，如果你在找出输入输出名字和类型遇到了点麻烦，查看[Preparing models](#)页了解你的模型如何检测这些，查看 summarize\_graph 工具也许给你一些帮助信息。

这不支持 iOS 下的命令行，因此在[tensorflow/examples/ios/benchmark](#)中有一个分割的例子，包的功能和标准 app 的功能相同。输出统计设备屏幕和调试日志。如果你想为 Android 例子在屏幕上统计，你可以通过按压音量上键打开它们。

#### 4.17.14 探测你的 app

你通过 benchmark 工具查看的从模型生成的输出包含作为标准 TensorFlow 的一部分，这意味着你必须在你自己的应用中访问它们，你可以查看一个[例子](#)

基本的步骤是：

- 创建 StatSummarizer 对象`tensorflow::StatSummarizer stat_summarizer(tensorflow_graph);`

- 建立选项:

```

1 tensorflow::RunOptions run_options;
2 run_options.set_trace_level(tensorflow::RunOptions::FULL_TRACE);
3 tensorflow::RunMetadata run_metadata;
```

- 运行图:

```

1 run_status = session->Run(run_options, inputs, output_layer_names, {},
2                             output_layers, &run_metadata);
```

- 计算打印结果

```

1 assert(run_metadata.has_step_stats());
2 const tensorflow::StepStats& step_stats = run_metadata.step_stats();
3 stat_summarizer->ProcessStepStats(step_stats);
4 stat_summarizer->PrintStepStats();
```

### 4.17.15 可视化模型

多数高效的加速你的代码的方法是通过修改你的模型，因为它做很少的工作。做到这个你需要明白你的模型在做什么，可视化它是一个好的一步。为了获取高级图概览，使用[TensorBoard](#)

### 4.17.16 线程

TensorFlow 桌面环境有一个高级的线程模式，如果可以尝试运行多个操作。在我们的术语中这被称为”Inter-op parallelism”(考虑避免和”intra-op”弄混，你可以把它当做”between-op”)，你可以通过在会话选项中指定 `inter_op_parallelism_threads`。

默认，移动设备连续的运行操作，这是 `inter_op_parallelism_threads` 被设置为 1。移动处理器通常有一些核心很小的的 cache，因此运行多个互不影响在分开的内存区域对性能没有帮助。”Intra-op parallelism” 或者 (“Within-op”) 可能很有用，特别是对于计算受限的操作像卷积 (不同的线程可以输入相同的小的存储区)。

在移动端，一个操作默认多少线程将使用这些核心数，或者当核心不被决定的时候为 2。你可以在会话选项设置 `intra_op_parallelism_threads` 覆盖默认的线程数。如果你有自己的线程处理繁重的任务一个好的想法是减少默认线程数量，以至于它们不能相互影响。为了查看更多会话选项，查看[ConfigProto](#)

### 4.17.17 使用移动数据重新训练

在移动 app 上运行模型的一个最大的问题是精度不能代表训练数据。例如，多数 ImageNet 照片是 well-gramed 以至于对象在图片的中间，well-lit 和正规 lens 的 shot。手机设

备上的相片经常帧率低，badly lit，可能有鱼眼畸变。

解决方案是扩展从你的应用捕获的训练集。这步设计额外的工作，因此你将必须自己标记样本，但是即使你用它扩展你的原始的训练数据，它可能对训练数据集有帮助。为了通过做这个改善训练集合，通过修复其它的像复制或者差的标签样本修复其它的质量问题是最好的提升精度分方法。通常比使用不同的技术修改你的模型架构更有效。

#### 4.17.18 减少模型载入时间和内存 footprint

大多数的操作系统允许你用内存映射载入一个文件，而不是通过通常的 I/O API。相比于在堆上分配一个内存区域而是从磁盘复制字节进去，你简单的告诉操作系统使得整个文件的内容在内存中出现。这有一些好处：

- 载入速度
- 减少 page (增加性能)
- 不为你的 app 计数 RAM 预算

TensorFlow 也支持能从一些模型文件映射权重。因为在 ProtoBuf 序列化格式的限制，我们必须改变我们的模型载入和处理代码。这个内存映射工作是我们有一个文件，第一部分是正常 GraphDef 序列为 protocol buffer wire 格式，但是权重以一种方式添加以至于能被直接映射。为了创建这个文件，运行 tensorflow/contrib/util/converter\_graphdef\_memmapped\_format 工具。这接受在 GraphDef 文件通过 freeze\_graph 运行，转化它到权重被添加到最后的格式。因此文件不在是一个标准的 GraphDef protobuf，你需要做一些改变载入代码。你可以在 LoadMemoryMappedModel() 函数中 [iOS Cammera demo app](#) 查看这个例子。

同样的代码（结合 Object C 调用获取文件的过去文件的替代）可以用在其它的平台。因为我们用内存映射，我们需要通过创建一个特殊的 TensorFlow 环境对象建立文件我们将使用：

```

1 std::unique_ptr<tensorflow::MemmappedEnv> memmapped_env;
2 memmapped_env->reset(
3     new tensorflow::MemmappedEnv(tensorflow::Env::Default()));
4 tensorflow::Status mmap_status =
5     (memmapped_env->get())->InitializeFromFile(file_path);

```

你然后需要在这个环境传递子序列，像这样载入图：

```

1 tensorflow::GraphDef tensorflow_graph;
2 tensorflow::Status load_graph_status = ReadBinaryProto(
3     memmapped_env->get(),
4     tensorflow::MemmappedFileSystem::kMemmappedPackageDefaultGraphDef,
5     &tensorflow_graph);

```

你也需要创建会话结合一个指针到环境:

```

1 tensorflow::SessionOptions options;
2 options.config.mutable_graph_options()
3     ->mutable_optimizer_options()
4     ->set_opt_level(::tensorflow::OptimizerOptions::L0);
5 options.env = memmapped_env->get();
6
7 tensorflow::Session* session_pointer = nullptr;
8 tensorflow::Status session_status =
9     tensorflow::NewSession(options, &session_pointer);

```

这里有一件事需要注意，我们禁用了自动优化，因此在一些情况下，这将折叠子树，创建一个 tensor 值的复制（我们不想用更多的 RAM）

当你进行这些步的时候，你可以用 session 和 graph 作为 normal，你应该看到载入时间和内存使用减小。

#### 4.17.19 从简单的复制保护模型文件

默认，你的模型将从一个被存在磁盘上标准的序列化的 protobuf 中。理论上任何人可以复制你的模型，你也许不想这样。然而，实际上多数模型用于专业领域，有类似在竞争者反编译和重用你的代码的风险，但是如果你想使得它结合普通的用户访问你的文件时可以采用如下步骤。

我们多数代码使用[ReadBinaryProto\(\)](#)方便的调用从磁盘载入 GraphDef。这要求一个没有加密的 protobuf。幸运的，调用实现是相当直接应该很容易在内存中写一个等价可以加密的。这里一些代码显示你可以使用自己的加密环境如何读和加密一个 protobuf。

```

1 Status ReadEncryptedProto(Env* env, const string& fname,
2                             ::tensorflow::protobuf::MessageLite* proto) {
3     string data;
4     TF_RETURN_IF_ERROR(ReadFileToString(env, fname, &data));
5
6     DecryptData(&data); // Your own function here.
7
8     if (!proto->ParseFromString(&data)) {
9         TF_RETURN_IF_ERROR(stream->status());
10        return errors::DataLoss("Can't parse ", fname, " as binary proto");
11    }
12    return Status::OK();
13 }

```

为了使用这，你将需要定义一个 DecryptData() 函数。它可能像下面这样简单:

```
1 void DecryptData(string* data) {
```

```
2  for (int i = 0; i < data.size(); ++i) {  
3      data[i] = data[i] ^ 0x23;  
4  }  
5 }
```

你也许想要一些更复杂的，但是明确你将需要当前 scope 之外的什么？



# Chapter 5

## 扩展

这个章节解释开发者如何增加功能到 TensorFlow。

### 5.1 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，操作会话。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，在大规模分布式系统上实现机器学习系统或者是任何想要了解 TensorFlow 的人。读完它后你应该能读和修改 TensorFlow 核心代码。

### 5.2 概述

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

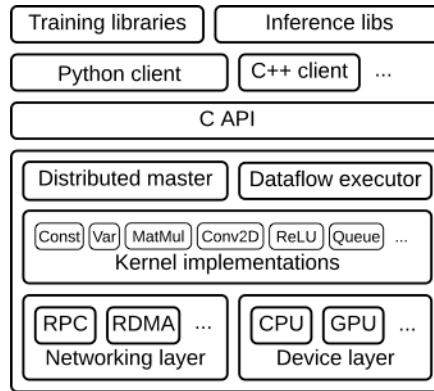


图 5.1: Tensorflow 框架

- Client
- 定义计算作为数据流图。
- 用 session 初始化图。
- Distributed Master
- 从图中获取一个子图作为定义的参数给 Session.run()
- 将图分成多块在不同的进程和设备上运行。
- 分配图块到 worker service。
- 用 worker services 初始化图块
- 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。

下图说明组件之间的交互。”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。

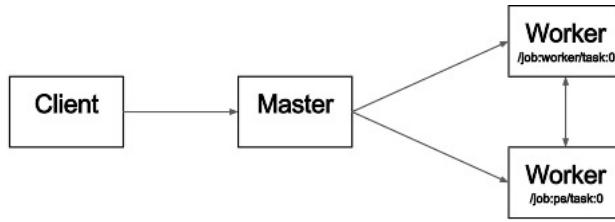


图 5.2: 交互

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。

### 5.2.1 Client

用户写 TensorFlow 程序构造计算图。这个程序既可以组成单个操作又可以用一个像 Estimators API 的方便的库组成神经网络乘和其它高级抽象。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它们。当特征被建立好后我们将它们接入 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。用户创建一个会话，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。

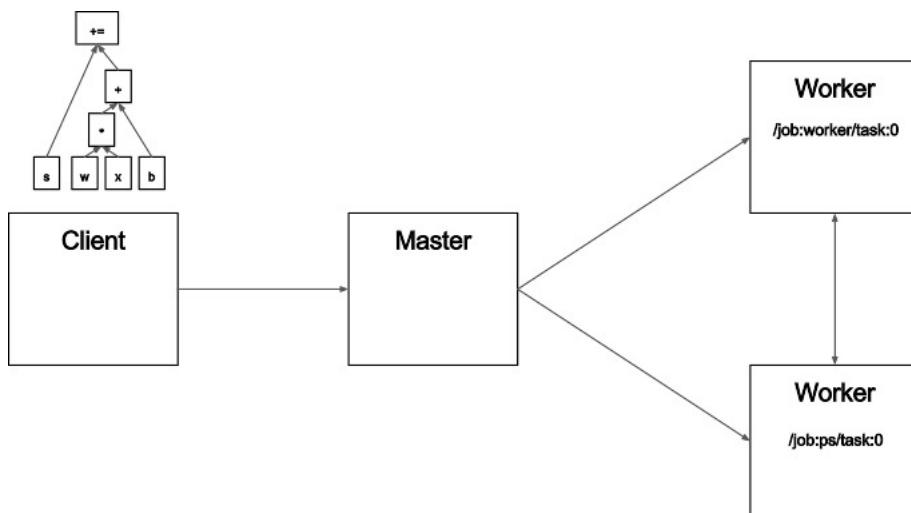


图 5.3: 分割子图

## 5.2.2 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。
- 缓存这些块以至于它们能用在自序列中。

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它们在参数服务器上。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递信息。下面的 distributed master 传输子图到分布的任务。

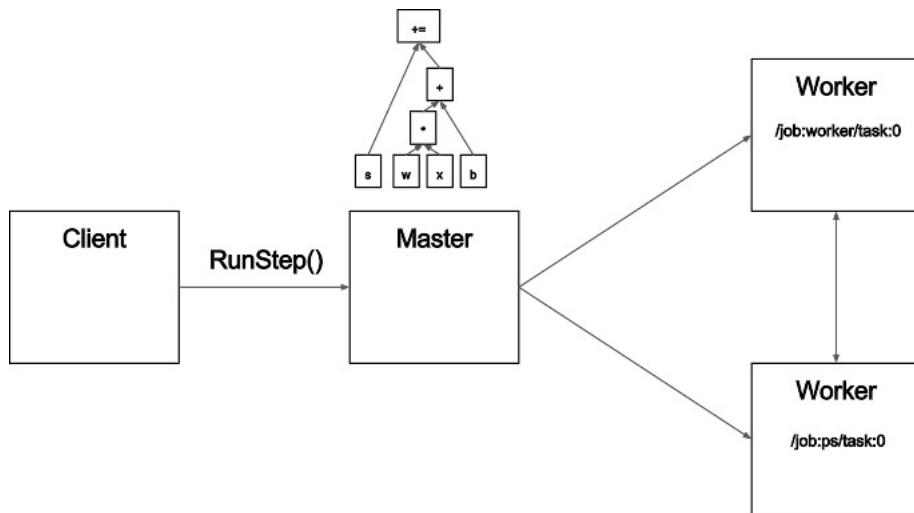


图 5.4: session 处理

下图 5 显示了我们例子图的一个可能分割。这个分布的 master 有组在一起的模型参数来防止他们在同一个 server 上。

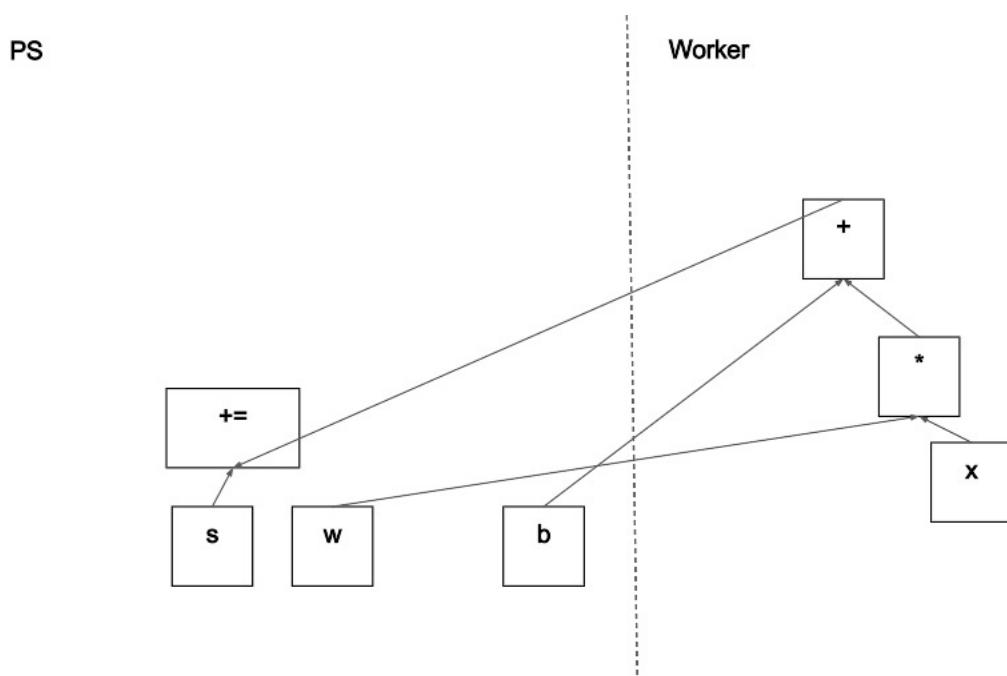


图 5.5: 图的分割

这里图的边被分割，分布的 master 插入发送接受节点在分布的 tasks 传递信息

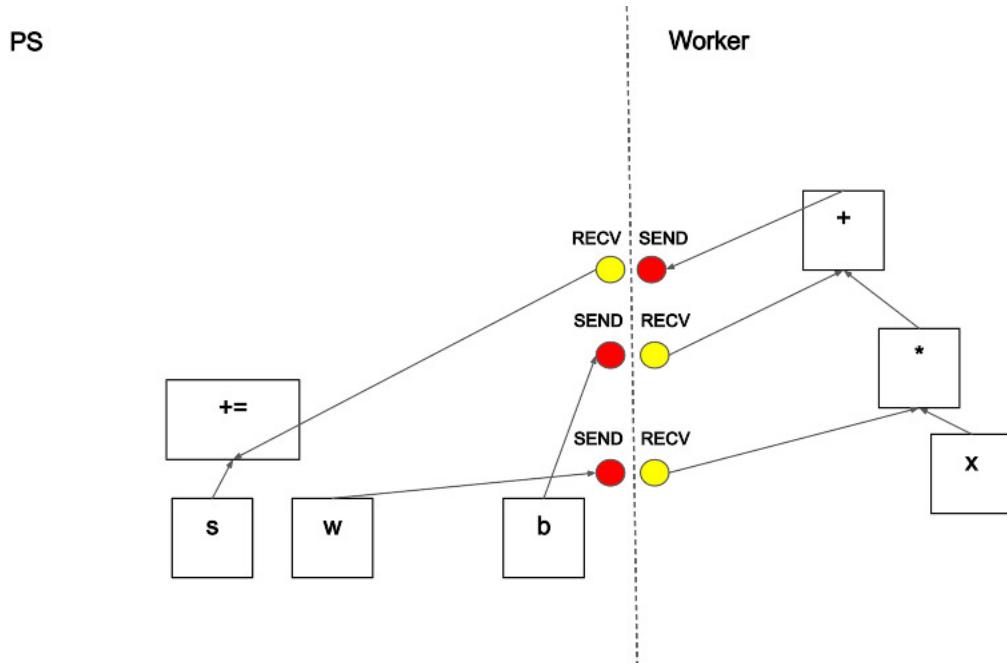


图 5.6: 消息传递

这个分布式的 master 传递子图到分布的 task

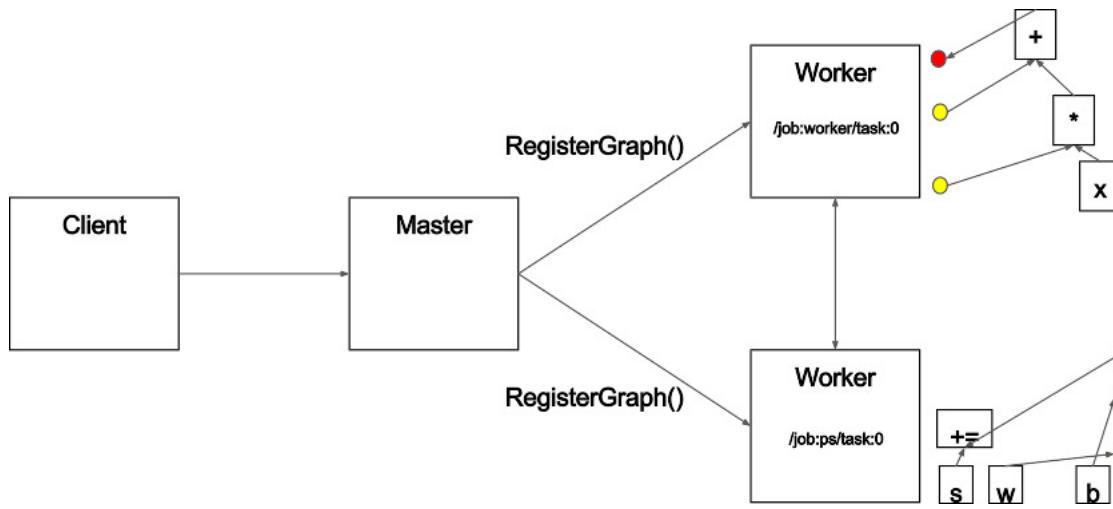


图 5.7: 子图传递到分布的 tasks

- [MasterService API definition](#)
- [Master interface](#)

### 5.2.3 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运行大的图。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对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：

- gPRC 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](#)

## 5.3 内核实现

运行环境包含超过 200 个标准操作包括数学，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，然而我们优先用 CuDNN 这类更高效内核实现的库。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 低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进程通过 C++ 提供更高效的实现，我们推荐你为一些重要的操作像 ReLU 和 Sigmoid 和相关的梯度注册你的融合内核，XLA 编译器有一些尝试实现实现自动内核融合。

### 5.3.1 代码

[OpKernel interface](#)

## 5.4 自定义文件读取器

要求：

- 熟悉 C++
- 必须下载[TensorFlow 源代码](#)，并能构建它。

分割支持文件格式的任务为两部分：

- 文件格式：我们用一个 Reader op 从一个文件读一个 record(可以为任何字符串)
- 记录格式：我们用解码解析操作转变一个字符串为 TensorFlow 可用的 tensor

例如，读一个[CSV 文件](#)，我通过用一个[an Op that parses CSV data from a line of text](#) 用 [a Reader for text files](#)

### 5.4.1 写一个 Reader 用于文件格式

一个 Reader 有时候从一个文件读取 record。有一些 Reader Ops 的例子已经构建进入 TensorFlow 了：

- [tf.TFRecordReader\(\(source in kernels/tf\\_record\\_reader\\_op.cc\)\)](#)
- [tf.FixedLengthRecordReader\(source in kernels/fixed\\_length\\_record\\_reader\\_op.cc\)](#)
- [tf.TextLineReader\(source in kernels/text\\_line\\_reader\\_op.cc\)](#)

你可以查看这些相同接口的所有的 expose, 唯一不同的是在他们的结构体中。最重要的方法是 read。最重要的方法是 read。它接受一个队列参数 (无论什么时候需要从这里获取文件, 例如当 read 操作第一次运行时, 之前的 read 从文件的最后读取), 它产生两个标量 tensor: 一个字符串 key 和一个字符串 value。调用 SomeReader 创建一个新的 SomeReader, 你将需要:

- 在 C++ 中, 调用 SomeReader 定义一个子类[tensorflow::ReaderBase](#)
- 在 C++ 中, 注册一个新的 reader 操作和名字为 SomeReader 的核心
- 在 Python 中, 调用 SomeReader 定义一个子类[tf.ReaderBase](#)

你可以仿照所有的 C++ 代码在 `tensorflow/core/user_ops/some_reader_op.cc` 中。这代码读一个文件将进入一个 C++ ReaderBase 类的后代, 这个类定义在 [tensorflow/core/kernel-reader\\_base.h](#)。你将需要实现下面方法:

- `OnWorkStartedLocked`: 打开下一个文件
- `ReadLocked`: 读一个 record 报告一个 EOF/error
- `OnWorkFinishedLocked`: 关闭当前文件
- `ResetLocked`: 获取一个干净的 slate, 例如一个错误

这些方法的名称都以 Locked 结尾, 因为 Readerbase 确保调用这些方法中的一个是获得一个互斥, 因此你通常不用单线程安全 (尽管只有类的保护成员, 没有全局状态) 对于 `OnWorkStartedLocked`, 文件的名字打开是 `current_work()` 方法返回的, `ReadLocked` 有这 signature: `Status ReadLocked(string* key, string* value, bool* produced, bool* at_end)` 如果 `ReadLocked` 成功的从一个文件读取一个 record, 它应该填充:

- `*key`: 结合对 record 的一个识别, 人能用于再次找到这个 record, 你可以从 `current_work()` 包含文件名, 添加一个记录号或者无论什么。
- `*value`: 结合 record 的内容
- `*produced`: 设置为 true

如果你达到的文件的结尾 (EOF), 设置 `*at_end` 为 true。在这种情况下, 染回 `Status::OK()`。如果有一个错误, 简单的用[tensorflow/core/lib/core/errors.h](#) 返回他, 不修改任何参数。

下一步, 你将创建一个实际的 Reader op。如果你知道[the adding an op how-to](#), 这将很有好处。主要步骤:

- 注册一个 op

- 定义一个 OpKernel

为了注册一个 op, 你将需要用定义在[tensorflow/core/framework/op.h](#)REGISTER\_OP 调用。Reader op 从来不接受任何输入总是有一个 resource 单个输出。他们应该有字符串 container 和 shared\_name 属性。你也许选择定义额外的属性配置或者包含在 Doc 中的文档。例如查看[tensorflow/core/ops/io\\_ops.cc](#)

```

1 #include "tensorflow/core/framework/op.h"
2
3 REGISTER_OP("TextLineReader")
4     .Output("reader_handle: resource")
5     .Attr("skip_header_lines: int = 0")
6     .Attr("container: string = ''")
7     .Attr("shared_name: string = ''")
8     .SetIsStateful()
9     .SetShapeFn(shape_inference::ScalarShape)
10    .Doc(R"doc(
11 A Reader that outputs the lines of a file delimited by '\n'.
12 )doc");

```

为了定义一个 OpKernel。Reader 能用从 ReaderOpKernel([tensorflow/core/framework/reader\\_op\\_kernel.h](#)) 继承的 shortcut, 调用 SetReaderFactory 实现一个构造体。在定义你的类后, 你将需要用 REGISTER\_KERNEL\_BUILDER 注册它。一个没有属性的例子:

```

1 #include "tensorflow/core/framework/reader_op_kernel.h"
2
3 class TFRecordReaderOp : public ReaderOpKernel {
4 public:
5     explicit TFRecordReaderOp(OpKernelConstruction* context)
6         : ReaderOpKernel(context) {
7         Env* env = context->env();
8         SetReaderFactory([this, env]() { return new TFRecordReader(name(), env); });
9     }
10 };
11
12 REGISTER_KERNEL_BUILDER(Name("TFRecordReader").Device(DEVICE_CPU),
13                         TFRecordReaderOp);

```

一个有属性的例子:

```

1 #include "tensorflow/core/framework/reader_op_kernel.h"
2
3 class TextLineReaderOp : public ReaderOpKernel {
4 public:
5     explicit TextLineReaderOp(OpKernelConstruction* context)

```

```

6   : ReaderOpKernel(context) {
7     int skip_header_lines = -1;
8     OP_REQUIRES_OK(context,
9                     context->GetAttr("skip_header_lines", &skip_header_lines));
10    OP_REQUIRES(context, skip_header_lines >= 0,
11                errors::InvalidArgument("skip_header_lines must be >= 0 not ",
12                                         skip_header_lines));
13    Env* env = context->env();
14    SetReaderFactory([this, skip_header_lines, env]() {
15      return new TextLineReader(name(), skip_header_lines, env);
16    });
17  };
18};
19
20 REGISTER_KERNEL_BUILDER(Name("TextLineReader").Device(DEVICE_CPU),
21                         TextLineReaderOp);

```

最后一步是添加 Python 包装器。你可以通过[compiling a dynamic library](#) 做到或者如果你从源文件构建 TensorFlow, 添加到 user\_op.py。后面的, 你将在[tensorflow/python/user\\_ops/user\\_ops.py](#)导入 tensorflow.python.ops.io\_ops 添加[io\\_ops.ReaderBase](#)继承。

```

1 from tensorflow.python.framework import ops
2 from tensorflow.python.ops import common_shapes
3 from tensorflow.python.ops import io_ops
4
5 class SomeReader(io_ops.ReaderBase):
6
7     def __init__(self, name=None):
8         rr = gen_user_ops.some_reader(name=name)
9         super(SomeReader, self).__init__(rr)
10
11 ops.NotDifferentiable("SomeReader")

```

你可以在[tensorflow/python/ops/io\\_ops.py](#)查看一些例子。

### 5.4.2 写一个操作用于记录格式

通常这是一个普通的接受一个标量字符串创记录作为输入的操作, 因此下面的[the instructions to add an Op](#)。你需要选择接受一个标量字符串 key 作为输入, 包含报告不合适格式数据的在输入错误消息。这种方法用户可以更容易记录坏数据来自哪里。Ops 的用于 Record 的例子:

- [tf.parse\\_single\\_example](#)

- `tf.decode_csv`
- `tf.decode_raw`

注意用多个操作解码类似格式是它可能很有用。例如，你也许有一张图片作为字符串保存在`tf.train.Example` protocol buffer。依赖于图片的格式，你将从`tf.parse_single_example`操作调用`tf.image.decode_jpeg`和`tf.image.decode_png`或者是`tf.decode_raw`得到对应的输出。通常使用`tf.slice`和`tf.reshape`得到 `tf.decode_raw` 部分输出。

## 5.5 用 tf.estimator 创建 Estimator

`tf.estimator` 框架使得通过他的高级 Estimator API 构造和训练机器学习模型变得很容易。Estimator 提供你能快速配置昌建模型类型的想 regressors 和 classifiers 类快速实例化。

- `tf.estimator.LinearClassifier`构造一个线性分类器模型
- `tf.estimator.LinearRegressor`构造一个线性回归模型
- `tf.estimator.DNNClassifier`构造一个神经网络分类器
- `tf.estimator.DNNRegressor`构造一个神经网络和线性结合的分类模型
- `tf.estimator.DNNLinearCombinedClassifier`构造一个神经网络和线性结合的回归模型
- `tf.estimator.DNNLinearCombinedRegressor`

但是如果 `tf.estimator` 中没有一个预定义的模型满足你的需要怎么办？也许你需要在模型配置上进行更加精细的配置，像为优化器自定义损失函数，或者为不同的神经层指定不同的激活函数。或者也许你正在实现一个排序或者推荐系统或者生成的预测既不分类也不回归。

这个导航包含如何通过使用 `tf.estimator` 提供的构建模块创建自己的 Estimator，基于他们的物理度量预测`abalone`年龄。你将学习：

- 实例化一个 Estimator
- 构造一个自定义的模型函数
- 用 `tf.feature_column` 和 `tf.layers` 配置神经网络
- 从 `tf.losses` 选择一个合适的损失函数
- 为你的模型定义一个训练操作
- 生成和返回预测

### 5.5.1 预先要求

这个导航需要你知道基础的 tf.estimator API, 像 feature columns, 输入函数和 train(), evaluate(), predict() 操作。如果你之前没有过 tf.estimator, 你应该首先查看下面的导航。

- [tf.estimator Quickstart](#) 用 tf.estimator 训练神经网络的快速介绍
- [TensorFlow Linear Model Tutorial](#) 介绍 feature columns, 和用 tf.estimator 构建一个线性分类器概述
- [Building Input Functions with tf.estimator](#) 如何构造一个 input\_fn 处理和输入数据到你的模型的概览

### 5.5.2 一个 Abalone 年龄预测器

通过壳上的环估计[abalon](#)(sea snail) 的年龄是可能的, 然而, 在显微镜下查看壳, 这个任务要求严且可能被污染, 希望能找到另一个测量方法预测年纪。

[Abalone Data Set](#) 包含关于 abalone 的下面特征数据

特征	描述
长度	abalone 的长度 (最长的方向, 单位 mm)
直径	abalone 的周长 (垂直方向上的长度单位 mm)
高度	abalone 的高度 (肉在壳中, 单位 mm)
整体重量	abalone 的整体重量 (单位 g)
去壳后的重量	肉的重量 (单位 g)
内脏重量	流血后的重量 (单位 g)
壳重量	abalone 壳的重量 (单位 g)

表 5.1: 特征信息

标签预测环数作为 abalone 的年龄。



图 5.8: Abalone shell (by Nicki Dugan Pogue, CC BY-SA 2.0)

### 5.5.3 开始

这个导航用三个数据集, `abalone_train.csv` 包含训练数据 3320 样本。`abalone_test.csv` 包含标记测试数据 850 个样本。`abalone_predict` 包含 7 个预测样本。下面的章节一步步写 Estimator 代码, 完整的代码在[这里](#)

### 5.5.4 载入 abalone csv 数据到 TensorFlow 数据集

为了输入数据进 model, 你讲需要下载 cvs 文件载入到 TensorFlow Dataset。首先添加一些标准的 Python 和 TensorFlow 导入, 设置 FLAGS:

```
1 from __future__ import absolute_import
2 from __future__ import division
3 from __future__ import print_function
4
5 import argparse
6 import sys
7 import tempfile
8
9 # Import urllib
10 from six.moves import urllib
11
12 import numpy as np
13 import tensorflow as tf
14
```

```
15 FLAGS = None
```

开启采集tf.logging.set\_verbosity(tf.logging.INFO)

然后定义一个函数载入 CSV 文件:

```
1 def maybe_download(train_data, test_data, predict_data):
2     """Maybe downloads training data and returns train and test file names."""
3     if train_data:
4         train_file_name = train_data
5     else:
6         train_file = tempfile.NamedTemporaryFile(delete=False)
7         urllib.request.urlretrieve(
8             "http://download.tensorflow.org/data/abalone_train.csv",
9             train_file.name)
10    train_file_name = train_file.name
11    train_file.close()
12    print("Training data is downloaded to %s" % train_file_name)
13
14    if test_data:
15        test_file_name = test_data
16    else:
17        test_file = tempfile.NamedTemporaryFile(delete=False)
18        urllib.request.urlretrieve(
19            "http://download.tensorflow.org/data/abalone_test.csv", test_file.name)
20        test_file_name = test_file.name
21        test_file.close()
22        print("Test data is downloaded to %s" % test_file_name)
23
24    if predict_data:
25        predict_file_name = predict_data
26    else:
27        predict_file = tempfile.NamedTemporaryFile(delete=False)
28        urllib.request.urlretrieve(
29            "http://download.tensorflow.org/data/abalone_predict.csv",
30            predict_file.name)
31        predict_file_name = predict_file.name
32        predict_file.close()
33        print("Prediction data is downloaded to %s" % predict_file_name)
34
35    return train_file_name, test_file_name, predict_file_name"
```

最后创建 main() 载入 csv 文件进入 Datasets, 定义 flags 允许用户通过命令行选择制定 CSV 文件训练, 测试和预测数据集

```
1 def main(unused_argv):
```

```

2 # Load datasets
3 abalone_train, abalone_test, abalone_predict = maybe_download(
4     FLAGS.train_data, FLAGS.test_data, FLAGS.predict_data)
5
6 # Training examples
7 training_set = tf.contrib.learn.datasets.base.load_csv_without_header(
8     filename=abalone_train, target_dtype=np.int, features_dtype=np.float64)
9
10 # Test examples
11 test_set = tf.contrib.learn.datasets.base.load_csv_without_header(
12     filename=abalone_test, target_dtype=np.int, features_dtype=np.float64)
13
14 # Set of 7 examples for which to predict abalone ages
15 prediction_set = tf.contrib.learn.datasets.base.load_csv_without_header(
16     filename=abalone_predict, target_dtype=np.int, features_dtype=np.float64)
17
18 if __name__ == "__main__":
19     parser = argparse.ArgumentParser()
20     parser.register("type", "bool", lambda v: v.lower() == "true")
21     parser.add_argument(
22         "--train_data", type=str, default="", help="Path to the training data.")
23     parser.add_argument(
24         "--test_data", type=str, default="", help="Path to the test data.")
25     parser.add_argument(
26         "--predict_data",
27         type=str,
28         default="",
29         help="Path to the prediction data.")
30     FLAGS, unparsed = parser.parse_known_args()
31     tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```

### 5.5.5 实例化一个 Estimator

当使用一个 `tf.estimator` 提供的类定义一个模型（想 `DNNClassifier`）的时候，你在结构体中应用正确的配置参数。

```

1 my_nn = tf.estimator.DNNClassifier(feature_columns=[age, height, weight],
2                                     hidden_units=[10, 10, 10],
3                                     activation_fn=tf.nn.relu,
4                                     dropout=0.2,
5                                     n_classes=3,
6                                     optimizer="Adam")

```

你不需要写任何代码说明 TensorFlow 如何训练模型，计算损失，返回预测；这些逻辑已经写入的 DNNClassifier。

通过对比，当你创建你自己的 estimator 的时候，构造体接受两个高级参数用于模型的配置，model\_fn 和 parms：nn = tf.estimator.Estimator(model\_fn=model\_fn, params=model\_params)

- model\_fn: 一个包含所有的上面提到的逻辑的函数对象支持训练，估计和预测。你只管实现功能，下一章节[Constructing the model\\_fn](#)构造包含创建一个模型函数的细节。
- params: 一个可选的词典超参数（学习率， dropout）等将被传输进 model\_fn。

**注意：**像 tf.estimator 的预先定义的回归和分类器一样，Estimator 初始化器也接受通常的配置参数 model\_dir 和 config

对于 abalone 预测器，模型将接受一个超参数：学习率。在你的代码的开头定义 LEARNING\_RATE 作为，之后配置采集：

```
1 tf.logging.set_verbosity(tf.logging.INFO)
2
3 # Learning rate for the model
4 LEARNING_RATE = 0.001
```

**注意：**这里的 LEARNING\_RATE 设置为 0.001，但是当你需要更好的结果时你可以在训练过程中改变这个值

然后，添加下面代码到 main()，创建字典 model\_params 包含学习率和实例化的 Estimator：

```
1 # Set model params
2 model_params = {"learning_rate": LEARNING_RATE}
3
4 # Instantiate Estimator
5 nn = tf.estimator.Estimator(model_fn=model_fn, params=model_params)
```

### 5.5.6 构造 model\_fn

Estimator API 的基本的框架像这样：

```
1 def model_fn(features, labels, mode, params):
2     # Logic to do the following:
3     # 1. Configure the model via TensorFlow operations
4     # 2. Define the loss function for training/evaluation
5     # 3. Define the training operation/optimizer
6     # 4. Generate predictions
7     # 5. Return predictions/loss/train_op/eval_metric_ops in EstimatorSpec object
8     return EstimatorSpec(mode, predictions, loss, train_op, eval_metric_ops)
```

model\_fn 必须接受三个参数:

- features: 一个包含通过 input\_fn 传递待模型的包含特征的字典
- labels: 一个 Tensor 包含通过 input\_fn 传递到模型的标签。当 model 自己推理的时候将出现空的调用
- mode: 一个`tf.estimator.ModeKeys`字符串值指明 model\_fn 被调用的上下文:
  - `tf.estimator.ModeKeys.TRAIN` model\_fn 调用在训练模式下, 也就是通过 `train()` 调用
  - `tf.estimator.ModeKeys.EVAL` model\_fn 在估计模式下调用, 也就是通过 `evaluate()` 调用
  - `tf.estimator.ModeKeys.PREDICT` model\_fn 在预测模式下调用, 也就是通过 `predict()` 调用

model\_fn 也接受一个包含用于训练的超参数参数 (正如上面框架介绍的)

函数执行的主题跟着下面的任务 (更多的细节查看下面的章节)

- 配置模型, 对于 abalone 预测器, 浙江是一个神经网络
- 定义用于计算预测结果和目标值接近程度的损失函数。
- 定义一个训练操作制定优化算法最小化损失值的计算

model\_fn 必须返回一个`tf.estimator.EstimatorSpec`对象, 包含下面的值:

- mode(要求)。模型运行的模式, 通常你将返回 model\_fn 的 mode 参数
- predictions(要求在 PREDICT 模式下)。一个字典映射你的选择的名字为包含模型预测的 Tensor。例如
  - 1 在 PREDICT 模式, 你在 EstimatorSpec 返回的字典将通过 predict() 返回, 因此你可以用你想使用的方式构造它
  - 2 \item loss (要求 EVAL 和 TRAIN 模式)。一个包含有损失标量值的 Tensor: 模型损失函数 (更深的讨论在 [{ Defining loss for the model }](https://www.tensorflow.org/extend/estimators#defining_loss)) 在输入计算后的输出。这用于 TRAIN 模式用于处理错误和采集, 自动作为度量包含在 EVAL 模式。
  - 3 \item train\\_op: (仅仅要求在 TRAIN 模式)。一个返回训练步数的操作
  - 4 \item eval\\_metric\\_ops (可选)。一个 name/value 对制定当模型在 EVAL 模式下运行时制定将被计算的度量。你的选择的标签的名字用于度量, 和你的度量计算的结果。[{ tf.metrics }](https://www.tensorflow.org/api_docs/python/tf/metrics) 模块提供预定义的函数用于多种常规测量。

```

5   下面的 eval\_metric\_ops 包含一个用 tf.metrics.accuracy 计算的 accuracy 方法
6   :\begin{lstlisting}[language=Bash]{\}
7     python eval_metric_ops = { "accuracy": tf.metrics.accuracy(labels,
predictions) }\}

```

如果你没有指定 eval\_metric\_ops, 仅仅 loss 将在估计的时候被计算。

### 5.5.7 结合 tf.feature\_column 和 tf.layers 配置神经网络

构造一个神经网络需要创建和连接输入层, 隐藏层和输出层。

输入层是一系列的节点 (在模型中的一个特征), 节点通过在 features 参数接受特征输入传入 model\_fn。如果 features 包含一个包含你的特征数据的 n 维 Tensor, 然后他可能为输入层服务。如果 features 包含一个 feature columns 通过输入函数人传递给模型, 你可以结合 `tf.feature_column.input_layer` 函数转化输入层 Tensor。

```

1 input_layer = tf.feature_column.input_layer(
2   features=features, feature_columns=[age, height, weight])

```

正如上面显示的, `input_layer()` 接受两个参数:

- features: 一个从字符串到包含对应 feature 数据的 Tensor 映射, 它在 model\_fn 中的 features 参数被明确的传递
- feature\_columns: 一个模型中所有的 FeatureColumns 的列表 (年龄, 高度, 重量 (上面例子))

神经网络的输入层必须通过一个 activation function 连接到一个或者更多的隐藏层对前面层的数据执行非线性变换。最后的隐藏层然后连接最后一层的输出层。tf.layers 提供 `tf.layers.dense` 函数构造全连接层。激活函数通过 activation 参数控制, 一些选项传递给 activation 参数:

- `tf.nn.relu` 下面的代码通过 `ReLU activation function(tf.nn.relu)` 创建一个全连接到前一层 `input_layer` 的层
- `tf.nn.relu` 后面的代码通过 `ReLU6` 激活函数创建一个和 `hidden_layer` 的全连接层全连接单元层 (`tf.nn.relu`)
- None 下面的代码没有激活函数创建一个和前一层 `second_hidden_layer` 全连接的单元层, 仅仅是一个线性变换

```

1   python output_layer = tf.layers.dense( inputs=second_hidden_layer,
2                                         units=3, activation=None)

```

其他可能的激活函数，例如

```
1 output_layer = tf.layers.dense(inputs=second_hidden_layer,
2                                units=10,
3                                activation_fn=tf.sigmoid)
```

上面的代码创建一个用 sigmoid 激活函数 (`tf.sigmoid`) 和 `second_hidden_layer` 全连接的 `output_layer`, 更多预定义函数的细节查看[API docs](#) 将它们放在一起, 下面的代码构造一个完整的神经网络用于 Abalone 预测器和捕获他的预测:

```
1 def model_fn(features, labels, mode, params):
2     """Model function for Estimator."""
3
4     # Connect the first hidden layer to input layer
5     # (features["x"]) with relu activation
6     first_hidden_layer = tf.layers.dense(features["x"], 10, activation=tf.nn.relu)
7
8     # Connect the second hidden layer to first hidden layer with relu
9     second_hidden_layer = tf.layers.dense(
10         first_hidden_layer, 10, activation=tf.nn.relu)
11
12    # Connect the output layer to second hidden layer (no activation fn)
13    output_layer = tf.layers.dense(second_hidden_layer, 1)
14
15    # Reshape output layer to 1-dim Tensor to return predictions
16    predictions = tf.reshape(output_layer, [-1])
17    predictions_dict = {"ages": predictions}
18    ...
```

这里因为你将用 `numpy_input_fn` 传递 abalone 的 Datasets, `features` 是一个字典'X':`data_tensor`, 因此 `features["x"]` 是输入层, 网络包含两个隐藏层, 每个层有 10 个结合 ReLU 激活函数的节点。输出层没有激活函数`tf.reshape`一个一维 tensor 捕获模型的预测存储在 `predictions_dict`。

### 5.5.8 为模型定义一个损失

`model_fn` 返回的 `EstimatorSpec` 必须包含 `loss`: 一个代表 loss 值的 Tensor, 用来衡量模型在训练和评估运行时预测反映的结果的好坏。`tf.losses`模块提供方便的函数用多种度量计算损失, 包括:

- `absolute_difference(labels,predictions)` 使用[absolute-difference formula](#)(正如  $L_1$  损失) 计算损失
- `log_loss(labels,predictions)` 用[logistic loss formula](#) 计算损失函数 (通常使用逻辑回归)

- mean\_squared\_error(labels,predictions) 使用mean squared error计算损失 ( $L_2$  损失)

下面的例子使用 mean\_squared\_error() 添加一个定义的 loss 到 abalone 的 model\_fn

```

1 def model_fn(features, labels, mode, params):
2     """Model function for Estimator."""
3
4     # Connect the first hidden layer to input layer
5     # (features["x"]) with relu activation
6     first_hidden_layer = tf.layers.dense(features["x"], 10, activation=tf.nn.relu)
7
8     # Connect the second hidden layer to first hidden layer with relu
9     second_hidden_layer = tf.layers.dense(
10         first_hidden_layer, 10, activation=tf.nn.relu)
11
12     # Connect the output layer to second hidden layer (no activation fn)
13     output_layer = tf.layers.dense(second_hidden_layer, 1)
14
15     # Reshape output layer to 1-dim Tensor to return predictions
16     predictions = tf.reshape(output_layer, [-1])
17     predictions_dict = {"ages": predictions}
18
19     # Calculate loss using mean squared error
20     loss = tf.losses.mean_squared_error(labels, predictions)
21     ...

```

查看[API guide](#)了解更多 loss 函数的使用和支持的参数。

丰富的估计度量可以被添加到一个 eval\_metric\_ops 字典。下面的代码定义一个 rmse 度量，计算模型预测的均方误差。注意 labels tensor 转化一个 float64 类型为 predictions 匹配的类型的 tensor，包含真的值：

```

1 eval_metric_ops = {
2     "rmse": tf.metrics.root_mean_squared_error(
3         tf.cast(labels, tf.float64), predictions)
4 }

```

### 5.5.9 定义为 model 训练操作

训练操作定义优化算法 TensorFlow 将你拟合模型到训练数据上是使用。通常当训练的时候，目标是最小化误差。一个简单的方法创建训练操作是实例化一个 tf.train.Optimizer 子类和调用 minimize 方法。下面的代码为 abalone 的 model\_fn 定义一个训练操作使用损失值计算[Defining Loss for the Model](#)，学习率传递到 params 中的函数，梯度下降优化器。对于 global\_step，方便的函数[tf.train.get\\_global\\_step](#)考虑生成一个整数变量。

```

1 optimizer = tf.train.GradientDescentOptimizer(
2     learning_rate=params["learning_rate"])
3 train_op = optimizer.minimize(
4     loss=loss, global_step=tf.train.get_global_step())

```

对于优化器的完整列表查看[API guid](#)

### 5.5.10 完整的 abalone model\_fn

这里最终，完整的 model\_fn 用于 abalone 年龄预测器。下面的代码配置神经网络，定义损失和训练操作；返回一个 EstimatorSpec 对象包含 mode, predictions\_dict, loss 和 train\_op：

```

1 def model_fn(features, labels, mode, params):
2     """Model function for Estimator."""
3
4     # Connect the first hidden layer to input layer
5     # (features["x"]) with relu activation
6     first_hidden_layer = tf.layers.dense(features["x"], 10, activation=tf.nn.relu)
7
8     # Connect the second hidden layer to first hidden layer with relu
9     second_hidden_layer = tf.layers.dense(
10         first_hidden_layer, 10, activation=tf.nn.relu)
11
12    # Connect the output layer to second hidden layer (no activation fn)
13    output_layer = tf.layers.dense(second_hidden_layer, 1)
14
15    # Reshape output layer to 1-dim Tensor to return predictions
16    predictions = tf.reshape(output_layer, [-1])
17
18    # Provide an estimator spec for 'ModeKeys.PREDICT'.
19    if mode == tf.estimator.ModeKeys.PREDICT:
20        return tf.estimator.EstimatorSpec(
21            mode=mode,
22            predictions={"ages": predictions})
23
24    # Calculate loss using mean squared error
25    loss = tf.losses.mean_squared_error(labels, predictions)
26
27    # Calculate root mean squared error as additional eval metric
28    eval_metric_ops = {
29        "rmse": tf.metrics.root_mean_squared_error(
30            tf.cast(labels, tf.float64), predictions)
31    }

```

```

32
33     optimizer = tf.train.GradientDescentOptimizer(
34         learning_rate=params["learning_rate"])
35     train_op = optimizer.minimize(
36         loss=loss, global_step=tf.train.get_global_step())
37
38 # Provide an estimator spec for `ModeKeys.EVAL` and `ModeKeys.TRAIN` modes.
39     return tf.estimator.EstimatorSpec(
40         mode=mode,
41         loss=loss,
42         train_op=train_op,
43         eval_metric_ops=eval_metric_ops)

```

### 5.5.11 运行 Abalone 模型

你已经为 abalone 年龄预测器初始化一个 Estimator 在 model\_fn 中定义它的行为；所有需要做的是训练，估计，和预测。下面的代码是 main() 的结尾，拟合神经网络训练数据和估算精度。

```

1 train_input_fn = tf.estimator.inputs.numpy_input_fn(
2     x={"x": np.array(training_set.data)},
3     y=np.array(training_set.target),
4     num_epochs=None,
5     shuffle=True)
6
7 # Train
8 nn.train(input_fn=train_input_fn, steps=5000)
9
10 # Score accuracy
11 test_input_fn = tf.estimator.inputs.numpy_input_fn(
12     x={"x": np.array(test_set.data)},
13     y=np.array(test_set.target),
14     num_epochs=1,
15     shuffle=False)
16
17 ev = nn.evaluate(input_fn=test_input_fn)
18 print("Loss: %s" % ev["loss"])
19 print("Root Mean Squared Error: %s" % ev["rmse"]))

```

**注意：**上面的代码用输入函数输入 features(x) 和 label(y) Tensor 到模型中训练 (train\_input\_fn) 和估计 test\_input\_fn，了解更多输入函数，查看[Building Input Functions with tf.EstimatorSpec](#)

然后运行代码，你应该看到类似下面的输出：

```

1 ...
2 INFO:tensorflow:loss = 4.86658, step = 4701
3 INFO:tensorflow:loss = 4.86191, step = 4801
4 INFO:tensorflow:loss = 4.85788, step = 4901
5 ...
6 INFO:tensorflow:Saving evaluation summary for 5000 step: loss = 5.581
7 Loss: 5.581

```

损失得分被报告是当运行 ABALONE\_TEST 数据集来自 model\_fn 的军方误差。为 ABALONE\_PREDICT 数据集预测年龄，添加下面代码到 main()：

```

1 # Print out predictions
2 predict_input_fn = tf.estimator.inputs.numpy_input_fn(
3     x={"x": prediction_set.data},
4     num_epochs=1,
5     shuffle=False)
6 predictions = nn.predict(input_fn=predict_input_fn)
7 for i, p in enumerate(predictions):
8     print("Prediction %s: %s" % (i + 1, p["ages"])))

```

这里 predict() 函数迭代的返回结果在 predictions。for 训练返回结果，返回代码类似下面：

```

1 ...
2 Prediction 1: 4.92229
3 Prediction 2: 10.3225
4 Prediction 3: 7.384
5 Prediction 4: 10.6264
6 Prediction 5: 11.0862
7 Prediction 6: 9.39239
8 Prediction 7: 11.1289

```

### 5.5.12 附加资源

祝贺你，你已经从 Estimator 成功地构建了一个 tf.estimator。更多关于构建 Estimator 的资料查看下面的 API 章节：

- [Layers](#)
- [Losses](#)
- [Optimization](#)

## 5.6 TensorFlow 用其他语言

### 5.6.1 背景

这个文档是针对那些想要在其他语言中创造或者开发 TensorFlow 功能的开发者。它描述了 TensorFlow 的特性和在其它编程语言中使用的推荐步骤。

Python 是 TensorFlow 支持的首选语言，当前支持的特性最多。更多的函数正在被移植进 TensorFlow (C++ 实现) 核心通过[C API](#)，客户端语言应该用语言的[外部函数接口](#)调用这个[C API](#)提供 TensorFlow 函数。

### 5.6.2 概览

在其它编程语言中提供 TensorFlow 功能可能分为下面多种情况：

- 运行一个预先定义好的图：给一个 GraphDef(或者 MetaGraphDef)protocol 消息，能创建一个绘画，运行查询得到 tensor 结果。对于想上手机 app 或者服务器来说通过预先训练好的模型推理足够的
- 图接口：每定义一个 TensorFlow 操作至少定义一个函数或者添加一个操作到图上。理想的这些函数将被自动生成因此他们保持同步直到操作的定义被修改
- 梯度 (AKA 自动微分)：给一个图和输入输出操作列表，添加操作到图上计算输出与对应输入的偏微分 (梯度)。对于图上的类似操作允许定义梯度函数
- 函数：定义一个在 GraphDef 中能被多处调用的子图。在 FunctionDefLibrary 定义一个 FunctionDef 包含在 GraphDef
- 控制流：如果用户指定子图，构造 if 和 while。最好的情况是他们和梯度一起工作
- 神经网络库：一些组件结合在一起支持创建神经网络模型和训练他们 (可能是分布式设置)。在其它语言有这些将很方便，当前没有计划支持 Python 外的其他语言。这些库同上面这些特性的包装器

在一个小的，语言绑定应该支持运行一个预先定义的图，但是多数应该支持图的构建。TensorFlow Python API 提供所有的这些特性。

### 5.6.3 当前状态

新的语言支持应该建立的[C API](#)之上。然而，正如下表所见，不是所有的函数在 C 中都可用。为[C API](#)提供更多支持的计划正在进行

#### 推荐方法

特性	Python	C
运行一个预定义的图	<code>tf.import_graph_def, tf.Session</code>	<code>TF_GraphImportGraphDef, TF_NewSession</code>
结合生成的操作构造图	Yes	Yes (The C API supports client languages that do this)
梯度	<code>tf.gradients</code>	
函数	<code>tf.python.framework.function.Defun</code>	
控制流	<code>tf.cond, tf.while_loop</code>	
神经网络库	<code>tf.train, tf.nn, tf.contrib.layers, tf.contrib.slim</code>	

表 5.2: 语言支持

#### 5.6.4 运行一个预定义的图

语言绑定希望定义下面的类:

- Graph: 一个图表示一个 TensorFlow 计算。有一些对应 C API 的 `TF_Graph` 的操作组成, 主要用作创建操作和启动会话的参数。在图 (`TF_GraphNextOperation`) 上也支持迭代, 通过名字 (`TF_GraphOperationByName`) 查看操作转换为或者从 `GraphDef` protocol 消息 (`TF_GraphToGraphDef` 和 `TF_GraphImportGraphDef`) 转换
- Operation: 表示图上的一个计算节点。对应 C API 中的 `TF_Operation`
- Output: 表示图中操作的输出。有一个 `DataType`(和最终的形状) 也许作为输入参数传送给一个函数进行加操作, 或者对于一个 Session 的 `Run()` 方法获取 `tensor` 作为输出。对应 C API 的 `TF_Output`。
- Session: 表示 TensorFlow 运行环境的类似实例的一个客户端呢。主要工作是构造一个 `Graph` 和在图上调用 `Run()` 选项。对应 C API 中的一个 `TF_Session`。
- Tensor 表示数据类型为 `DataType` 的 N 维数组。获取数据进出 `Session` 的 `Run()` 调用。对应 C API 的 `TF_Tensor`。
- `DataType`: 一些 TensorFlow 支持的可能的类型, 对应 C API 中的 `TF_DataType` 和 Python API 中的 `dtype` 对应

### 5.6.5 图的构造

TensorFlow 有一些操作和列表不是静态的，因此我们推荐通过手写生成添加操作到图上（军官厚些一些是一个好的找出生成器应该生成什么），需要生成一个包含一个 OpDef protocol 消息的函数。有一些方法获取注册操作的 OpDefs 列表：

- 在 CAPI 中的 TF\_GetAllOpList 获取所有祖册的 OpDef protocol 消息。这可能用用于客户端语言写生成器。这要求客户端语言为了解释 OpDef 消息有 Protocol buffer 支持
- C++ 函数 OpRegistry::Global()->GetRegisteredOps() 返回所有注册的 OpDef 的相同的列表（定义在 tensorflow/core/framework/op.h）。这可以用与在 C++（对于没有 protocol buffer 支持的语言来说特别管用）写生成器
- 通过一个自动的处理 ASCII 序列版本被周期性的检查 tensorflow/core/ops/ops.pbtxt
  - 在 CameICase 中操作的名字。为了按照语言习惯生成函数。例如，如果语言使用 snake\_case，使用 CameICase 态体操作的函数名称
  - 一个输入和输出列表。对于一字儿通过属性访问的多台，如在 [Adding an op](#) 描述
  - 一些属性值是否认的。注意一些将被自动推断（如果他们由输入决定），一些将可选（如果他们有一个默认），一些将要要求（非默认），
  - 常用操作，输入，输出和非推理的属性的文档
  - 一些通过运行环境使用的领域可能被代码生成器胡萝卜

一个 OpDef 能转化为 text 函数添加用 TF\_OperationDescription C API 添加操作到图上（打包语言的外部函数接口）

- 开始 TF\_NewOperation() 创建 TF\_OperationDescription\*
- 当输入（依赖于是否输入有一个列表类型）调用 TF\_AddInput() 或者 TF\_AddInputList()
- Call TF\_SetAttr\*() 函数设置非推理的属性。如果你不想覆盖默认值也许跳过属性
- 如果需要设置选项范围
  - TF\_SetDevice(): 抢播操作到指定设备
  - TF\_AddControlInput(): 添加请求在一个操作开始前另一个操作完成前
  - TF\_SetAttrString("\_kernel") 设置内核标记（很少使用）
  - TF\_ColocateWith() 布置一个操作

- 调用 `TF_FinishOperation()` 调用时，添加这个操作到图上，之后不能被修改

存在样本运行代码生成器作为构建程序 (Bazel genrule) 的一部分。带把生成能用一个自动的 cron 程序运行，可能在结果中检测。这在生成代码和 OpDef 的生成之间创建一个分歧进入仓库。但是对于 go 语言的 go get 和 Rust cargoops 希望代码之前生成期望被生成的情况。在最后，对于一些语言代码可能从 `tensorflow/core/ops/ops.pbtxt` 动态生成。

### 5.6.6 处理常数

如果用户可以提佛那个仓鼠给输入参数调用将边的更简单。生成的代码转化常熟为操作添加到图上用于作为输入对操作被实例化。

### 5.6.7 可选参数

如果语言允许对一个函数 (想关键字参数 (默认使用 Python) ) 的可选参数，为可选属性，操作名称，设备，控制输入等使用阿门。在一些语言中这些选项参数可以用于动态 scopes (像 Python 的 with 块)。没有这些特性，库也许如在 C++ 版本的而 TensorFlow API 存储构建的样本。

### 5.6.8 Name scopes

用一些 scope 问津结构支持命名图的操作是一个好的想法，特别是考虑 TensorBoard 依赖它用合理的方式显示大图。Python 和 C++ 有不同的方法：在 Python 中，使用 with 块，目录作为名字的一部分。有一个本地线程栈结合 scopes 定义名字层级结构。名称的最新组件通过用户明确使用或者按照操作被默认添加。在 C++ 中名字的目录部分存储在 Scopes 对象。`NewSubScope()` 方法添加名字的一部分返回一个新的 Scope。最新的名字的组件用 `WithOpName()` 方法设置，想 Python 默认的通过添加的操作命名。Scope 对象明确的传递指定的名字。

### 5.6.9 包装器

确保一些函数的私有属性以至于包装器函数做一点额外的工作可能就被替代。这也给一个 escape 从生成代码的 scope 外支持特性。

包装器的一个用法是支持 `SparseTensor` 输入输出，一个 `SparseTensor` 是一个三个 dense tensor 的 `SparseTensor`: 索引，值，和形状。值是响亮的大小 n，形状是响亮的 rank，索引时一个矩阵 [n,rank]。

另一个原因四用包装器保持状态。有一些这样的操作 (例如变量) 有一些同伴操作用于操作状态。在 Python API 有类用于操作，操作的构造体创建 op，类的方法添加组件到图上操作状态。

### 5.6.10 其它的考虑

- 它有些关键字用于和语言关键字（或者其他符号哦将残生困难，想库函数的名字或者生成的代码中的变量引用）冲突的关键字重命名操作函数和参数
- 通常对于添加一个 Const 操作到图上的函数是一个包装器因此生成的函数将有同于的 DataType 输入

### 5.6.11 梯度，函数和控制流

在这里支持梯度，函数和控制流操作在 Python 外的语言中不可用，当C API提供需要的支持后这将被跟新。

## 5.7 一个 TensorFlow 模型文见得开发者工具

大多数用户不需要考虑 TensorFlow 存储在磁盘中文件的内部细节，但是如果你是一个工具开发者也许需要考虑。例如你也许想分析模型或者在 TensorFlow 和其他格式之间转化。这个向导尝试向你解释一些你如何结合主要文件保存模型数据 的详细工作，确保容易开发一些工具。

### 5.7.1 Protocol Buffers

所有的 TensorFlow 的文件格式都是基于Protocol Buffers，因此熟悉它是如何工作的很有价值。总结来说你在文本文件中定义数据结构，protobuf 用 C, Python 或者你可以载入的其他语言生成类，以一种友好的方式访问数据。我们经常认为 Protocol Buffers 作为 protobufs，我们将在这个向导中使用用这个这个惯例。

### 5.7.2 GraphDef

在 TensorFlow 中图对象是计算的基础。这包含一些节点组成的网络，每个节点代表一个操作，节点作为输入或者输出 被连接到其它节点，你可以通过调用 `as_default_def()` 保存它，返回一个 GraphDef 对象。

GraphDef 类是一个定义在 tensorflow/core/framework/graph.proto 定义的 ProtoBuf 库创建的 GraphDef 类。protobuf 工具解析这个文本文件，生成代码载入，存储和操作图定义。如果你看到一个标准的 TensorFlow 文件表示一个模型，它很可能包含 protobuf 代码保存的一些列序列化的的 GraphDef 对象。这个生成代码 用于保存和从文件中载入 GraphDef 文件。代码通常像下面这样载入这个模型 `graph_def = graph_pb2.GraphDef()` 这行创建一个空的 GraphDef 对象，这个类从定义在 `grapg.proto` 中定义的本质文件创建。这个对象将从我

们的文件 操作这个对象 `with open(FLAGS.graph, "rb") as f` 这里我们传递进脚本一个路径获取文件

```
1 if FLAGS.input_binary:  
2     graph_def.ParseFromString(f.read())  
3 else:  
4     text_format.Merge(f.read(), graph_def)
```

## 文本或者二进制

事实上一个 Protobuf 我们可以存储进入两种不同的格式。文本格式是一个人类可读的形式，对于调试和编辑十分方面，但是当有一些想权重的数值数据存储文本格式的文件将很大。你可以查看[graph\\_run\\_run2.pbtxt](#)一个晓得样本 二进制文件相比文本文件小得多，尽管他们对我们不可读，在这个脚本，我们要求用户应用一个 flag 指示 师傅输入是二进制还是文本，因此我们知道正确的函数调用。你可以在[inception\\_v3 archive](#) 找到一个大型二进制文件样本。正如 `inception_v3_2016_08_28_frozen.pb`。这个 API 本身可以有一些让人困惑，二进制调用事实上是 `ParseFromString()`，然而我们用一个来自 `text_format` 模块的使用函数 载入原始的文件。

### 5.7.3 Nodes

当你载入一个文件进入 `graph_def` 变量时，你可以访问里面的数据，对于多数常见目的，重要的章节是节点列表 存储在节点数目中。这里的代码循换处理这个操作 `for node in graph_def.node` 每个节点是一个 `NodeDef` 对象，定义在[tensorflow/core/framework/node\\_def.proto](#)，这是 TensorFlow 图的基本构建模块，每个定义的单独操作用输入连接，这里的 `number` 是 `NodeDef`。

name

每个节点应该有一个独一无二的 id，不被其他的图中的节点使用。如果你不指定一个作为你用 Python API 构建一个 graph。一个翻译到操作的名字上，想”MatMul” 和单调递增的数连接，像为你选择”5”，名字被用于定义两个节点的连接。当运行时为你的整个图设置输入和输出。

op

这定义了运行的操作，例如”Add”，”MatMul” 或者”Conv2”。当一个图运行的时候，op 的名字被在实现中注册查找。注册通过调用 `REGISTER_OP()` 宏操作。像在 [tensorflow/core/ops/nn\\_ops.cc](#)

### input

一个字符串列表，列表中每个元素是其他节点的名字，通常跟着引号和输出端口。例如，一个节点和两个输入有一个像 `["some_node_name", "another_node_name"]`, 等价于 `["some_node_name:0", "another_node_name:0"]` 定义节点的第一个输入作为节点的来自 “`some_node_name`” 的第一个输出，第二个输入来自“`another_node_name`” 的 第一个输出。

### device

在大多数情况下你可以忽略这个，因为它定义在分布式环境下节点运行的位置，或者什么时候你想强制操作运行在 CPU 或者 GPU 上。

### attr

这是一个 key/value 存储节点的属性。有节不变的参数，一些在像卷积滤波器的尺寸不改变。或者常数操作 的值。因为有如此多不同类型的属性值，对于字符串，对证书或者 tensor 值的数组，这里有一个分割的 protobuf 文件定义的数据结构保存他们，在 [tensorflow/core/framework/attr\\_value.proto](#). 每个属性有一个独一无二的名字字符串，期待的属性被列出然后操作被定义。如果一个属性被节点呈现。但是如果一个操作定义中的默认的列表， 默认被用于图的创建。你可以通过调用 `node.name` 访问所有的数据,`node.op` 等在 Python 中节点列表列表存储在 GraphDef 是一个模型架构的完整定义。

### 5.7.4 Freezing

一个让人难以理解的是训练中的权重不被存储在文件中。相反，他们被保存在 单个的 checkpoint 文件中，在 graph 中的 Variable 操作，当他们被初始化的时候 载入最新的值，因此有[freeze\\_graph.py](#) 脚本接受一个图定义和一些 checkpoint 文件同时 freeze 他们到一个单独的文件，载入 GraphDef 的时候，从最新的 chepoint 文件获取所有变量的值，当取代 Variable 操作和 Const 权重的数值数据存储在他的属性中，然后剔除额外的不能用于钱箱推理的节点，vaocun 输出结果的 GraphDef 进一个输出文件。

### 5.7.5 权重格式

如果你正在处理 TensorFlow 模型表达的神经网络模型，一个常见的问题是提取和解释权重值。一个通用的 方法存储他们，例如在图中 `freezed_graph` 脚本创建，它作为 Const 操作包含作为 Tensors 的权重。这些定义在 [tensorflow/core/framework/tensor.proto](#) 包含数据的类型和尺寸，正如值本身。在 Python 中，你通过 `NodeDef` 从一个 `NodeDef` 操作获取 `TensorProto` 对象表达一个 Const 操作 通过调用想 `some_node_def.attr['value'].tensor`.

这样给你一个权重数据的对象表达。数据将被存储在有 suffix\_val 的列表中作为对象类型的索引，例如 float\_val 对于 32 位浮点数据类型。卷及权重的顺序经常对于处理在不同的框架中转化有技巧的。在 TensorFlow 中，滤波器权重对于 Conv2D 操作被存储在第二行。期待的数据顺序 [filter\_height,filter\_width,input\_depth,output\_depth]，这里 filter\_count 随着 内存中一个邻接的值滑动平均增加。



# Chapter 6

## 性能向导

这个向导包含了一些很好的实践来优化你的 TensorFlow 代码。向导被分成下面的章节：

- 一般的最佳实践包括常用的多种模型和硬件
- 优化 GPU 详细的技巧和 GPU 有关
- CPU 优化 详细的 CPU 特定信息

### 6.0.1 一般的最佳实践

最佳的实践包括下面章节：

- 输入 pipeline 优化
- 数据格式
- 常用的融合操作
- 从源代码构建和安装

#### 输入 pipeline 优化

常用的模型从磁盘获取数据在通过网络发送数据之前提前处理它，例如模型按照下面处理 JPEG 图像：从磁盘载入图像，解码 JPEG 为 tensor，剪切填充，可能还有翻转和扭曲，然后分批处理。下面的图是输入 pipeline。正如 GPU 和其它的硬件极速器会更快，预处理数据可能成为一个瓶颈。如果输入 pipeline 是瓶颈可能变得很复杂。一个直接的方法是在输入 pipeline 后减少模型为单个操作，每秒测量样本。如果对于完整的模型和单个的模型没表中样本的差别很小不同的，输入 pipeline 可能是瓶颈。下面有一些其它的方法识别这个问题。

- 
- 通过 `watch -n 2 nvidia-smi` 检查 GPU 是否被完全使用。如果 GPU 利用没有达到 80-100%，输入 pipeline 可能是瓶颈。
  - 生成时间线查看大模块空白。一个生成时间线的例子在 [XLA JIT](#)
  - 检查 CPU 使用。他可能有一个优化的 pipeline 和缺乏 CPU 循环处理 pipeline
  - 评估生产力需要和验证在这个生产力条件下磁盘使用。一些云方案有网络添加磁盘速度 50MB/sec，这比机械硬盘的 150/MS/sec 和 SATA SSD 的 500MB/sec,PCIe SSD 的 2000+ MB/sec 低得多

## 在 CPU 上处理

防止输入 pipeline 在 CPU 上能极大地提高性能。利用 CPU 处理输入 pipeline, GPU 训练。确保预处理在 CPU 上，按照如下操作打包:

```
1 with tf.device('/cpu:0'):  
2     # function to get and process images or data.  
3     distorted_inputs = load_and_distort_images()  
4
```

如果你用 `tf.estimator.Estimator` 输入函数自动放置到 CPU 上。

## 用 Dataset API

[Dataset API](#) 防止 `queue_runner` 作为推荐的 API 建立输入 pipelines，这个 API 在 TensorFlow 1.2 被天际到 contrib 中之后将被移动到 TensorFlow 核心。[ResNet example\(arXiv:1512.03385\)](#) 训练 CIFAR-10 说明通过 `tf.estimator.Estimator` 使用 Dataset API。Dataset API 用 C++ 多线程和一些低层调用而不是限制 Python 多线程性能的 `queue_runner`。尽管用 `feed_dict` 输入数据提供了恒高的灵活性，在多数实例中 `feed_dict` 不能规模化的优化。然而，在单个 GPU 上的实例使用差别可能微不足道。用 Dataset API 是强雷推荐的，尝试下面的代码:

```
1 # feed_dict often results in suboptimal performance when using large inputs.  
2 sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})  
3
```

## 用大文件

读大量的小文件会极大地影响 I/O 的性能。一个方法是通过处理输入数据为（大约 100MB 或者更大）的 TFRecord 文件得到最大得到最大的 I/O 性能。对于小型数据集（200MB-1GB），最好的方法是载入整个数据集到内存。文档[Downloading and converting](#)

---

[to TFRecord format](#)包含一些创建 TFRecord 和转化 CIFAR-10 数据集为 TFRecords 的信息。

### 6.0.2 数据格式

数据格式涉及到传给 Op 的 Tensor 的结构。下面的讨论明确 4D Tensor 代表图像，在 TensorFlow 中 4 维 Tensor 的各个部分分别代表如下：

- N: 图象的批数
- H: 图象的高
- W: 代表凸显的宽
- C: 代表图象的通道数，1 代表黑白图像，3 代表真彩图像

在 TensorFlow 中有两种命名惯例，代表两种常用的格式：

- NCHW 或者 channels\_first
- NHWC 或者 channels\_last

NHWC 是 TensorFlow 默认的，NCHW 是在 NVIDIA GPU 上用 cuDNN 优化后的格式。构建模块的最佳实践是结合两种格式。最简单的是在 GPUs 上训练然后在 CPUs 上推理。如果 TensorFlow 结合[Intel MKL](#)优化编译的，一些操作，特别是和基于 CNN 模型的将被优化支持 NCHW。如果不使用 MKL，用 NCHW 时一些操作将不支持在 CPU 上运行。

## 常见的融合操作

融合操作结合多个操作为一个内核提高性能，在 TensorFlow 有一些融合操作和 CLA 可能提高性能时创建融合操作。下面被选择的融合操作可能极大地提高性能而未被忽视。

### 融合批规范

融合批规范结合多个需要批正规化的操作为一个内核。批规范对那些建立大的操作时间的比例是一个高昂的操作。用融合规范可能导致 12%-30% 的加速。有两个常见的批操作支持融合。核心 `tf.layers.batch_normalization` 在 TensorFlow 1.3 开始添加融合

```
1 bn = tf.layers.batch_normalization(input_layer, fused=True, data_format = 'NCHW')
  )
```

2

contrib 中的 `tf.contrib.layers.batch_norm` 方法从 TensorFlow1.0 起也有融合选项

```
1 bn = tf.contrib.layers.batch_norm(input_layer, fused=True, data_format='NCHW')
```

2

---

### 6.0.3 从源代码构建安装

默认 TensorFlow 二进制针对最广泛的硬件使得 TensorFlow 对于每个人都可以使用。如果用 CPU 进行训练或者推理，推荐结合对于使用的 CPU 可用的优化去编译 CPU。在 CPU 上加速推理和训练在[Comparing compiler](#)被记录。

安装优化后的 TensorFlow 版本，从源代码[构建安装](#)，如果有在不同的硬件平台上构建 TensorFlow 的需要，交叉编译对于目标平台有最高的优化。下面没了一个用 bazel 为指定平台编译的例子：

```
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
```

## 环境构建和安装技巧

- ./configure 要求计算兼容性在构建的时候被包含。着不影响总体性能但是影响初始化启动。在运行 TensorFLow 一次以后，编译的内核荣光 CUDA 缓存。如果用一个 docker 容器，数据不缓存每次 TensorFlow 启动时间较长。通过 GPUs 的最佳实践是包含[compute capabilities](#)，李如意 P100:6.0,Titan X(Pascal):6.1,Titan X(Maxwell):5.2 和 k80:3.7。
- 用支持所有的目标 CPU 优化的 gcc 编译器。推荐最小的 gcc 版本是 4.8.3。在 OS X 上升级最新的 Xcode 版本用 clang 版本结合 Xcode。
- 安装 TensorFlow 支持的最新的稳定 CUDA 平台和 cuDNN 库。

## 优化 GPU

这部分包含指定 GPU 的没有在[General best practices](#)被包含的技巧。在多个 GPU 上获取优化性能是一个挑战。一个常见的方法是数据并行。通过使用数据并行利用创建多个模型的拷贝，这些模型作为一个 tower，放置一个 tower 在每个 GPU 上。每个 tower 在不同的小批数据上操作每个 tower 得到更新的变量和梯度对性能有什么影响，方法，模型的收敛。下面提供了一个在多个 GPUs 上放置变量和 tower 的概览。在下一章将详细的讨论更多用于在 tower 分享和更新变量复杂的方法。

最好的处理变量更新的方法是依赖模型，甚至是硬件被如何配置。一个例子是一个例子是两个系统都有 NVIDIA Tesla P100s 一个是用 PCIe 另一个用[NVLink](#)在这种场景下对于每个系统的优化方案也许不同。对于真实世界的例子，读[benchmark](#)详细的设置多平台优化。下面是一个 benchmark 多平台和配置的总结：

- 
- Tesla K80: 如果 GPU 在相同的 PCI 中线跟复杂能够用NVIDIA GPUDirectPeer to Peer, 然后然后放置多个相等的 GPUs 用于训练是最好的方法。如果 GPUs 不能用GPUDirect, 然后放置变量在 CPU 上是最好的选择。
  - Titan X(Maxwell 和 Pascal), M40,P100 和类似的: 详细的像 ResNet 和 InceptionV, 放置变量在 CPU 上是优化设置, 但是对于有很多变量的模型像 AlexNet 和 VGG, 用 GPUs 和 NCCL 是更好。

一场用的管理变量放置的方法是创建一个方法决定每个操作放置在哪里和通过 `tf.device()` 用方法指定设备名字。考虑一个场景是一个模型在两个 GPU 上训练变量被放在 CPU 上。则将在每个 GPU 上循环创建放置 tower, 一个习惯的设备放置方法将创建监视查看操作的 Variable, VariableV2 和 VarHandleOp 的类型, 遗失者他们被放在 CPU 上。所有的其他操作将放在目标 GPU 上, 构件图将按照下面处理:

- 第一次循环模型中的一个 tower 将为 `gpu:0` 创建。当值操作期间, 通常设备防治方法将指示变量被昂在 `cpu:0` 上其他操作放在 `gpu:0` 上。
- 第二次循环, `resue` 设置为 `True` 预示着变量被重用 tower 被放置在 `cpu:0` 上被重用所有的其他操作将被创建放置在 `gpu:1` 上

最后的结果是所有的变量放在 CPU 上每个 GPU 结合模型拷贝计算操作。下面的代码段说明两个不同的变量防治方法: 一个放置变量在 CPU 上, 一个放置变量通过 GPUs。

```
1 class GpuParamServerDeviceSetter(object):
2     """Used with tf.device() to place variables on the least loaded GPU.
3
4     A common use for this class is to pass a list of GPU devices, e.g. ['gpu:0',
5         'gpu:1','gpu:2'], as ps_devices. When each variable is placed, it will be
6         placed on the least loaded gpu. All other Ops, which will be the computation
7         Ops, will be placed on the worker_device.
8     """
9
10    def __init__(self, worker_device, ps_devices):
11        """Initializer for GpuParamServerDeviceSetter.
12        Args:
13            worker_device: the device to use for computation Ops.
14            ps_devices: a list of devices to use for Variable Ops. Each variable is
15                assigned to the least loaded device.
16        """
17        self.ps_devices = ps_devices
18        self.worker_device = worker_device
19        self.ps_sizes = [0] * len(self.ps_devices)
20
```

---

```

21 def __call__(self, op):
22     if op.device:
23         return op.device
24     if op.type not in [ 'Variable' , 'VariableV2' , 'VarHandleOp' ]:
25         return self.worker_device
26
27 # Gets the least loaded ps_device
28 device_index, _ = min(enumerate(self.ps_sizes), key=operator.itemgetter(1))
29 device_name = self.ps_devices[device_index]
30 var_size = op.outputs[0].get_shape().num_elements()
31 self.ps_sizes[device_index] += var_size
32
33 return device_name
34
35 def _create_device_setter(is_cpu_ps, worker, num_gpus):
36     """Create device setter object."""
37     if is_cpu_ps:
38         # tf.train.replica_device_setter supports placing variables on the CPU, all
39         # on one GPU, or on ps_servers defined in a cluster_spec.
40         return tf.train.replica_device_setter(
41             worker_device=worker, ps_device='/cpu:0', ps_tasks=1)
42     else:
43         gpus = [ '/gpu:%d' % i for i in range(num_gpus) ]
44         return ParamServerDeviceSetter(worker, gpus)
45
46 # The method below is a modified snippet from the full example.
47 def _resnet_model_fn():
48     # When set to False, variables are placed on the least loaded GPU. If set
49     # to True, the variables will be placed on the CPU.
50     is_cpu_ps = False
51
52     # Loops over the number of GPUs and creates a copy ("tower") of the model on
53     # each GPU.
54     for i in range(num_gpus):
55         worker = '/gpu:%d' % i
56         # Creates a device setter used to determine where Ops are to be placed.
57         device_setter = _create_device_setter(is_cpu_ps, worker, FLAGS.num_gpus)
58         # Creates variables on the first loop. On subsequent loops reuse is set
59         # to True, which results in the "towers" sharing variables.
60         with tf.variable_scope('resnet', reuse=bool(i != 0)):
61             with tf.name_scope('tower_%d' % i) as name_scope:
62                 # tf.device calls the device_setter for each Op that is created.
63                 # device_setter returns the device the Op is to be placed on.

```

---

```
64     with tf.device(device_setter):
65         # Creates the "tower".
66         _tower_fn(is_training, weight_decay, tower_features[i],
67                   tower_labels[i], tower_losses, tower_gradvars,
68                   tower_preds, False)
```

在不远的将来上面的代码将被用于说明意图，这将被很容易用高级方法支持更广泛的方法。这个[例子](#)将更新更新作为 API 扩展和进展处理多个 GPU 的场景。

## 优化 CPU

包含 Intel Xeon Phi 的 CPU，当从 TensorFlow 原来马构建时获得最优性能所有的说明被目标 CPU 支持。

超过于使用最新的说明，Intel 的 Intel Math Kernel Library 对 TensorFlow 深度神经网络添加了支持。尽管名字不返券静却，这些优化经常被简单的写为 MKL 或者 tensorflow，[TensorFlow with Intel MKL-DNN](#)包含了 MKL 优化的详细说明。

下面的两个配置通过调整线程池优化 CPU 性能

- intra\_op\_parallelism\_treads: 可以用多线程并行执行将调度单个片段进线程池
- inter\_op\_parallelism\_threads: 线程池所有的节点被调度

入下面显示这些配置通过 `tf.ConfigProto`, 传递给 `tf.Session` 中的 `config` 属性。对于两个配置选项，如果他们没有设置或者设置为 0 将默认 CPU 的逻辑数。测试显示对于逻辑核数为 4 和到多 CPU 的 70+ 是高效的。一个常用的优化是设置在池中的线程数等于物理核数而不是逻辑核数。

```
1 config = tf.ConfigProto()
2 config.intra_op_parallelism_threads = 44
3 config.inter_op_parallelism_threads = 44
4 tf.session(config=config)
```

下面比较编译器优化章节包含了不同编译器优化测试结果

## TensorFlow 和 Intel MKL DNN

Intel 通过用 Intel MKL-DNN 为 Intel Xeon 和 Xeon Phi 添加了对 TensorFlow 的优化。优化对于消耗处理器行提供了加速，如 i5 和 I5 的 Intel 处理器。在论文[TensorFlow\\* Optimizations on Modern Intel® Architecture](#) 包含了详细的实现。

MKL 在 tensorflow 1.2 就被添加，当前尽在 Linux 上有小。如果用 `config=cuda` 它将不工作

---

另外对基于 CNN 的模型提供了极大地性能提升，和 MKL 编译创建对 AVXhe AVX2 的优化的一个二进制。结果是一个二进制被优化兼容多数处理器 (2011 年以后的)。TensorFlow 可以通过下面的命令在结合 MKL 优化通过源代码被编译。对于 TensorFlow 之后的源代码版本：

```
1 ./configure  
2 # Pick the desired options  
3 bazel build --config=mkl -c opt //tensorflow/tools/pip_package:build_pip_package
```

对于 TensorFlow 版本为到 1.3.0：

```
1 ./configure  
2 Do you wish to build TensorFlow with MKL support? [y/N] Y  
3 Do you wish to download MKL LIB from the web? [Y/n] Y  
4 # Select the defaults for the rest of the options.  
5  
6 bazel build --config=mkl --copt="-DEIGEN_USE_VML" -c opt //tensorflow/tools/  
    pip_package:build_pip_package
```

## 调整 MKL 获得最佳性能

这部分将介绍不同的配置和环境变量用于调整 MKL 得到优化的性能。在调整多环境变量之前确保模型使用 NCHW (chennel\_first) 数据核实。MKL 对于 NCHW 被优化过当用 NCHW 时 Intel 将获得最高性能。MKL 用下面的环境变量调整性能：

- KMP\_BLOCKTIME-设置时间，毫秒，表示线程应该等待的时间在完成并行执行后，休眠之前
- KMP\_AFFINITY-启动运行时间库绑定线程到物理处理器单元
- KMP\_SETTING-在程序中启动或者警用 OpenMP\* 打印运行事件库环形变量
- OMP\_NUM\_THREADS 指定使用的线程数

更多的关于 KMP 变量在[Intel's](#)网站上 OMP 变量在[gnu.org](#) 尽管通过调整环境变量可能有极大的提升，下面的讨论简单的建议通过下面的环境变量设置 inter\_op\_parallelism\_threads 等于物理 CPU 的核数用

- KMP\_BLOCKTIME=0
- KMP\_AFFINITY=granularity=fine,verbose,compact,1,0

用命令行设置 MKL 环境变量：

---

```
1 KMP_BLOCKTIME=0 KMP_AFFINITY=granularity=fine ,verbose ,compact ,1 ,0 \
2 KMP_SETTINGS=1 python your_python_script.py
```

通过 Python 的 os.environ 设置 MKL 环境变量:

```
1 os.environ[ "KMP_BLOCKTIME" ] = str(FLAGS.kmp_blocktime)
2 os.environ[ "KMP_SETTINGS" ] = str(FLAGS.kmp_settings)
3 os.environ[ "KMP_AFFINITY" ]= FLAGS.kmp_affinity
4 if FLAGS.num_intra_threads > 0:
5   os.environ[ "OMP_NUM_THREADS" ]= str(FLAGS.num_intra_threads)
```

有一些模型和硬件平台在不同的设置上得到好处。每个变量影响性能在下面被讨论:

- KMP\_BLOCKTIME: MKL 默认为 200ms, 没有为我们的测试优化 0ms 是一个好的默认 CNN 基础的测试模型 AlexNet 的最好性能设置为 30ms, GoogleNet 和 VGG11 最好设置为 1ms。
- KMP\_AFFINITY: 推荐设置是 granularity=fine,verbose,compact,1,0
- OMP\_NUM\_THREADS: 默认物理核数, 当用 Intel Phi 在模型上是调整参数超过匹配的核数有些印象, 更多的模型查看[TensorFlow\\* Optimizations on Modern Intel Architecture](#)获得优化性能
- intra\_op\_parallelism\_threads: 推荐设置这个等于物理核数, 设置值为 0 默认导致值被设置逻辑核的个数, 对一些架构是一个尝试选项, 值和 OMP\_NUM\_THREADS 应该相等
- inter\_op\_parallelism\_threads: 推荐设置等于 sockets 的数量, 默认设置值为 0 将导致值被设置为逻辑核数

## 比较编译器的优化

在不同的 CPU 平台上用不同的编译器优化收集训练和推理的执行信息。模型被用在 ResNet-50[arXiv:1512.03385](#)和 Inception V3[arXiv:1512.00567](#)对于更多的测试, 当在环境变量 KMP\_BLOCKTIME MKL 优化被使用被设置为 0ms, KMP\_AFFINITY 为 granularity=fine,verbose,compact,1,0

推理 InceptionV3

## 环境

- 实例类型: AWS EC2 m4.xlarge
- CPU: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz (Broadwell)
- 数据集: Imagenet

- 
- TensorFlow 版本 1.2.0 RC2
  - 测试脚本:[tf\\_cnn\\_benchmarks.py](#)

Batch Size 为 MKL 测试执行命令

```
1 python tf_cnn_benchmarks.py --forward_only=True --device=cpu --mkl=True \
2 --kmp_blocktime=0 --nodistortions --model=inception3 --data_format=NCHW \
3 --batch_size=1 --num_inter_threads=1 --num_intra_threads=4 \
4 --data_dir=<path to ImageNet TFRecords>
```

优化	数据格式	图像/秒 (步时)	Intra 线程	Inter 线程
AVX2	NHWC	6.8(147ms)	4	0
MKL	NHWC	6.8(147ms)	4	1
MKL	NHWC	5.95(168ms)	4	1
AVX	NHWC	4.7(211ms)	4	0
SSE3	NHWC	2.7(370ms)	4	0

Batch Size:32 执行 MKL 测试命令:

```
1 python tf_cnn_benchmarks.py --forward_only=True --device=cpu --mkl=True \
2 --kmp_blocktime=0 --nodistortions --model=inception3 --data_format=NCHW \
3 --batch_size=32 --num_inter_threads=1 --num_intra_threads=4 \
4 --data_dir=<path to ImageNet TFRecords>
```

优化	数据格式	图像/秒 (步时)	Intra 线程	Inter 线程
MKL	NCHW	10.24 (3125ms)	4	1
MKL	NHWC	8.9 (3595ms)	4	1
AVX2	NHWC	7.3 (4383ms)	4	0
AVX	NHWC	5.1 (6275ms)	4	0
SSE3	NHWC	2.8 (11428ms)	4	0

推理 ResNet-50

- 实例类型: AWS EC2 m4.xlarge
- CPU: Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz (Broadwell)
- 数据集: ImageNet
- TensorFlow 版本: 1.2.0 RC2
- 测试脚本: [tf\\_cnn\\_benchmarks.py](#)

---

优化	数据格式	图像/秒 (步时)	Intra 线程	Inter 线程
AVX2	NHWC	6.8 (147ms)	4	0
MKL	NCHW	6.6 (151ms)	4	1
MKL	NHWC	5.95 (168ms)	4	1
AVX	NHWC	4.7 (211ms)	4	0
SSE3	NHWC	2.7 (370ms)	4	0

Batch size: 32

执行 MKL 测试的命令

```

1 python tf_cnn_benchmarks.py --forward_only=True --device=cpu --mkl=True \
2 --kmp_blocktime=0 --nodistortions --model=resnet50 --data_format=NCHW \
3 --batch_size=32 --num_inter_threads=1 --num_intra_threads=4 \
4 --data_dir=<path to ImageNet TFRecords>

```

优化	数据格式	图像/秒 (步时)	Intra 线程	Inter 线程
MKL	NCHW	10.24 (3125ms)	4	1
MKL	NHWC	8.9 (3595ms)	4	1
AVX2	NHWC	7.3 (4383ms)	4	0
AVX	NHWC	5.1 (6275ms)	4	0
SSE3	NHWC	2.8 (11428ms)	4	0

训练 InceptionV3

环境

- 实例类型: Dedicated AWS EC2 r4.16xlarge (Broadwell)
- CPU: Intel Xeon E5-2686 v4 (Broadwell) Processors
- 数据集: ImageNet
- TensorFlow 版本: 1.2.0 RC2
- 测试脚本: [tf\\_cnn\\_benchmarks.py](#)

执行 MKL 测试命令

```

1 python tf_cnn_benchmarks.py --device=cpu --mkl=True --kmp_blocktime=0 \
2 --nodistortions --model=resnet50 --data_format=NCHW --batch_size=32 \
3 --num_inter_threads=2 --num_intra_threads=36 \
4 --data_dir=<path to ImageNet TFRecords>

```

优化	数据格式	图像/秒 (步时)	Intra 线程	Inter 线程	
MKL	NCHW	20.8	36	2	ResNet 和 AlexNet 在这个配置上运行但是在一个 ad hoc manner 中。没有足够运行执行结合标的结果。不完整的结果强烈的暗示了最后的结果将雷士上面的 MKL 比 AVX2 大约 3x+ 的提升。
AVX2	NHWC	6.2	36	0	
AVX	NHWC	5.7	36	0	
SSE3	NHWC	4.3	36	0	

配置上运行但是在一个 ad hoc manner 中。没有足够运行执行结合标的结果。不完整的结果强烈的暗示了最后的结果将雷士上面的 MKL 比 AVX2 大约 3x+ 的提升。

## 6.1 高性能模式

这个文档和伴随的[脚本](#)详细介绍了日结构建大规模的针对多系统类型和网络拓扑模型的大规模模型。在这个文档中的技术使用一些低级的 TensorFlow Python 原语。在将来这些技术中的一些江北整合进高级 API。

### 6.1.1 输入 pipeline

在[Performance Guide](#)中解释了如何确定可能的输入 pipeline 问题和最优的实践。我们发现使用[tf.FIFOQueue](#)和 [tf.train.queue\\_runner](#) 在每秒处理大型输入 (像训练 ImageNet 的[AlexNet](#)) 和样本的时候不能跑满多个当前常用的 GPU。这是因为使用 Python 线程作为实现。Python 线程的开支太大。

另一个实现方案是我们在 TensorFlow 上用本地并行构建一个输入 pipeline，我们在[脚本](#)中实现了这个方法。w 我们的实现由三个 stage 组成：

- I/O 读：选择从磁盘中读取图像文件
- 图像处理：解码图像记录为图像，处理组织他们为一个 mini-batch
- CPU-to-GPU 数据转换。从 CPU 转换图像到 GPU

每一个的主要部分结合其他部分使用 `data_flow_ops.StringArea,StagingArea`(类似 `tf.FIFOQueue`)。不同的是 StagingArea 不保证 FIFO 的孙旭，但是提供简单的函数，结合其他的 stage 在 CPU 和 GPU 上平行执行。分开输入 pipeline 为三个 stage，操作独立并行运行是规模化的充分利用的多核环境。下面的章节详细的介绍了 `data_flow_ops.String`。

### 6.1.2 并行化 I/O 读取

`data_flow_ops.RecordInput` 用于从磁盘上并行化读取。给定一个输入文件表示 TFRecord，`RecordInput` 持续使用后台线程读取记录。记录被放在他自己的打的内部线程池单后在至少一半容量的时候载入，产生输出 tensor。

这个操作自己的内部的线层被小号最小 CPU 的 I/O 时间决定，这允许模型剩下的部分平稳运行。

### 6.1.3 并行化图像处理

在图像从 RecordInput 读取后他们被作为 tensor 传递给图像处理 pipeline。为了确保图像处理 pipeline 容易解释，假设输入 pipeline 是每 256 大小的批有 8GPU。

并行的时候有 256 个记录被单独处理。这在图上开启了 256 个独立的 RecordInput 读 op。每个读 op 跟着一些用于图像处理的操作在并行运行时独立执行。图像预处理操作包含想图像解码，变形和变换大小。

当图像被处理的时候他们每批 32 被连接进入 8 个 tensor。相比于使用 `tf.concat` 达到这个目的，作为一个单独的操作实现等待所有的输入在连接他们在一起之前被准备好，`tf.parallel_stack` 被使用。`tf.parallel_stack` 分配一个没有初始化的 tensor 作为输出，每个输入 tensor 被写入，他的设计输出部分的 tensor，只要输入是可用的。

当所有的输入完成后，输出 tensor 沿着图传递。这结合所有输入 tensor 产生的长尾隐藏内存占有。

### 6.1.4 并行化 CPU 到 GPU 数据转化

继续这个假设，目标是 8GPU 每批大小 256 (每个 GPU32)。当输入图像通过 CPU 处理和连接，我们每批大小 32 有 8 个 tensor。

TensorFlow 使得来自于一个设备的 tensor 能在其他任何设备上使用。tensor 实际使用之前在设备之间复制运行调度。然而，如果复制不能及时完成，需要这些 tensor 的计算将停止导致性能下降。

在这个实现中 `data_flow_ops.StagingArea` 用于明确的在并行处理中调度。结果时当计算从 GPU 上开始时。所有的 tensor 已经可用。

### 6.1.5 软件 pipeline

结合所有由不同处理器驱动的 stage 能力，`data_flow_ops.StagingArea` 用在他们之间因此他们在并行处理中运行。StagingArea 是类似 `tf.FIFOQueue` 一个队列的操作，它提供简单的函数在 CPU 和 GPU 上执行。

在模型开始运行所有的 stages 之前，输入 pipeline stage 被 warmed up 到主要的 staging 缓存。在每个运行步，一些数据在每个 stage 开始从 staging buffer 读取，一个被推到结尾。

例如：如果有三个 stage A, B, C，有两个 staging 在 S1 和 S2 之间，warmi up 中，我们运行：

```

1 Warm up:
2 Step 1: A0
3 Step 2: A1 B0
4
5 Actual execution:
6 Step 3: A2 B1 C0
7 Step 4: A3 B2 C1
8 Step 5: A4 B3 C2

```

在 warm up 后，S1 和 S2 有一些数据，实际执行的每一步，一些数据从每个 stageing area 被移出，一些被添加到另一个集合。使用这种机制的好处：

- 所有的 stages 都是非模块的，因为 stage area 总有一些数据在 warm up 后
- 每个 stage 能并行运行，因此他们可以立即开始
- 在固定内存的 staging buffer。我们将有一些额外的数据
- 仅仅单个 session.run() 调用需要运行所有步骤的 stage，使得探测和吊事更容易

### 6.1.6 构建高性能模型的最佳实践

手机下面一些额外的联系可以提高性能增加模型的灵活度。

### 6.1.7 用 NHWC 和 NCHW 构建模型

大多数的 CNN 使用的 TensorFlow 操作支持 NHWC 和 NCHW 数据核实。在 GPU 上，NCHW 更快，但是在 CPU 上，NHWC 有时候更快。

构建一个模型支持两种数据格式保证模型灵活和忽略平台优化操作的能力。CNN 中多数 TensorFlow 操作支持 NHWC 和 NCHW 数据格式。基准脚本支持 NCHW 和 NHWC。NCHW 应该总是当在 GPU 上训练的时候用。NHWC 有时在 CPU 上运行更快。一个灵活的模型可以在 GPU 上用 NCHW 训练在 CPU 上用 NHWC 上结合训练的权重推导。

### 6.1.8 使用融合的 Batch-Normalization

一个 TensorFlow 默认的 BN 作为组件操作被实现。这是非常常见的，但是进仓导致次优的性能。一个可选方案是用融合在 GPU 上有更高性能的 normalization。下面的例子是使用 `tf.contrib.layers.batch_norm` 实现融合的 BN。

```

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

```

### 6.1.9 变量分布和梯度聚合

在训练中，训练的变量值私用聚合的梯度和 delta 更新训练变量。在基准测试脚本中，我们展示了灵活和通常目的的 TensorFlow 原语，一个从该性能分布和聚合方案的多种构建。变量分布和聚合的例子包含在脚本中：

- `parameter_server` 每个训练模型的复制从参数服务器读取变量，单独的更新变量。当每个模型需要变量的时候，他们被通过标准的隐含复制由 TensorFlow 运行环境添加。示例[脚本](#)参数在本地训练使用这个方法，分布的同步训练，分布的一步训练。
- `replicated` 在每个 GPU 上放置完全相同的的复制的变量。前向计算和后向计算可以立即开始作为变量数据。梯度通过所有的 GPU 累积，聚合在一起用于每个 GPU 的肤质变量保证他们同步。
- `distributed_replicated` 在每个 GPU 上沿着主要的参数服务器放置完全相同的训练参数，当变量数据可用的时候前向后向计算立即开始。梯度通过在服务器上的 GPU 累积然后每个服务器聚合梯度用于主要的 copy。在所有的 worker 做这件事后，每个 worker 更新他从主机复制来的参数

下面是额外的关于每个方法的详细信息。

### 6.1.10 参数服务器变量

多数情况下可训练的变量在 TensorFlow 模型中管理是以参数服务器模式。

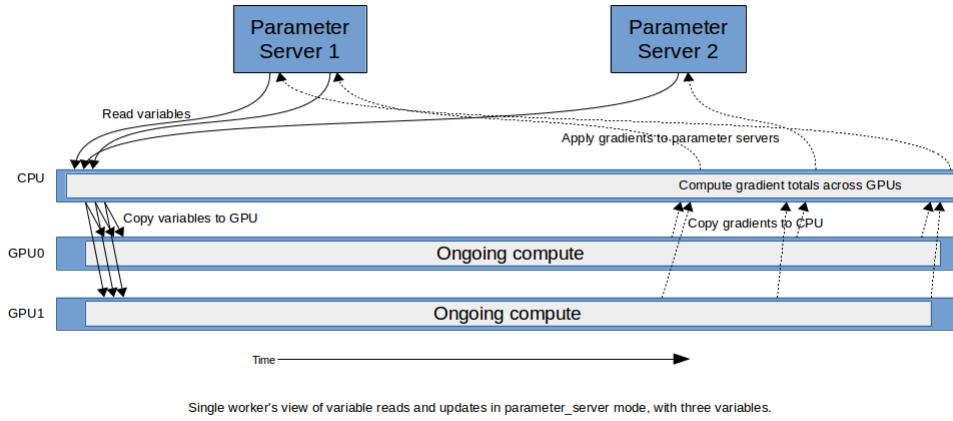
在一个分布式系统，每个 worker 处理同一 model 的运行，每个参数服务器处理自己的住的从主机 (master) 复制过来的变量。当 worker 需要从参数服务器获得变量的时候。它直接访问。在 TensorFlow 运行环境中添加隐含的复制到图上确保变量需要的时候的值在计算设备之间可用。当一个梯度在 worker 上计算的时候，它发送到参数服务器自己的变量，对应的优化器用于更新变量。

有一些技术提高吞吐率：

- 变量基于他的大小在参数服务器间分散，用于负载均衡
- 每个 worker 有多个 GPU，梯度在 GPU 上累积单个聚合梯度发送到参数服务器。这减少的网络带宽和参数服务器的 work 数量

对普通的两个 worker，一个常见模式是异步更新，这里每个 worker 不用和他和其他的 worker 同步更新各自从 master 得到的参数。我们展示这是一个公平的方法在 worker 之间介绍同步因此对于所有的 worker 更新在下一步能开始前完成。

参数服务器方法也能用于本地训练，在这种情况下扩展从 master 负载来的变量到哥哥服务器，他们技能在 CPU 上可能在可用的 GPU 上运行。



因为这个开启的简单性质，这个架构已经在社区中流行起来了。

模式可能通过传递`variable_update=parameter_server`用在脚本中。

### 6.1.11 复制的变量

在这个设计中，每个在 server 上的 GPU 有自己的复制的变量。值通过应用完整的局和提取到每个 GPU 的复制变量保持同步。

变量和数据在训练开始的时候是可用的，因此钱想传递训练可以立即开始。梯度通过 device 聚合，完整的聚合的梯度是应用在每个本地复制。梯度聚合通过 server 扩山到 server 有不同的方法：

- 使用标准的 TensorFlow 操作聚合在一个单独的 device(CPU 或者 GPU) 然后复制它回到所有的 GPUs
- 使用 NVIDIA NCCL，下面的 NCCL 章节描述。

这个模式可以通过传递`variable_update=replicated`用在脚本中。

### 6.1.12 在分布式系统上训练时复制变量

对于变量的复制方法可能被扩展到分布式训练。一个处理这样工作的方法是 replicated mode: 完全通过集群聚合梯度然后应用他们到每个本地复制变量。这在这个脚本的将来版本也许会显示，脚本呈现一个不同的变量，描述。

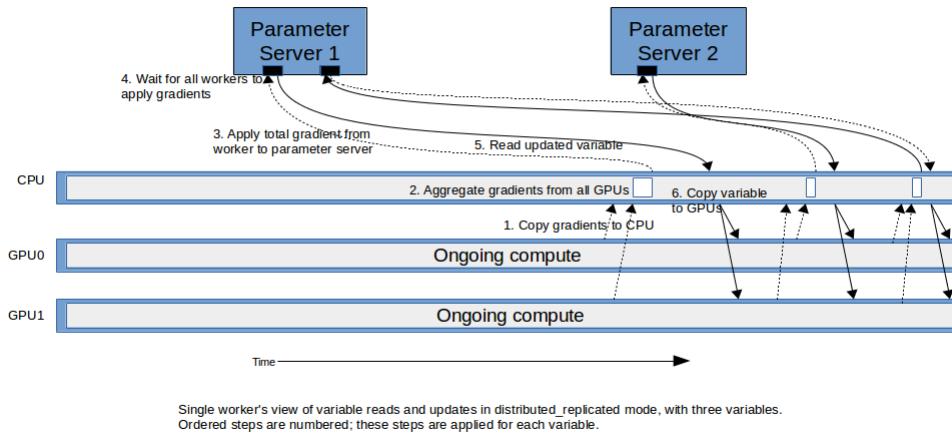
在这个模式，额外的每个 GPU 的复制变量，master 的复制存储在村塾服务器。正如复制模式，训练可以用本地的复制的变量立即开始。

当权重的梯度可用的时候，他们送回参数服务器所有的本地复制被更新：

1. 所有从同一个 worker 来的梯度被聚集在一起

2. 从每个 worker 聚集的梯度被发送到参数服务器拥有的变量，这里制定的优化器用于更新 master 变量的复制
3. 每个 worker 更新它从 master 复制的变量。在例程模型，这集合 cross-replica 做等待所有 workers 完成更新变量，仅仅在所有复制的障碍被释放后获取新的变量。当一个对所有变量的复制完成后，这标记训练的末端，下一步训练可以开始。

尽管这听起来类似使用参数服务器的标准，但是性能进场更好。很可能是应为计算没有延迟，大量复制的梯度的时延可能通过之后的计算曾被隐。这个模式可以传递参数 `variable_update=distributed_replica` 用于脚本。



### 6.1.13 NCCL

为了广播变量在同一机器上通过不同的 GPU 聚合梯度梯度，我们可以使用默认的 TensorFlow 隐含的复制机制。

然而，我们可以使用选项 NCCL(`tf.contrib.nccl`)。NCCL 是 NVIDIA 一个能在不同的 GPU 上可以高效广播和聚合数据库。它在每个 GPU 上调度一个 cooperatting 核心了解基础硬件拓扑下的最佳利用。这可信使用一个 GPU 的 SM。

在我们的试验中，我们通过 NCCL 展示经常比本身有更快的数据聚合。我们的假设是隐含的复制隐含的基本的 free 因此他们能在 GPU 上复制引擎，只要时延能通过主要的计算本身隐藏。尽管 NCCL 可以更快的转换数据，接受一个 SM，给基于 L2cache 添加更多压力。我们的结果显示 8GPU，NCCL 载入更好的性能。然而很少的 GPU，隐含的复制性能更好。

### 6.1.14 Staged 变量

我们进一步介绍一个 staged 变量模式，这里我们用 staging area 到变量读和他们的更新。类似输入 pipeline 的软件的 pipelining，这可以隐藏数据复制时延。如果计算时间花费被复制和聚合更长，复制它本身变得 essentially free。

缺点是所有的权重从之前训练步骤读取。因此它不同于 SGD 算法。但是它能通过掉着那个学习率和其它草参数可能改善收敛。

### 6.1.15 执行脚本

这个章节列出命令行参数的核心和有一些基本的执行主要脚本的例子 ([tf\\_cnn\\_benchmarks.py](#))

**注意:**tf\_cc\_benchmarks.py 用在 TensorFlow 1.1 之后引入的 force\_gpu\_compatible，知道 TensorFlow 1.2 被从源代码被释放被建议。

### 6.1.16 基本的命令行参数

- model: 模型使用。如 resnet50, inception3, vgg16 和 alexnet
- num\_gpus: 使用 GPU 的数量
- data\_dir: 处理数据的路径。如果没有设置，中和数据被使用。用 imagenet 数据这些说明作为起点。
- batch\_size: 每个 GPU 的批的尺寸
- variable\_update: 管理变量的方法:parameter\_server ,replicated, distributed\_replicated, independent
- local\_parameter\_device: 用作参数服务器的设备:CPU 或者 GPU

单个实例的例子

```

1 # VGG16 training ImageNet with 8 GPUs using arguments that optimize for
2 # Google Compute Engine.
3 python tf_cnn_benchmarks.py --local_parameter_device=cpu --num_gpus=8 \
4 --batch_size=32 --model=vgg16 --data_dir=/home/ubuntu/imagenet/train \
5 --variable_update=parameter_server --nodistortions
6
7 # VGG16 training synthetic ImageNet data with 8 GPUs using arguments that
8 # optimize for the NVIDIA DGX-1.
9 python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
10 --batch_size=64 --model=vgg16 --variable_update=replicated --use_nccl=True

```

```

11
12 # VGG16 training ImageNet data with 8 GPUs using arguments that optimize for
13 # Amazon EC2.
14 python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
15 --batch_size=64 --model=vgg16 --variable_update=parameter_server
16
17 # ResNet-50 training ImageNet data with 8 GPUs using arguments that optimize for
18 # Amazon EC2.
19 python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
20 --batch_size=64 --model=resnet50 --variable_update=replicated --use_nccl=False

```

### 6.1.17 分布式的命令行参数

- ps\_host: 逗号分隔的主机用作参数服务器, 格式为 <host>:port, 例如 10.0.0.2:50000
- worker\_hosts: 逗号分隔的主机用作 worker, 格式 <host>:port, 例如 10.0.0.2:50001
- task\_index: 在 ps\_hosts 或者 worker\_hosts 开始的主机索引
- job\_name: 输入 job, 例如 ps 或者 worker

### 6.1.18 分布式例子

下面是一个在两台主机上训练 ResNet-50 的例子 host\_0 (10.0.0.1) 和 host\_1(10.0.0.2) 使用中和数据的例子, 用通过传递--data\_dir 参数传递真实数据。

```

1 # Run the following commands on host_0 (10.0.0.1):
2 python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
3 --batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
4 --job_name=worker --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
5 --worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=0
6
7 python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
8 --batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
9 --job_name=ps --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
10 --worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=0
11
12 # Run the following commands on host_1 (10.0.0.2):
13 python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \
14 --batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
15 --job_name=worker --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
16 --worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=1
17
18 python tf_cnn_benchmarks.py --local_parameter_device=gpu --num_gpus=8 \

```

```

19 --batch_size=64 --model=resnet50 --variable_update=distributed_replicated \
20 --job_name=ps --ps_hosts=10.0.0.1:50000,10.0.0.2:50000 \
21 --worker_hosts=10.0.0.1:50001,10.0.0.2:50001 --task_index=1

```

## 6.2 基准测试

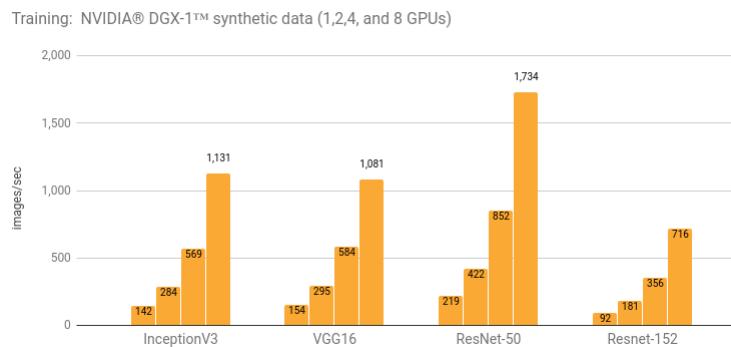
### 6.2.1 概览

选择的图像分类模型被多平台测试为 TensorFlow 社区创建一个参考点。在[Methodology](#) 章节详细描述了测试如何执行和连接到使用脚本。

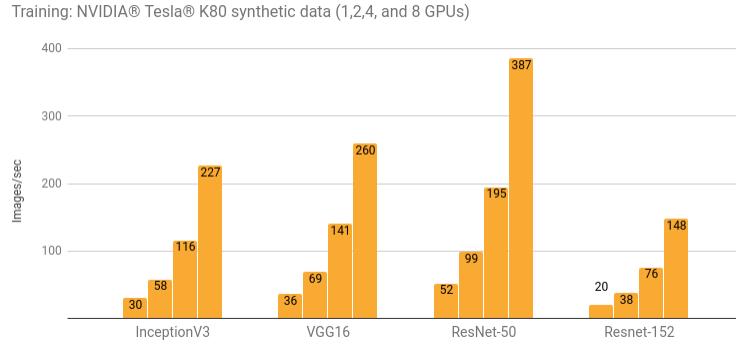
### 6.2.2 图形分类模型的结果

InceptionV3([arXiv:1512.00567](#)), ResNet-50([arXiv:1512.03385](#)), ResNet-152([arXiv:1512.03385](#)), VGG16([arXiv:1409.1556](#)) 和 AlexNet 使用 ImageNet 数据集。测试运行在 Google Compute Engine, Amazon Elastic Compute Cloud(Amazon EC2), 和 NVIDIA DGX-1。多数测试运行在合成和真实数据上。测试合成数据通过使用 tf.Variable 设置相同的形状作为 ImageNet 模型的数据期望。我们相信在一个平台的基准测试上它是很有用的。这载入基础的永健和实际训练准备数据的框架测试。我们结合合成数据移除磁盘 I/O 作为变量设置 baseline。真真的数据用于验证 TensorFlow 输入 pipeline 和基础的磁盘 I/O 在计算单元上是饱和。

### 6.2.3 在 NVIDIA DGX-1(NVIDIA Tesla P100)

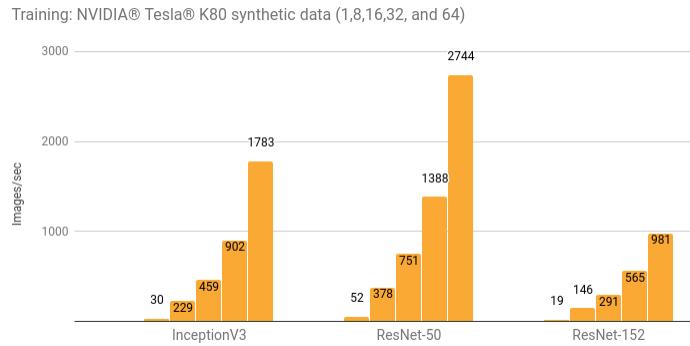


详细的额外的结果在[Details for NVIDIA® DGX-1 \(NVIDIA TeslaP100\)](#)



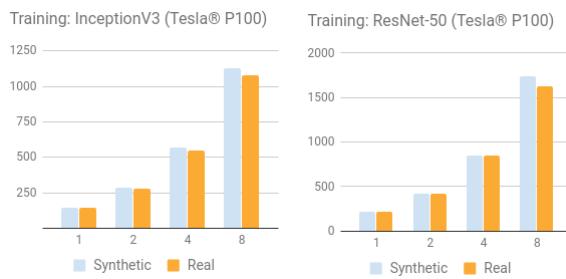
详细的结果在[Details for Google Compute Engine \(NVIDIA Tesla K80\)](#) 和[Details for Amazon EC2 \(NVIDIA Tesla K80\)](#)

#### 6.2.4 用 Tesla K80 分布式的训练



详细的信息在??

#### 6.2.5 结合真是数据训练比较



NVIDIA Tesla K80



### 6.2.6 详细的环境

- 实例类型:NVIDIA DGX-1
- GPU:8x NVIDIA Tesla P100
- OS: 通过 Docker 特使运行 Ubuntu 16.0.4 LTS
- CUDA/cuDNN”8.0/5.1
- TensorFlow GitHub b1e174e
- Benchmark GitHub hash:9165a70
- 构建命令:bazel build -c opt --copt=-march=“haswell” --config=cuda //tensorflow/tools/pip\_package:build\_pip\_package
- Disk: 本地 SSD
- Dataset:ImageNet
- Test Data:2017 五月

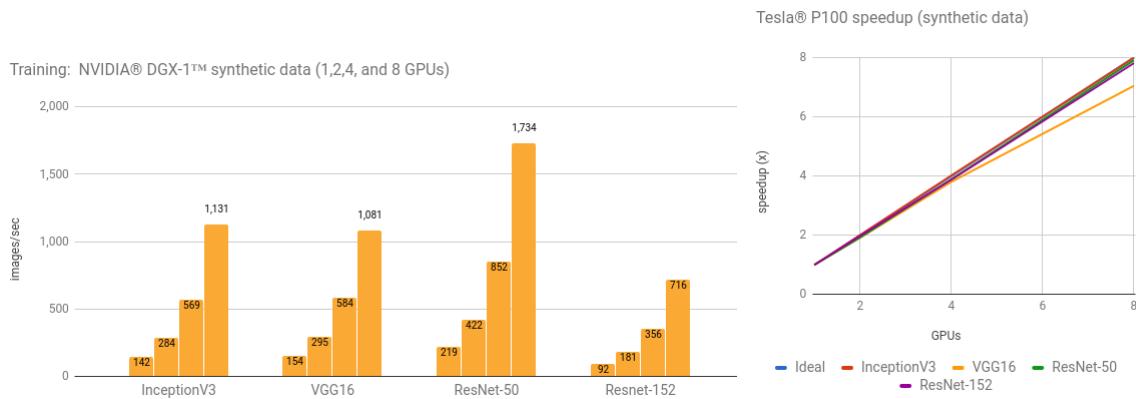
用于模型的批大小和优化器在下表。另外在表中的批大小, Inception V3, ResNet-50, ResNet152 和 VGG16 用于在批大小为 32 上测试, 这些值在其他的章节:

选项	Inception v3	ResNet-50	ResNet-152	AlexNet	VGG16
每个 GPU 批的大小	64	64	64	512	64
优化器	sgd	sgd	sgd	sgd	sgd

为每个模型配置:

模型	变量更新	本地变量设备
InceptionV3	参数服务器	cpu
ResNet50	参数服务器	cpu
ResNet152	参数服务器	cpu
AlexNet	参数服务器	cpu
VGG16	参数服务器	cpu

结果:



### 训练聚合数据

GPUs	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
1	142	219	91.8	2987	154
2	284	422	181	5658	295
4	569	852	356	10509	584
8	1131	1734	716	17822	1081

### 训练真是数据

GPUs	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
1	142	218	91.4	2890	154
2	278	425	179	4448	284
4	551	853	359	7105	534
8	1079	1630	708	N/A	898

在 8GPUs 结合真实数据训练 AlexNet 从图上包含上面的表格最大化输入 pipeline。其他

GPUs	InceptionV3	ResNet-50	ResNet-152	VGG16
1	130	193	82.4	144
2	257	369	159	253
4	507	760	317	457
8	966	1410	609	690

的结果

下面的结果每批 32 训练综合数据

GPUs	InceptionV3	ResNet-50	ResNet-152	VGG16
1	128	195	82.7	144
2	259	368	160	281
4	520	768	317	549
8	995	1485	632	820

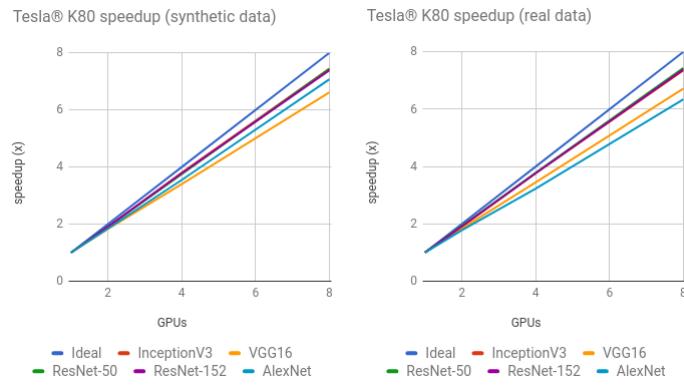
训练真实数据 在 Google Compute Engine 上详情 (NVIDIA Tesla K80)

- 实例类型:n1-standard-32-k80x8
- GPU:8 NVIDIA Tesla K80
- OS Ubuntu 16.04 LTS
- CUDA/cuDNN:8.0/5.1
- TensorFlow GitHub hash:b1e174e
- Benchmark Github hash:9165a70
- 构建命令 `bazel build -c opt --copt=-march="haswell" --config=cuda //tensorflow/tools/pip_package:build_pip_package`
- Disk:1.7TB 共享 SSD 永久磁盘
- Dataset:ImageNet
- Test Data:2017 5 月

每个模型的批大小和优化器在下表中。另外批大小列在表格，InceptionV3 和 ResNet-50 被测试一个 batch 为 32，这些结果在其他章节。

Options	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
Batch size per GPU	64	64	32	512	32
Optimizer	sgd	sgd	sgd	sgd	sgd

这配置用于每个模型 variable\_update 等于 parameter\_server 和 local\_parameter\_device 等于 cpu。结果



GPUs	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
1	30.5	51.9	20.0	656	35.4
2	57.8	99.0	38.2	1209	64.8
4	116	195	75.8	2328	120
8	227	387	148	4640	234

表 6.1: Training synthetic data

GPUs	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
1	30.6	51.2	20.0	639	34.2
2	58.4	98.8	38.3	1136	62.9
4	115	194	75.4	2067	118
8	225	381	148	4056	230

表 6.2: 训练真实数据

GPUs	InceptionV3 (batch size 32)	ResNet-50 (batch size 32)
1	29.3	49.5
2	55.0	95.4
4	109	183
8	216	362

表 6.3: Training real data

6.2.7 Amazon Ec2 详 情 (NVIDIA Tesla K80) 环境

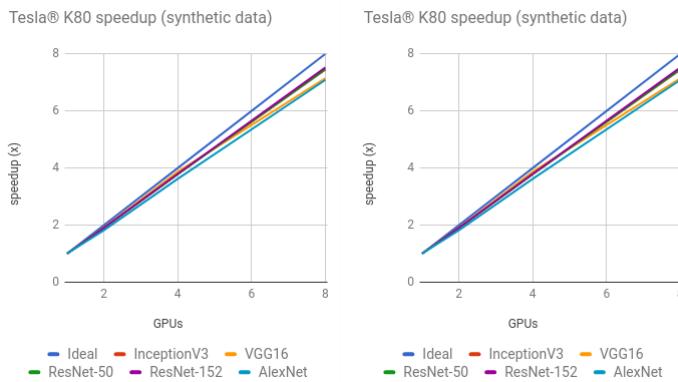
- 实例类型:p2.8xlarge
- GPU:8x NVIDIA Tesla K80
- OS:Ubuntu16.0.4 LTS
- CUDA/cuDNN:8.0/5.1
- TensorFlow Github hash:b1e174e
- Benchmark GitHub hash:9165a70
- 构建命令`bazel build -c opt --copt=-march="haswell" --config=cuda /tensorflow/tools/pip_package:build_pip_package`
- Disk:1TB Amazon EFS(brush 100MiB/sec for 12 hours,continous 50 MiB/sec)
- Dataset:ImageNet
- TestData:2017 5 月

用于每个模型的 Batch size 和优化器在下表。另外 batch size 列出在表格中，InceptionV3 和 ResNet-50 用于在 batch 为 32 上测试。这些结果在其它章节 用于每个模型的配置

Options	InceptionV3	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
Batch size per GPU	64	64	32	512	32	
Optimizer	sgd	sgd	sgd	sgd	sgd	

Model	variable_update	local_parameter_device
InceptionV3	parameter_server	cpu
ResNet-50	replicated (without NCCL)	gpu
ResNet-152	replicated (without NCCL)	gpu
AlexNet	parameter_server	gpu
VGG16	parameter_server	GPUs

## 结果



GPUs	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
1	30.8	51.5	19.7	684	36.3
2	58.7	98.0	37.6	1244	69.4
4	117	195	74.9	2479	141
8	230	384	149	4853	260

表 6.4: Training synthetic data

GPUs	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
1	30.5	51.3	19.7	674	36.3
2	59.0	94.9	38.2	1227	67.5
4	118	188	75.2	2201	136
8	228	373	149	N/A	242

结合 8GPU 从图上和表格训练真实数据因为我们的 RFS 设置没有提供足够的吞吐量。

其他的结果

GPUs	InceptionV3 (batch size 32)	ResNet-50 (batch size 32)
1	29.9	49.0
2	57.5	94.1
4	114	184
8	216	355

表 6.5: Training synthetic data

GPUs	InceptionV3 (batch size 32)	ResNet-50 (batch size 32)
1	30.0	49.1
2	57.5	95.1
4	113	185
8	212	353

表 6.6: Training real data

### 6.2.8 在 Amazon EC2 上 (NVIDIA Tesla K80)

- Instance type: p2.8xlarge
- GPU: 8x NVIDIA® Tesla® K80
- OS: Ubuntu 16.04 LTS
- CUDA / cuDNN: 8.0 / 5.1
- TensorFlow GitHub hash: b1e174e
- Benchmark GitHub hash: 9165a70
- Build Command: `bazel build -c opt --copt=-march="haswell" --config=cuda //tensorflow/tools/pip_package:build_pip_package`
- Disk: 1.0 TB EFS (burst 100 MB/sec for 12 hours, continuous 50 MB/sec)
- DataSet: ImageNet
- Test Date: May 2017

Model	variable_update	local_parameter_device	cross_replica_sync
InceptionV3	distributed replicated	n/a	True
ResNet-50	distributed replicated	n/a	True
ResNet-152	distributed replicated	n/a	True

用于测试的批和优化器在下表。另外批大小在表格中，InceptionV3 和 ResNet-50 用于在批大小为 32 上测试。这结果在其它章节。

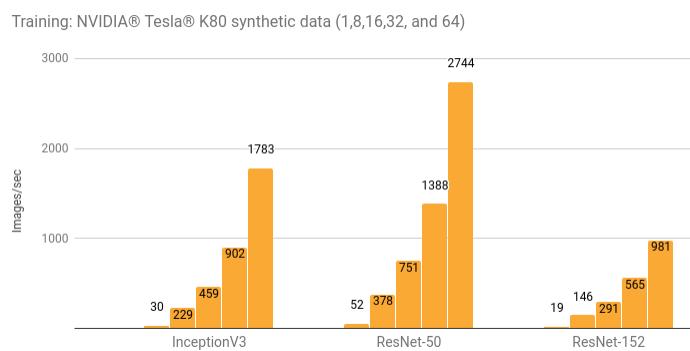
Options	InceptionV3	ResNet-50	ResNet-152
Batch size per GPU	64	64	32
Optimizer	sgd	sgd	sgd

### 用于每个模型的配置

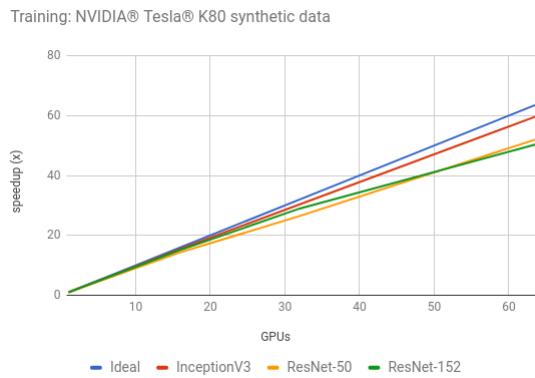
为了简化服务器设置，EC2 实例 (p2.8xlarge) 运行在 worker 服务器上也运行在参数服务器上。参数服务器数量和 worker 服务器用：

- InceptionV3:8 实例/参数服务器
- ResNet50:(批大小为 32)8 实例/4 参数服务器
- ResNet-152:8shili /4 参数服务器

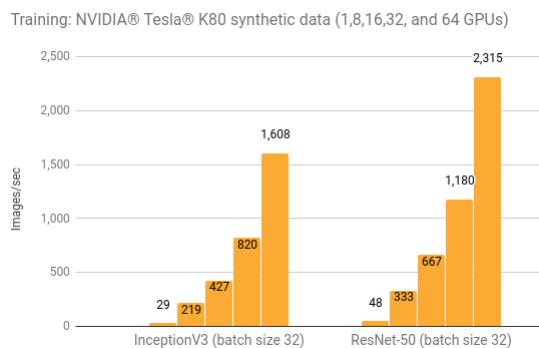
### 结果



GPUs	InceptionV3	ResNet-50	ResNet-152
1	29.7	52.4	19.4
8	229	378	146
16	459	751	291
32	902	1388	565
64	1783	2744	981



## 其他结果



GPUs	InceptionV3 (batch size 32)	ResNet-50 (batch size 32)
1	29.2	48.4
8	219	333
16	427	667
32	820	1180
64	1608	2315

### 6.2.9 方 法 论

这个脚本运行在多个平台生成上面的结果。High-Performance Models如何执行脚本的例子的详细技术在脚本中。

为了创建结果可重复，没个测试运行 5 次然后时间被平均。GPU 运行在默认给定的平台。对于 NVIDIA Tesla K80 这意味着让 GPU Boost, 10 warmup 步被做然后下一个 100 步被平均。



## Chapter 7

# 常用的 python 模块

### 7.1 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，  
argaprsr 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
   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))
```

```
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

## 7.1.1 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)

## 7.1.2 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 useage message。

```

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

```

(a) name of the subfigure      (b) name of the subfigure

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()

```

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()

```

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

```
npc@npc-322:~/TensorFlow_Notebooks$ python code/demo7.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。

ArgumentDefaultsHelpFormatter: 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/foo 就可以设置 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 ArgumentParser 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

就得参数时有用的，为了得到参数的行为值'resolcve' 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 ArgumentParser 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

```

### 7.1.3 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: 传递参数读额数据类型。
- choices: 参数的允许值的容器。
- required: 是否命令行选项被乎略。
- 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(types=[<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)s 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('%s or %s' % (namespace, values, option_string))
7         setattr(namespace, self.dest, values)
8 parser = argparse.ArgumentParser()
9 parser.add_argument('--foo', action=FooAction)
10 parser.add_argument('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=int)
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 support 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 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')

```

```

1 parser.add_argument('--foo', action='store_true', help='foo the bars before
                     frobbing')
2 parser.add_argument('bar', nargs='+', help='foo the bars before frobbed')
3 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]

positional 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

positional 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 YY

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 meiyiou 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 可能当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 arser.parse_args ['-bad WOOD'.split()]
7 Namespace(bacon=None, badger='WOOD')
8 arser.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 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。
- matavar: 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。

```

>>> parser = argparse.ArgumentParser()
>>> 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'])
6Namespace(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)
262.0
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_argparsers()` 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'])
8Namespace(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', type=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=<`_io.FileIO` 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_argument_group('group2', 'group2 description')
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` 将确保惟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 时 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\_know\_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，记住显现出的前向，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 \%prog, \%(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")

```

```

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.verbosity:
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.verbosity 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", "--verbosity", type=int, 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)

```

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:

```

1 #args10.py
2 import argparse
3 parser = argparse.ArgumentParser()

```

```

4 group = parser.add_mutually_exclusive_group()
5 parser.add_argument("-v", "--verbose", action="store_true")
6 group.add_argument("-q", "--quit", action="store_true")
7 parser.add_argument("x", type=int, help="The base")
8 parser.add_argument("y", type=int, help="The exponent")
9 args = parser.parse_args()
10 answer = args.x**args.y
11 if args.quit:
12     print(answer)
13 elif args.verbose:
14     print("{} to the power {} equals {}".format(args.x, args.y, answer))
15 else:
16     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。

## 7.2 path

### 7.2.1 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)`: 删去冗余得分和服, 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 `samefile()` 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。

## 7.2.2 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))

```

### 7.2.3 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                                Traceback (most recent call last)
6 <ipython-input-21-0ec9cf64e174> in <module>()
7     1 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                                Traceback (most recent call last)
6 <ipython-input-2-51d3384e9645> in <module>()
7     1 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，套接字等）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 shyingopen() 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 = r.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/tmp4dwoxyf
```

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 delete=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 makedirs() 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.makedirs()
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.makedirs()
```

```

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。6. 序列化 Python 对象你需要将一个 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

到某个库可以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)
4b'\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
4T-minus 29
5T-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)
3countdown.Countdown object at 0x10069e2d0>
4T-minus 19
5T-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。

## 7.3 正则表达式介绍

### 常用的正则表达式

- ‘^\$’: 表示空白行
- ‘oo\*’: 至少两个 o。
- ‘g.\*g’: 表示 g...g。
- ’t[ae]st’: 搜索 tast 或者 test
- ‘[^g]oo’: 表示 oo 但是 oo 的前面不能为 g
- ‘[a-zA-Z0-9]’: 表示 a 到 z, A-Z,0-9 之间的字符
- ‘^[a-z]’: 小写字母开头
- ‘^[^a-zA-Z]’: 表示首字符不是英文字符
- ‘\.\$’: 表示小数. 结尾
- ‘g..d’: 表示 gd 之间有两个字符
- ‘[0-9][0-9]\*’: 查找任意数字
- ‘o\{2\}’: 表示查找 o 两次
- ‘go\{n,m\}g’: 表示查找 goog 或者 gooog
- ‘g[abc]g’: 表示查找 gag, gbg 或者 gcg

### 扩展正则表达式

- ‘go+d’:+ 表示前面的 o 出现了一次以及以上
- ‘go?d’:? 表示前面的 o 出现一次或者零次, gd 或者 god
- ‘gd|good’: 表示 gd 或者 good
- ‘g(la|oo)d’: 表示 glad 或者 good

sed [-nefr] [动作]:

参数:

-n : 使用 silent 模式。在一般的 sed 用法中，所有来自 STDIN 的数据一般都会被输出到屏幕上。但是如果加上-n 参数后，则只有警告 sed 处理的行（或者动作）才会被列出来

-e : 直接在指令列模式下进行 sed 的动作编辑

-f : 直接将 sed 动作写入一个文件内，-f filename 则可以执行 filename 内的 sed 动作

-r : sed 的动作支持的是扩展型正则表达式语法而不是预设的基本正规表达式语法

动作 : [n1[,n2]]function,n1,n2 不一定存在，一般表达选择进行动作的行数。

function a : 新增，a 的后面可以接字符串，二这些字符串会在新的一行出现（目前行的下一行）

c : 取代，c 后面可以接字符串，这些字符串可以取代 n1,n2 之间的行

d : 删除，因为是删除，所以 d 后面没有任何东西

i : 插入，i 后面接字符串，而这些字符串在新的行出现（当前行的上一行）

p : 打印，将摸个选择的数据输出。通常 p 会和参数 sed -n 一起

s : 取代,可以直接进行取代的工作,通常这个s可以搭配正则表达式.例如 1,2s/old/new/g

#### 例子

- cat -n test.tex|sed '1,2d': 将 test.tex 文件的第一行和第二行删除，并不改变 test.tex 文件的内容
- cat -n test.tex|sed '1a endfigure': 在第一行后（占据第二行）添加 endfigure
- cat -n test.tex|sed '1a beginfigure\\>endfigure': 在第一行末尾输入 \ 回车然后输入第二行内容
- cat -n test.tex|sed '1,3c documentclass': 将 1,3 行内容替换为 doucumentclass
- cat -n test.tex|sed -n '1,3p' : 将 test.tex 文件的 1-3 行打印出来, 加上 n 参数为了显示最后输出，而不是打印一行输出一下。

awk 工具 awk ' 条件类型 1 动作 1 条件类型 2 动作 2'filename:awk 乐意处理后续的档案，也可以读取来自前个制定的标准输出。但是 wak 主要处理每一行的字段的内容而预设的字段分隔符为空格键或者 tab 键。

#### 例子

- last|awk 'print \$1 "\t" \$3' : 查看当前登录用户, 每一行都有变量名称, \$1 表示第一列,\$0 代表全部

变量名称	代表含义
NF	每行拥有的字段总数
NR	目前 awk 所处理的是第几行的数据
FS	目前的分割字符, 预设是空格键

示例:

- last |awk 'print \$1 "\t lines:" \$NR "\t columns:" \$NF': 处理第一行, 将第一列取出打印然后输出制表符分割加上自己需要加上的打印信息 lines:, 输出变量行数, 然后输出制表符输出列数

awk 的逻辑运算字。

运算单元	代表含义
<	小于
>=	大于等于
<=	小于等于
==	等于
!=	不等于

例子

- ```

1 cat /etc/passwd |\
2 > awk '{FS=":"} $3<10 {print $1 "\t" $3}'
3

```

:passwd 中的内容是用: 分隔的, 此条命令查看第三列小于 10 的数据, 并且; 列出第三列 (第一行没有正确显示, 这是因为我们读入第一行的时候, 把鞋变量 \$1,\$2,... 预设还是以空格为分割的, 所以虽然定义了 FS=":", 但却仅仅只能在第二行后开始生效)

- ```

1 cat /etc/passwd|\
2 > awk 'BEGIN {FS=":"} $3<10 {print $1 "\t" $3}'
3

```

操作符	说明	例子
.	任何单个字符	
[^]	非字符集字符, 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

表 7.1: 常用参数

- ...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。

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

(?:...) 非捕获版本的正则括号，匹配括号中无论什么正则表达式，但是在执行一个匹配或者查询之后 group 中子字符串匹配不能被获得

\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 对于 unicode 字符串类型：匹配 unicode 空格字符串（包括 [\t\n\r\f\v]，因此一些其它字符，例如不间断的空格），如果 ascii flag 被用，仅仅 [\t\n\r\f\v] 被匹配（但是 flag 影响整个正则表达我时），因此在这样的情况下用 [\t\n\r\f\v] 也许是更好的选择。

\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, 不小号任何字符串。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]

## 7.4 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。

## 7.4.1 re

## 表 达 式 中 的

## flags

<code>re.A</code>	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。
<code>re.DEBUG</code>	显示编译表达式的调试信息
<code>re.I</code>	忽略正则表达式的大小写，[A-Z] 能够匹配小写。
<code>re.L</code>	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。
<code>re.M</code>	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

```

1 a = re.compile(r"""
2     \d + # the integral part
3         \. # the decimal point
4         \d * # some fractional digits""", re.X)
5 b = re.compile(r"\d+\.\d*")

```

`re.error(msg, pattern=None, pos=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一个位置，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'

## 7.5 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。
- `sts.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 <sup>(e-1)</sup> 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，getalloctedblocks() 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  
 platform 将被 2(VER\_PLATFORM\_WIN32\_NI) (VER\_NT\_DOMAIN\_CONTROLLER) 是系统是域  
 (VER\_NT\_SERVER) 系统是服  
 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 firsttier 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"> <li>• sys.has_info: 数值 hash 实现参数给一个结构序列。</li> </ul> | width hash 值的位宽<br>modulus 用于数值 hash 方案的主要模块 P<br>inf 返回正无穷大的 hash 值<br>nan 返回非数的 hash 值<br>imag 返回复数的虚部<br>algorithm str,bytes 和 memoryview 的 hash 算法的实现<br>hash_bits hash 算法的内部输出大小<br>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.

## 7.6 collections

collections 是 Python 内建的一个集合模块，提供了许多有用的集合。

### 7.6.1 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'])
```

### 7.6.2 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'])
```

### 7.6.3 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。

### 7.6.4 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)

```

### 7.6.5 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})

```

## 7.7 base64

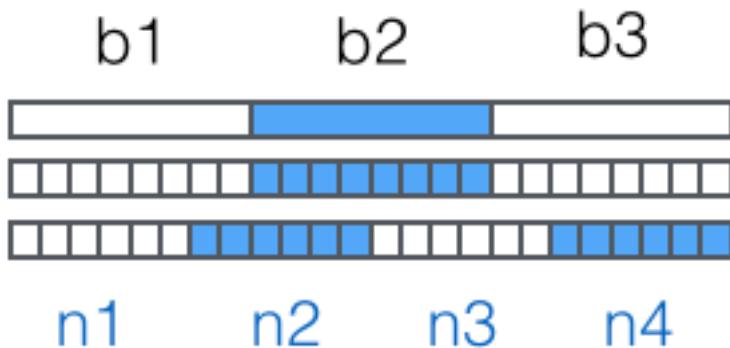
base64 是一种用 64 个字符来表示任意二进制数据的方法。

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“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。

## 7.8 struct

## 7.9 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。

## 7.10 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其。所以上述代码会一直打印。

### 7.10.1 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，他不会事先把无线个元素生成出来，是时尚也不可能在内存中创建无线多个元素/无线序列虽然可以无线迭代下去，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]
```

### 7.10.2 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']

```

## 7.11 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()
```

## 7.12 XML

## 7.13 HTMLParser

## 7.14 ZipFile

7.15 url

7.15.1 urllib.request

## 7.16 requests

### 7.16.1 发送请求

```

1 import requests
2 r = requests.get('https://github.com.timeline.json')
3 r = requests

```

### 7.16.2 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。

### 7.16.3 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

#### 7.16.4 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。

#### 7.16.5 理解 Requests 库的异常

异常	说明
requests.ConnectionError	网络链接错误异常, 如 DNS 查询失败, 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, 不需要额外的 if 语句, 该语句便于利用 try-except 进行异常处理。

#### 7.16.6 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 8

## Bazel

### 8.1 Bazel start

#### 8.1.1 用 工 作 空 间

所有的构建发生在一个存在于你的文件系统并且包含你想构建软件的源代码的目录 workspace 中，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做 WORKSPACE 的顶层目录；一个空文件是一个可用的 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
```

#### 8.1.2 创 建 一 个 构 建 文 件

为了知道你的工程中什么目标能被构建，Bazel 查看 BUILD 文件。它的语法类似 Python 被写进 Bazel 的构建语言中。通常它们仅仅是一个声明规则的序列。每个规则指定它的输入，输出和一个从输入到输出的计算方法。

规则对于那些已经在[genrule](#)前用过 Makefile 的人来说通常很熟悉，可以通过 shell 命令指定如何生成输出。

```
1 genrule(            
2     name = ‘‘hello’’,             
3     outs = [‘‘hello_world.txt’’],   
4     cmd = ‘‘echo Hello World > $@’’,   
5 )
```

shell 命令包含[make 变量](#) 用上面的 BUILD 文件你可以要求 Bazel 生成目标

```

1 $ bazel build :hello
2 .
3 INFO: Found 1 target...
4 Target //:hello up-to-date:
5   bazel-genfiles/hello_world.txt
6   INFO: Elapsed time: 2.255s, Critical Path: 0.07s

```

我们注意到两件事，第一：目标通常通过它们的label访问 label 规则的name属性指定，第二：Bazel 将生成的文件放进一个分割的目录 (bazel-genfiles 目录通常是符号链接) 只要不污染你的源树。

规则用其它规则的输出作为输入，下面的例子，再一次生成的源代码被它们的 label 访问：

```

1 genrule(
2   name = "'hello'",
3   outs = ['hello_world.txt'],
4   cmd = "echo Hello World > $@",
5 )
6 genrule(
7   name = "double",
8   srcs = [":hello"],
9   outs = ["double_hello.txt"],
10  cmd = "cat $< $< > $@",
11 )

```

最后注意生成规则看着很类似，它不是最好的规则。它是为不同的语言指定规则的首选。

### 8.1.3 下一步

在 Java 和 C++ 程序上构建。

## 8.2 构建 C++ 工程

在这个导航中，你将学习通过 Bazel 构建 C++ 应用的基础，你讲设置你的 workspace 建立一个简单的工程说明关键的 Bazel 概念，像 target 个 BUILD 文件，在完成这个导航后，查看[Common C++ Build User Cases](#)了解更多像写，运行 C++ 测试的高级概念。

### 8.2.1 你将学习练习

- 构建目标
- 可视化项目依赖
- 分割项目成多个目标和包

- 在包上控制目标可视化
- 通过 labels 访问目标

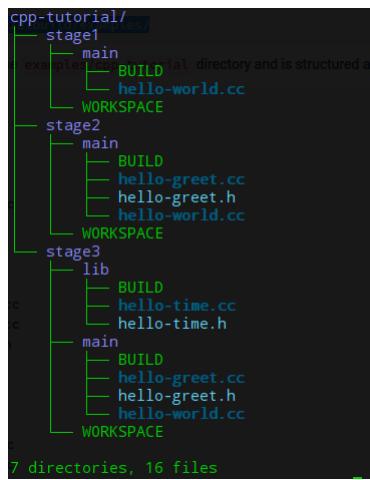
### 8.2.2 准备

备

开始之前你需要确保安装合适的 Bazel，然后访问 Bazel 的 GitHub 仓库获取示例工程：

```
git clone https://github.com/bazelbuild/examples/
```

示例工程的 examples/cpp-tutorial 目录结构如下：



如你所见导航中有一些文件，每个集合代表一个 stage，你将构建一个单独的目标到单独的包，第二个 stage 你讲分割你的工程为多个目标但是保持在一个单独的包中。在第三个和最后的 stage，你讲分割你的工程为多个 package 并且用不同的 targets 构建它。

### 8.2.3 构建

Bazel

#### 建立你的 workspace

workspace

在你可以构建你的工程之前，你需要建立你的 workspace。一个 workspace 是一个保存你工程的源代码文件和 Bazel 的构建输出的目录。它也包含 Bazel 识别为特殊文件的文件：

- WORKSPACE 文件识别一个目录和它的内容作为一个 Bazel workspace 同时生成工程的根目录结构。
- 一个或者更多的 BUILD 文件告诉 Bazel 如何构建工程的不同部分。（在 workspace 的一个目录包含 BUILD 文件是一个包。你将在之后的导航中学习包）

为了设计一个目录作为 Bazel workspace，在目录中创建一个空的名字为 WORKSPACE 的文件，当 Bazel 构建工程的时候，所有的输入和依赖必须在同一个 workspace。文件保留在不同的 workspace 是是独立于其它的文件的除非被链接。

#### 8.2.4 明白 BUILD 文件

一个 BUILD 文件包含一些针对 Bazel 的不同类型的指令。最重要的类型是 build rule，它告诉 Bazel 如何构建想要的输出，像执行二进制或者库。在 BUILD 文件中的构建规则的实例称为 target，指定一个指定的源代码文件和依赖。一个 target 也指出其它的目标。

查看在 cpp-tutorial/stage1/main 目录中 BUILD 文件

```
1 cc_binary(
2     name = "hello-world",
3     srcs = ["hello-world.cc"],
4 )
```

在我们的例子中，hello-world 目标实例 Bazel 的**构建cc\_binary rule**。规则告诉 Bazel 从 hello-world.cc 源文件的构建一个没有任何依赖自我包含执行二进制。目标中的属性明确了它的依赖和选项状态。当 name 属性是命令，一些选项，例如 hello-greet 目标 name 是自说明，srcs 指定 Bazel 构建目标的源文件。

#### 8.2.5 构建工 程

让我们建立你的样本工程，进入 cpp-tutorial/stage1 目录运行下面的代码

```
1 bazel build //main:hello-world
```

注意目标 label//main: 部分是 BUILD 文件获取 workspace 根的位置，hello-world 是我们在 BUILD 文件中命名目标的名字。

Bazel 将产生类似下面的输出：

```
1 INFO: Found 1 target ...
2 Target //main:hello-world up-to-date:
3     bazel-bin/main/hello-world
4 INFO: Elapsed time: 6.817s, Critical Path: 0.54s
```

恭喜你你已经构建了你的第一个 Bazel 目标！Bazel 防止 build 输出在 workspace 的根目录的 bazel-bin 目录。通过它的内容得到 Bazel 的输出结构，现在测试你的构建库：

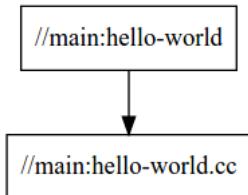
```
1 bazel-bin/main/hello-world
```

## 8.2.6 回顾依赖图

一个成功的构建在 BUILD 文件有一些明确状态的依赖。Bazel 用这些状态创建工程的依赖图，是的增量构建更精确。让我们可视化我们样例工程依赖，首先生成一个依赖图的文字表示（在 workspace 根目录运行命令）：

```
1  bazel query --nohost_deps --noimplicit_deps 'deps(//main:hello-world)' \
2  --output graph
```

上面的命令告诉 Bazel 为目标 //main:hello-world 查找所有的依赖（包括多数依赖和隐藏依赖）输出为图表格式，然后粘贴文字到 GraphViz。正如你所见，样例工程的第一个 stage 有一个目标没有额外依赖构建一个单个源文件：



注意，你已经设置你的 workspace，构建你的项目检查它的依赖，让我们增加一点复杂度。

## 8.2.7 提炼你的 Bazel 构建

尽管对于小的工程一个 target 是足够的，你也许将分割大的工程为多个目标包，运行快速的增量构建加速你的构建通过同时构建工程的多个部分。

## 8.2.8 指定多个构建目标

让我们分割我们的样本工程为两个目标，查看 `cpp-tutorial/stage2/main` 目录中的 BUILD 文件

```
1 cc_library(
2     name = "hello-greet",
3     srcs = ["hello-greet.cc"],
4     hdrs = ["hello-greet.h"],
5 )
6 cc_binary(
7     name = "hello-world",
8     srcs = ["hello-world.cc"],
9     deps = [
10         ":hello-greet",
11     ],
12 )
```

通过 BUILD 文件 Bazel 首先构建 hello-greet 库 (用 Bazel 的 [构建 cc\\_library rule](#))，然后 hello-world 二进制，在 hello-world 目标的 deps 属性告诉 Bazel hello-greet 库要求构建 hello-world 二进制。构建一个新版，进入 cpp-tutorial/stage2 目录运行下面的命令：

```
1  bazel build //main:hello-world
```

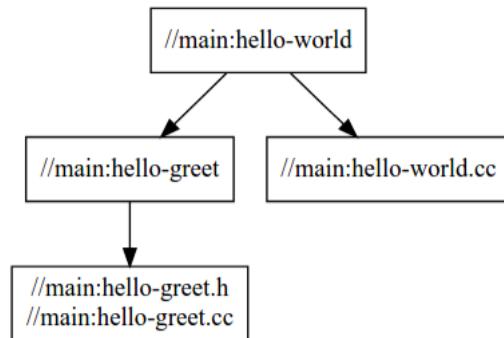
Bazel 生成类似下面的输出：

```
1 INFO: Found 1 target ...
2 Target //main:hello-world up-to-date:
3   bazel-bin/main/hello-world
4 INFO: Elapsed time: 6.625s, Critical Path: 0.76s
```

现在测试你构建的二进制：

```
1  bazel-bin/main/hello-world
```

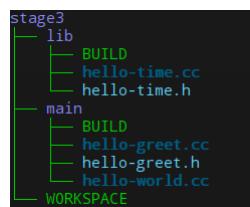
如果你修改 hello-greet.cc，重新构建工程 Bazel 将仅仅重新编译这个文件，查看依赖图，你可以看到 hello-world 依赖和之前相同的输入，但是构建的结构有点不同：你也许构建两



个目标，hello-world 目标构建一个源文件依赖另一个其它的目标（//main:hello-greet），构建两个额外的源文件。

### 8.2.9 用 多 个 包

让我们分割工程为多个包，查看 cpp-tutorial/stage3 目录：



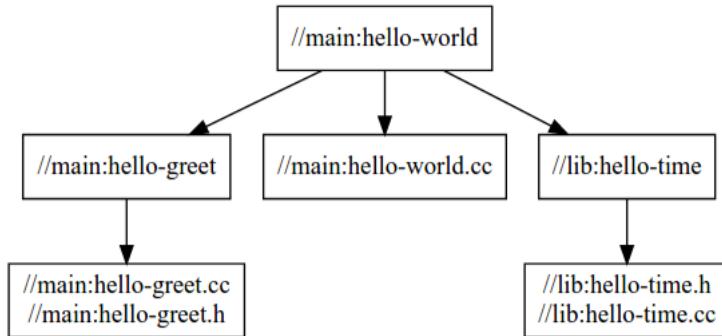
注意我们现在有两个子目录，每个包含一个 BUILD 文件，因此对于 Bazel，workspace 现在包含两个包 lib 和 main. 查看 lib/BUILD 文件:

```
1 cc_library(
2     name = "hello-time",
3     srcs = ["hello-time.cc"],
4     hdrs = ["hello-time.h"],
5     visibility = ["//main:__pkg__"],
6 )
```

main/BUILD 文件:

```
1 cc_library(
2     name = "hello-greet",
3     srcs = ["hello-greet.cc"],
4     hdrs = ["hello-greet.h"],
5 )
6 cc_binary(
7     name = "hello-world",
8     srcs = ["hello-world.cc"],
9     deps = [
10         ":hello-greet",
11         "//lib:hello-time",
12     ],
13 )
```

正如你看到的 main 包中的 hello-world 目标依赖 lib 中的于 hello-time 目标 (因此目标标签 //lib:hello-time) Bazel 知道这通过 deps 属性。查看依赖图: 注意对于成功的构建，我们



在 lib/BUILD 创建一个//lib:hello-time 目标在 main/BUILD 中用 visibility 属性明确的可视化目标。这是因为默认目标仅仅对在同一 BUILD 文件中的其它的目标可见。(Bazel 用目标可视化阻止了包含实现细节泄露到公开的 api 的问题) 当我们构建我们工程的最终版本，进入 `cpp-tutorial/stage3` 目录运行下面的命令:

```
1 bazel build //main:hello-world
```

Bazel 生成类似下面的输出:

```
1 INFO: Found 1 target ...
2 Target //main:hello-world up-to-date:
3   bazel-bin/main/hello-world
4 INFO: Elapsed time: 6.584s, Critical Path: 0.76s
```

现在测试最新的构建二进制文件:

```
1 bazel-bin/main/hello-world
```

你现在将构建有两个包三个目标的工程明白它们之间的依赖。

#### 8.2.10 用 标 签 访 问 目 标

在 BUILD 文件和在命令行, Bazel 用 labels 访问目标, 例如//main:hello-world 或者 lib//hello-time。它们的语法:

```
1 //path/to/package:target-name
```

如果目标是一个规则目标, 然后 path/to/package 是一个包含 BUILD 文件的路径, target-name 是你在 BUILD 文件中命名的名字。如果目标是一个规则目标然后 path/to/package 是 package 的根目录, target-name 是目标文件的名字, 包含完整的路径。当在同一个包中访问目标, 你可以用//:target-name 跳过包。

#### 8.2.11 进 一 步 阅 读

恭喜你! 你现在知道的用 Bazel 构建 C++ 工程的基础, 下一步, 读更常用的[C++ build use cases](#), 然后检查下面:

- [External Dependencies](#)了解更多本地和远程仓库。
- [Build Encyclopedia](#)了解更多 Bazel
- [Java build tutorial](#)开始用 Bazel 构建 Java 程序
- [mobile application tutorial](#)开始用 Bazel 为 Android 和 IOS 构建移动程序。

### 8.3 常 用 的 C++ 构 建 情 况

#### 8.3.1 一 个 目 标 中 有 多 个 文 件

你可以用[glob](#)包含多个文件, 例如:

```

1 cc_library(
2     name = "build-all-the-files",
3     srcs = glob(["*.cc"])
4     hdrs = glob(["*.h"]),
5 )

```

在这个目标中 Bazel 将构建所有在相同目录中找到的.cc 和.h 文件作为包含这个目标的 BUILD 文件

### 8.3.2 用

include

如果一个文件包含一个头文件，然后文件的规则依赖于头文件的库。相关，仅仅直接以来需要制定为依赖。例如，假设 sandwich.h 包含 bread.h 同事 bread.h 包含 flour.h。sandwith.h 不包含 flour.h(在他的 sandwich 中谁想要 flour?) 因此 BUILD 文件将看起来像这样：

```

1 cc_library(
2     name = "sandwich",
3     srcs = ["sandwich.cc"],
4     hdrs = ["sandwich.h"],
5     deps = [":bread"],
6 )
7 cc_library(
8     name = "bread",
9     srcs = ["bread.cc"],
10    hdrs = ["bread.h"],
11    deps = [":flour"],
12 )
13 cc_library(
14     name = "flour",
15     srcs = ["flour.cc"],
16     hdrs = ["flour.h"],
17 )

```

这里 sandwich 库依赖于 bread， bread 依赖于 flour 库。

### 8.3.3 添 加 包 路 径

有时候你不想根目录在 workspace 目录中包含，存在的库也许已经有一个包含目录，但是这个不露不和你的 workspace 路径匹配。例如，假设你有下面的文件结构：

```

1 my-project
2     third_party
3         some_lib

```

```

4      BUILD
5          include
6              some_lib.h
7              some_lib.cc
8
WORKSPACE

```

Bazel 将希望 `some_lib.h` 被包含作为 `third_party/some_lib/include/some_lib.h`, 但是假设 `some_lib.cc` 包含”`include/some_lib.h`”。为了包含路径可用。

`third_party/some_lib/BUILD` 将需要制定 `some_lib`/目录是一个包含目录:

```

1 cc_library(
2     name = "some_lib",
3     srcs = ["some_lib.cc"],
4     hdrs = ["some_lib.h"],
5     copts = ["-Ithird_party/some_lib"],
6 )

```

这对于外部依赖特别有用，尽管他们的头文件必须用一个/前缀包括。

#### 8.3.4 包含外部的库

假设你用[Google Test](#)。你可以在 `WORKSPACE` 文件用一个 `new_` 仓库函数下载 Google Test 使得它是你的可用仓库:

```

1 new_http_archive(
2     name = "gtest",
3     url = "https://github.com/google/googletest/archive/release-1.7.0.zip",
4     sha256 = "b58cb7547a28b2c718d1e38aee18a3659c9e3ff52440297e965f5edffe34b6d0",
5     build_file = "gtest.BUILD",
6 )

```

注意：如果目标已经包含一个 `BUILD` 文件，你可以用一个 `non-new_` 函数

然后创建 `gtest.BUILD` 和 `BUILD` 文件编译 Google Test。Google Test 有一些特别的要求使得他的 `cc_library` 规则更复杂:

- `googletest-release-1.7.0/src/gtest-all.cc` #includes 所有的在 `googletest-release-1.7.0/src/` 的文件，因此我们需要通过编译器执行它或者我们对于一样的符号将得到连接错误
- 它用和 `googletest-release-1.7.0/include/` 目录下的头文件 (`"gtest/gtest.h"`)，因此我们必须添加这个目录到包含路径。
- 它需要连接进 `pthread`，因此我们添加 `linkopt`

完整的规则如下:

```

1 cc_library(
2     name = "main",
3     srcs = glob(
4         ["googletest-release-1.7.0/src/*.cc"],
5         exclude = ["googletest-release-1.7.0/src/gtest-all.cc"]
6     ),
7     hdrs = glob([
8         "googletest-release-1.7.0/include/**/*.h",
9         "googletest-release-1.7.0/src/*.h"
10    ]),
11    copts = [
12        "-Iexternal/gtest/googletest-release-1.7.0/include"
13    ],
14    linkopts = ["-pthread"],
15    visibility = ["/visibility:public"],
16 )

```

这多少有些混乱: 一切都是 googletest-release-1.7.0 的前缀作为一个归档结构的副产品。你可以创建 new\_http\_archive 通过添加 strip\_prefix 属性删除前缀:

```

1 new_http_archive(
2     name = "gtest",
3     url = "https://github.com/google/googletest/archive/release-1.7.0.zip",
4     sha256 = "b58cb7547a28b2c718d1e38aee18a3659c9e3ff52440297e965f5edffe34b6d0",
5     build_file = "gtest.BUILD",
6     strip_prefix = "googletest-release-1.7.0",
7 )

```

然后 gtest.BUILD 看起来像这样:

```

1 cc_library(
2     name = "main",
3     srcs = glob(
4         ["src/*.cc"],
5         exclude = ["src/gtest-all.cc"]
6     ),
7     hdrs = glob([
8         "include/**/*.h",
9         "src/*.h"
10    ]),
11    copts = ["-Iexternal/gtest/include"],
12    linkopts = ["-pthread"],
13    visibility = ["/visibility:public"],
14 )

```

现在 cc\_ 规则可以依赖于 gtest//:main

## 8.3.5 写，运行一个

C++

Test

例如，我们可以创建一个测试像下面这样./test/hello-test.cc:

```

1 #include "gtest/gtest.h"
2 #include "lib/hello-greet.h"
3
4 TEST(HelloTest, GetGreet) {
5     EXPECT_EQ(get_greet("Bazel"), "Hello Bazel");
6 }
```

然后为你的 tests 床键文件./test/BUILD 文件:

```

1 cc_test(
2     name = "hello-test",
3     srcs = ["hello-test.cc"],
4     copts = ["-Iexternal/gtest/include"],
5     deps = [
6         "@gtest//:main",
7         "//lib:hello-greet",
8     ],
9 )
```

注意为了使得 hello-greet 对于 hello-test 可见，我们必须添加”//test:\_\_pkh\_\_”可视化在./lib/BUILD 下的属性。现在你可以运行 test

```

1 bazel test test:hello-test
2 \end{language=Bash}
3 这产生下面的输出:
4 \begin{lstlisting}[language=Bash]
5 INFO: Found 1 test target...
6 Target //test:hello-test up-to-date:
7   bazel-bin/test/hello-test
8 INFO: Elapsed time: 4.497s, Critical Path: 2.53s
9 //test:hello-test PASSED in 0.3s
10
11 Executed 1 out of 1 tests: 1 test passes.
```

## 8.3.6 为 预 编 译 库 添 加 依 赖

如果你想用你有一个编译过的库 (例如，头文件和.so 文件) 打包它进 cc\_library 规则:

```

1 cc_library(
2     name = "mylib",
3     srcs = ["mylib.so"],
4     hdrs = ["mylib.h"],
5 )
```

---

### 8.3. 常用的 C++ 构建情况

用这种方法，在你的 workspace 中的其它的 C++ 目标可能依赖这个规则。



# Chapter 9

## 实践部分

### 9.1 TensorFlow

例 子

#### 9.1.1 CNN

手 写 体 数 据 识 别

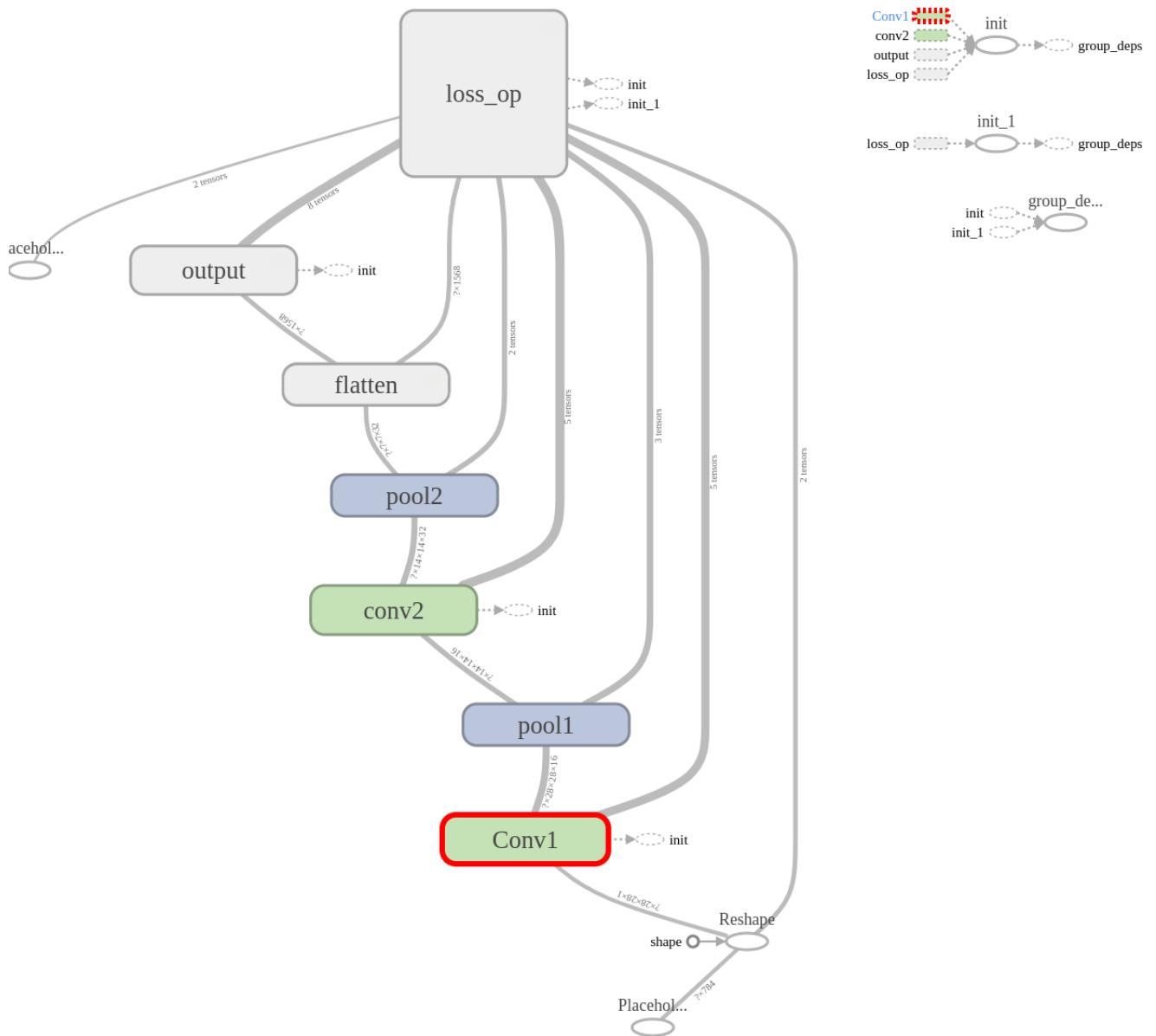
#### 9.1.2 mnist

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  $32 \times 32$  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  $5 \times 5$ ,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  $5 \times 5$ ,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:168。

## 9.1. TENSORFLOW 例子



```

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)

```

```

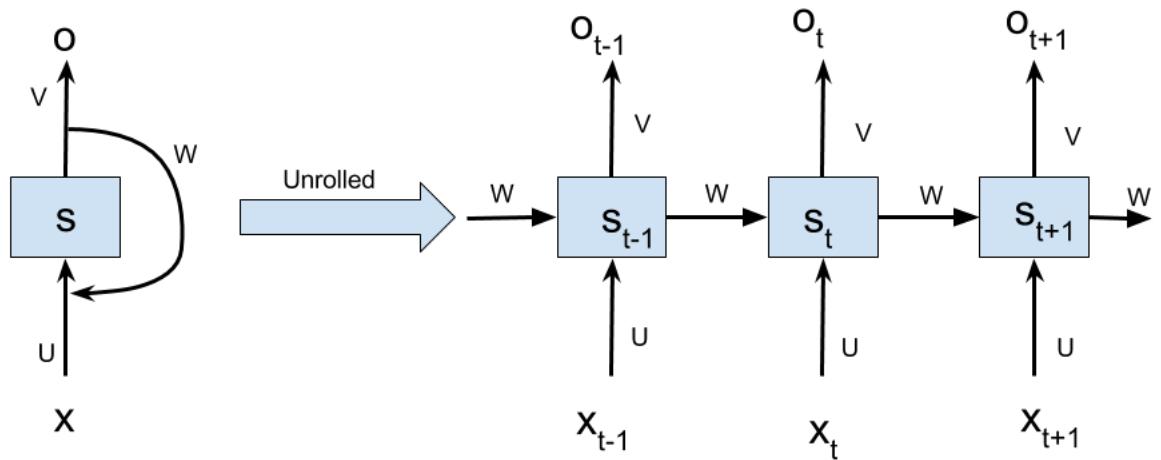
50    tf.summary.scalar('accuracy', accuracy)
51 sess = tf.Session()
52 merge_op = tf.summary.merge_all()
53 init_op = tf.group(tf.global_variables_initializer(), tf.
54                     local_variables_initializer())
55 sess.run(init_op)
56 writer = tf.summary.FileWriter('./log', sess.graph)
57 for 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]})  

67 pred_y = np.argmax(test_output, 1)

```

LSTM 通常用来解决复杂的序列处理问题，比如包含了 NLP 概念（词嵌入、编码器等）的语言建模问题。这些问题本身需要大量理解，那么将问题简化并集中于在 TensorFlow 上实现 LSTM 的细节（比如输入格式化、LSTM 单元格以及网络结构设计），会是个不错的选择 MNIST 就正好提供了这样的机会。其中的输入数据是一个像素值的集合。我们可以轻易地将其格式化，将注意力集中在 LSTM 实现细节上。

循环神经网络按时间轴展开的时候如下图：



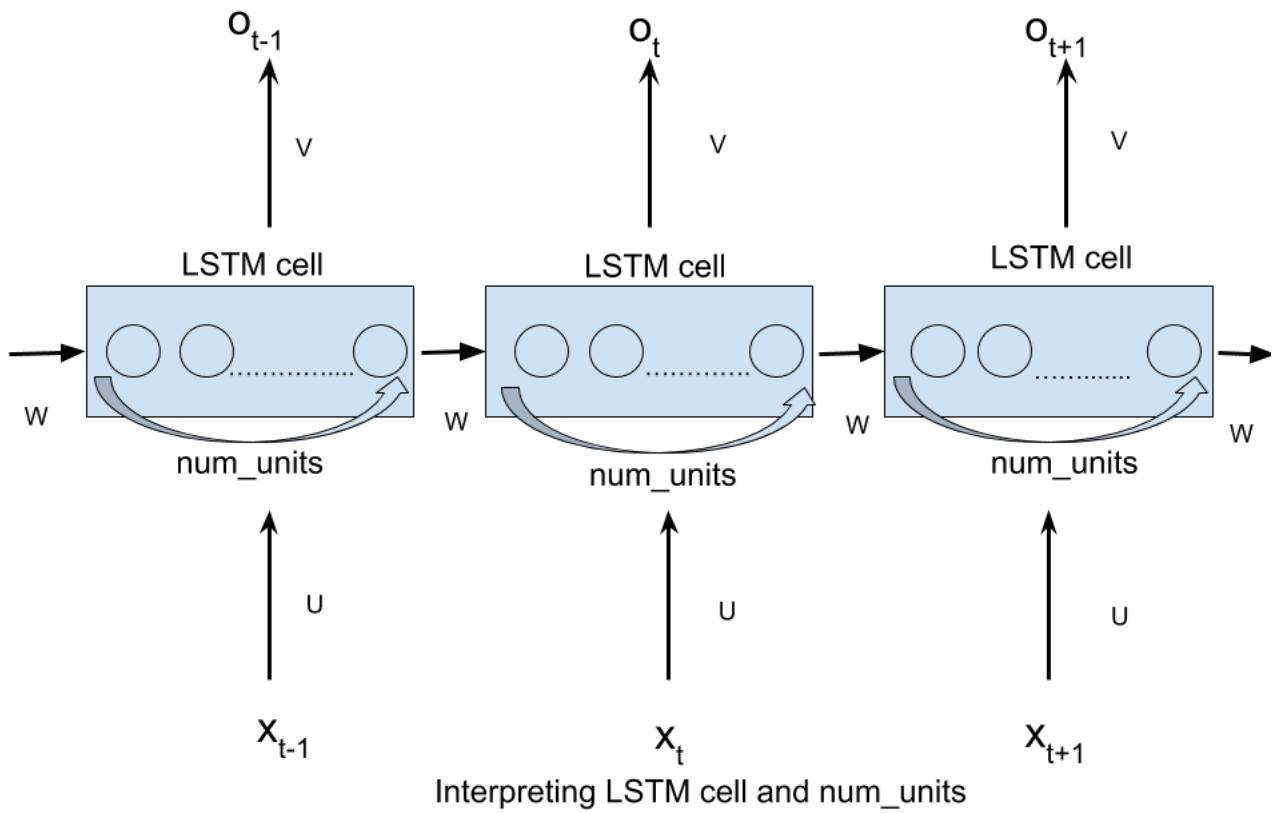
A Recurrent neural network unrolled through time.

图 9.1: 循环神经网络的展开图

1.  $x_t$  代表时间步  $t$  的输入；
2.  $s_t$  代表时间步  $t$  的隐藏状态，可看作该网络的「记忆」；
3.  $o_t$  作为时间步  $t$  时刻的输出；
4.  $U$ 、 $V$ 、 $W$  是所有时间步共享的参数，共享的重要性在于我们的模型在每一时间步以不同的输入执行相同任务。

当把 RNN 展开的时候，网络可被看作每一个时间步都受上一时间步输出影响（时间步之间存在连接）的前馈网络 在 TensorFlow 中基本的 LSTM 单元声明如下：

`tf.contrib.rnn.BasicLSTMCell(null_units)` 这里，`num_units` 指一个 LSTM 单元格中的单元数。`num_units` 可以比作前馈神经网络中的隐藏层，前馈神经网络的隐藏层的节点数量等于每一个时间步中一个 LSTM 单元格内 LSTM 单元的 `num_units` 数量。看下图：

图 9.2: LSTM 单元和 `num_units`

每个 `num_units` LSTM 单元可以看做是一个标准的 LSTM 单元:

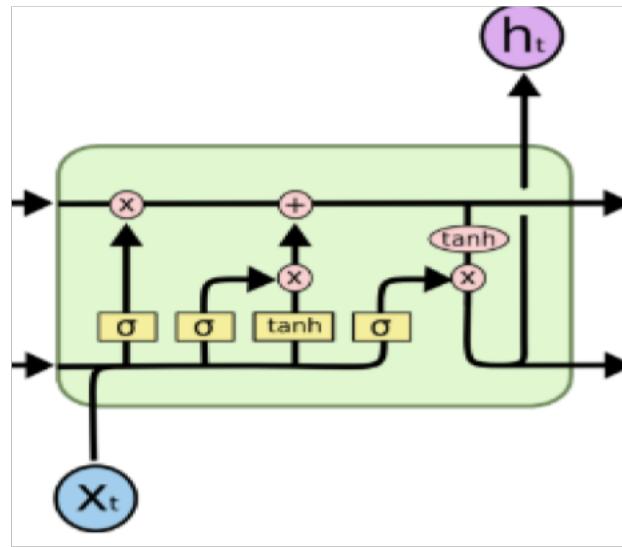


图 9.3: lstm 单元

数据输入 TensorFlow 前先格式化，在 TensorFlow 中最简单的 RNN 形式是 static\_rnn，  
在 TensorFlow 中定义如下：tf.static\_rnn(cell, inputs)，inputs 接受形状为  
[batch\_size, input\_size] 的张量列表。列表的长度为将网络展开后的时间步数，即列表中每  
个元素都分别对应网络展开的时间步。比如 MNIST 数据集中，我们有  $28 \times 28$  像素的图  
像，每张图像都可以看成拥有 28 行 28 个像素的图像。我们将按 28 个时间步展开，以使  
在每个时间步中，可以输入一行 28 个像素 (input\_size)，而仅经过 28 个时间步输入整张  
图像。给定图像的 batch\_size 值，则每一个时间步将分别收到 batch\_size 个图像。如下图

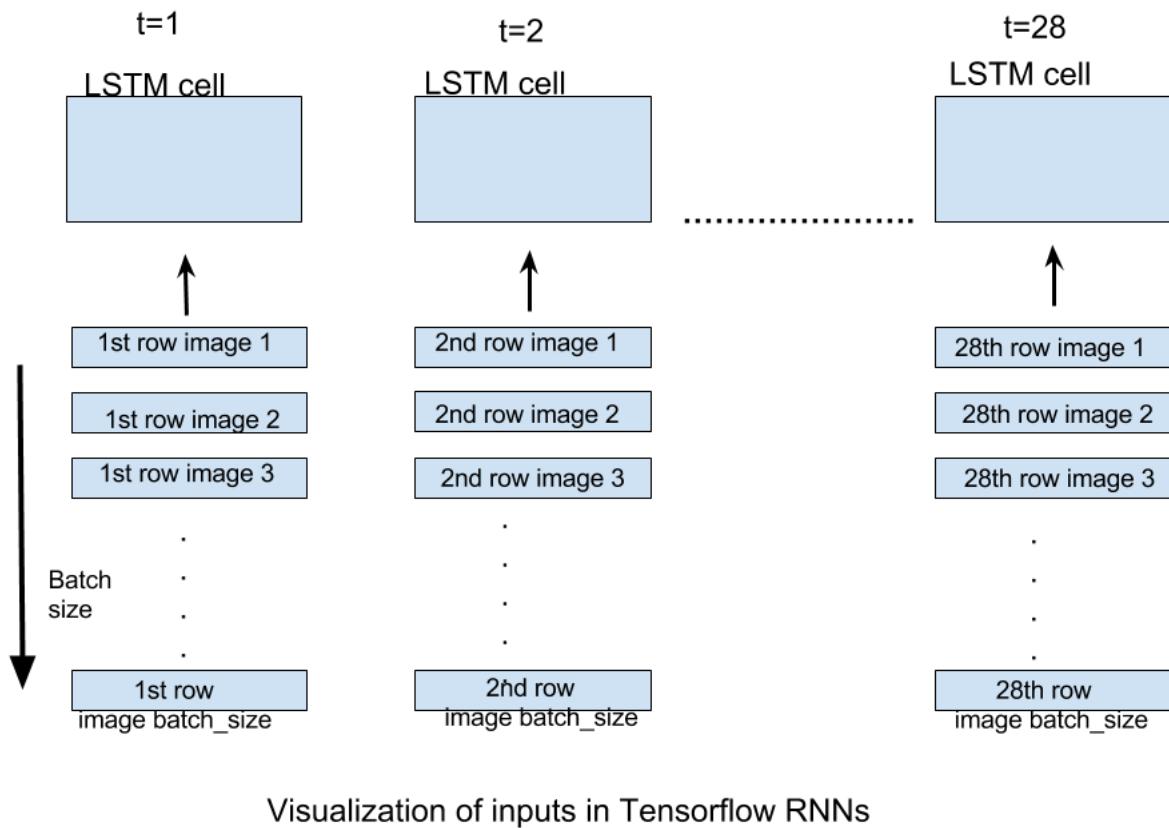


图 9.4: 可视化 TensorFlow RNN

由 `static_rnn` 生成的输出是一个形态为 `[batch_size, n_hidden]` 的张量列表。列表的长度为将网络展开后的时间步数，即每一个时间步输出一个张量。在这个实现中我们只需关心最后一个时间步的输出，因为一张图像的所有行都输入到 RNN，预测即将在最后一个时间步生成。

### 9.1.3 卷 积 神 经 网 络 处 理 序 列 数 据

数据集([HAR Dataset](#))介绍: 该数据集来自于智能手机获取的人活动数据，追踪人的活动状态，总共有如下六种情况：

- 行走
- 上楼梯
- 下楼梯

- 坐立

- 站立

- 平躺

数据有来自加速度计和陀螺仪的数据, 数据集分为测试集和训练集, 训练集数据有 7352 个数据, 数据包括如下:

- body\_acc: 从总的加速度中减去重力获得的加速度信号 (128 个值组成)

- body\_gyro: 陀螺仪获得角速度 (128 个值组成)

- total\_acc: 总的加速度 (128 个值组成)

每个数据有 128 个特征。测试集有 2947 个数据

卷积神经网络架构如下:

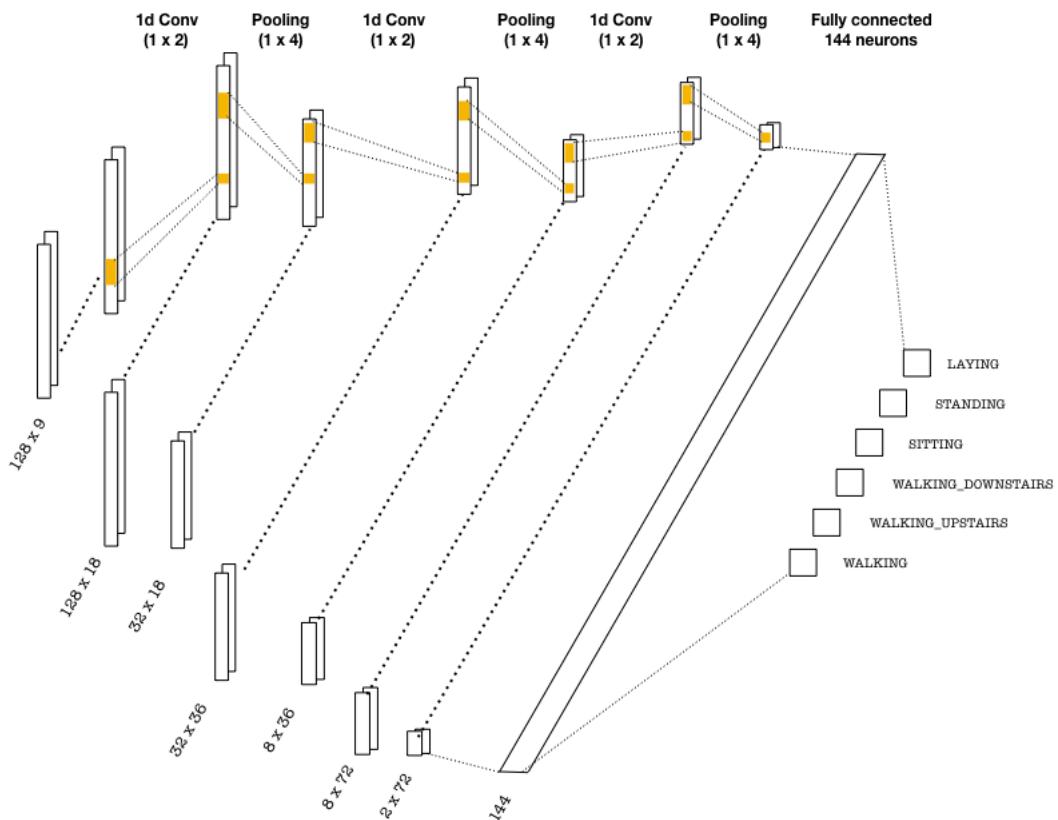


图 9.5: CNN 处理 HAR 数据的架构

数据预处理模块:

```

1 # HAR classification
2 # Author: Burak Himmetoglu
3 # 8/15/2017
4 # Burak Himmetoglu 'github
5 # https://github.com/healthDataScience/deep-learning-HAR
6 # Environment TensorFlow 1.3 Deepin 64bit
7 # 9/13/2017 modified by bleedingfight
8
9 import pandas as pd
10 import numpy as np
11 import os
12

```

```

13 def read_data(data_path, split = "train"):
14     """ Read data """
15
16     # Fixed params
17     n_class = 6
18     n_steps = 128
19
20     # Paths
21     path_ = os.path.join(data_path, split)
22     path_signals = os.path.join(path_, "Inertial Signals")
23
24     # Read labels and one-hot encode
25     label_path = os.path.join(path_, "y_" + split + ".txt")
26     labels = pd.read_csv(label_path, header = None)
27
28     # Read time-series data
29     channel_files = sorted(os.listdir(path_signals))
30     n_channels = len(channel_files)
31     posix = len(split) + 5
32
33     # Initiate array
34     list_of_channels = []
35     X = np.zeros((len(labels), n_steps, n_channels))
36     i_ch = 0
37
38     for fil_ch in channel_files:
39         channel_name = fil_ch[:-posix]
40         dat_ = pd.read_csv(os.path.join(path_signals, fil_ch),
41                            delim_whitespace = True, header = None)
42         X[:, :, i_ch] = dat_.as_matrix()
43
44     # Record names
45     list_of_channels.append(channel_name)
46
47     # iterate
48     i_ch += 1
49
50     # Return
51     return X, labels[0].values, list_of_channels
52
53 def standardize(train, test):
54     """ Standardize data """
55     all_data = np.concatenate((train, test), axis = 0)
56     assert np.allclose(all_data[:len(train)], train), "Wrong training set!"

```

```

56 assert np.allclose(all_data[len(train):], test), "Wrong test set!"
57
58 # Standardise each channel
59 all_data = (all_data - np.mean(all_data, axis=1)[:,None,:]) / np.std(all_data,
60             axis=1)[:,None,:]
61
62 # Split back and return
63 X_train = all_data[:len(train)]
64 X_test = all_data[len(train):]
65
66 return X_train, X_test
67
68 def one_hot(labels, n_class = 6):
69     """ One-hot encoding """
70     expansion = np.eye(n_class)
71     y = expansion[:, labels-1].T
72     assert y.shape[1] == n_class, "Wrong number of labels!"
73
74     return y
75
76 def get_batches(X, y, batch_size = 100):
77     """ Return a generator for batches """
78     n_batches = len(X) // batch_size
79     X, y = X[:n_batches*batch_size], y[:n_batches*batch_size]
80
81     # Loop over batches and yield
82     for b in range(0, len(X), batch_size):
83         yield X[b:b+batch_size], y[b:b+batch_size]

```

神经网络实现模块:

```

1 import numpy as np
2 import os
3 from utilities import *
4 from sklearn.model_selection import train_test_split
5 import matplotlib.pyplot as plt
6 import tensorflow as tf
7
8
9
10 data_path = '/home/liushuai/Pictures/UCI HAR Dataset'
11 X_train, labels_train, list_ch_train = read_data(data_path=data_path) # train
12 X_test, labels_test, list_ch_test = read_data(data_path=data_path, split="test")
13     # test
14 assert list_ch_train == list_ch_test, "Mismatch in channels!"

```

```

14 X_train, X_test = standardize(X_train, X_test)
15 X_tr, X_vld, lab_tr, lab_vld = train_test_split(X_train, labels_train,
16                                         stratify = labels_train, random_state = 123)
17 y_tr = one_hot(lab_tr)
18 y_vld = one_hot(lab_vld)
19 y_test = one_hot(labels_test)
20 batch_size = 600      # Batch size
21 seq_len = 128        # Number of steps
22 learning_rate = 0.001
23 epochs = 500
24
25 n_classes = 6
26 n_channels = 9
27
28 graph = tf.Graph()
29 with graph.as_default():
30     inputs_ = tf.placeholder(tf.float32,[None,seq_len,n_channels],name='input')
31     labels_ = tf.placeholder(tf.float32,[None,n_classes],name = "labels")
32     keep_prob_ = tf.placeholder(tf.float32,name = 'keep')
33     learning_rate_ = tf.placeholder(tf.float32,name='learning_rate')
34 with graph.as_default():
35     # (batch, 128, 9) —> (batch, 32, 18)
36     conv1 = tf.layers.conv1d(inputs=inputs_, filters=18,
37                             kernel_size=2, strides=1,
38                             padding='same', activation = tf.nn.relu)
39     max_pool_1 = tf.layers.max_pooling1d(inputs=conv1, pool_size=4,
40                                         strides=4, padding='same')
41
42     # (batch, 32, 18) —> (batch, 8, 36)
43     conv2 = tf.layers.conv1d(inputs=max_pool_1, filters=36,
44                             kernel_size=2, strides=1,
45                             padding='same', activation = tf.nn.relu)
46     max_pool_2 = tf.layers.max_pooling1d(inputs=conv2, pool_size=4,
47                                         strides=4, padding='same')
48
49     # (batch, 8, 36) —> (batch, 2, 72)
50     conv3 = tf.layers.conv1d(inputs=max_pool_2, filters=72, kernel_size=2,
51                             strides=1,padding='same', activation = tf.nn.relu)
52     max_pool_3 = tf.layers.max_pooling1d(inputs=conv3, pool_size=4,
53                                         strides=4, padding='same')
54
55 with graph.as_default():
56     #Flatten and dropout

```

```

57     flat = tf.reshape(max_pool_3,(-1,2*72))
58     flat = tf.nn.dropout(flat,keep_prob=keep_prob_)
59 # predictions
60     logits = tf.layers.dense(flat,n_classes)
61
62 #Cost function and optimizer
63     cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
64             logits=logits,labels=labels_))
65     optimizer = tf.train.AdamOptimizer(learning_rate_).minimize(cost)
66 # Accuracy
67     correct_pred = tf.equal(tf.argmax(logits,1),tf.argmax(labels_,1))
68     accuracy = tf.reduce_mean(tf.cast(correct_pred,tf.float32),name = 'accuracy')
69
70 if (os.path.exists('checkpoints-cnn') == False):
71     os.mkdir('checkpoints-cnn')
72 validation_acc = []
73 validation_loss = []
74
75 train_acc = []
76 train_loss = []
77
78 with graph.as_default():
79     saver = tf.train.Saver()
80
81 with tf.Session(graph=graph) as sess:
82     sess.run(tf.global_variables_initializer())
83     iteration = 1
84
85 # Loop over epochs
86 for e in range(epochs):
87
88     # Loop over batches
89     for x,y in get_batches(X_tr, y_tr, batch_size):
90
91         # Feed dictionary
92         feed = {inputs_ : x, labels_ : y, keep_prob_ : 0.5, learning_rate_ :
93             learning_rate}
94
95         # Loss
96         loss, __, acc = sess.run([cost, optimizer, accuracy], feed_dict =
feed)
train_acc.append(acc)

```

```

97     train_loss.append(loss)

98

99     # Print at each 5 iters
100    if (iteration % 5 == 0):
101        print("Epoch: {} / {}".format(e, epochs),
102              "Iteration: {:d}".format(iteration),
103              "Train loss: {:.6f}".format(loss),
104              "Train acc: {:.6f}".format(acc))

105

106    # Compute validation loss at every 10 iterations
107    if (iteration%10 == 0):
108        val_acc_ = []
109        val_loss_ = []

110

111    for x_v, y_v in get_batches(X_vld, y_vld, batch_size):
112        # Feed
113        feed = {inputs_ : x_v, labels_ : y_v, keep_prob_ : 1.0}

114

115        # Loss
116        loss_v, acc_v = sess.run([cost, accuracy], feed_dict = feed)
117        val_acc_.append(acc_v)
118        val_loss_.append(loss_v)

119

120        # Print info
121        print("Epoch: {} / {}".format(e, epochs),
122              "Iteration: {:d}".format(iteration),
123              "Validation loss: {:.6f}".format(np.mean(val_loss_)),
124              "Validation acc: {:.6f}".format(np.mean(val_acc_)))

125

126        # Store
127        validation_acc.append(np.mean(val_acc_))
128        validation_loss.append(np.mean(val_loss_))

129

130        # Iterate
131        iteration += 1

132

133        saver.save(sess, "checkpoints-cnn/har.ckpt")
134    # Plot training and test loss
135    t = np.arange(iteration - 1)
136

137    plt.figure(figsize = (6,6))
138    plt.plot(t, np.array(train_loss), 'r-', t[t % 10 == 0], np.array(validation_loss),
              'b*')

```

```

139 plt.xlabel("iteration")
140 plt.ylabel("Loss")
141 plt.legend(['train', 'validation'], loc='upper right')
142 plt.savefig('trained_result.png', dpi=800)

```

## 9.1.4 LSTM

## 处 理 序 列 数 据

LSTM 架构如下：

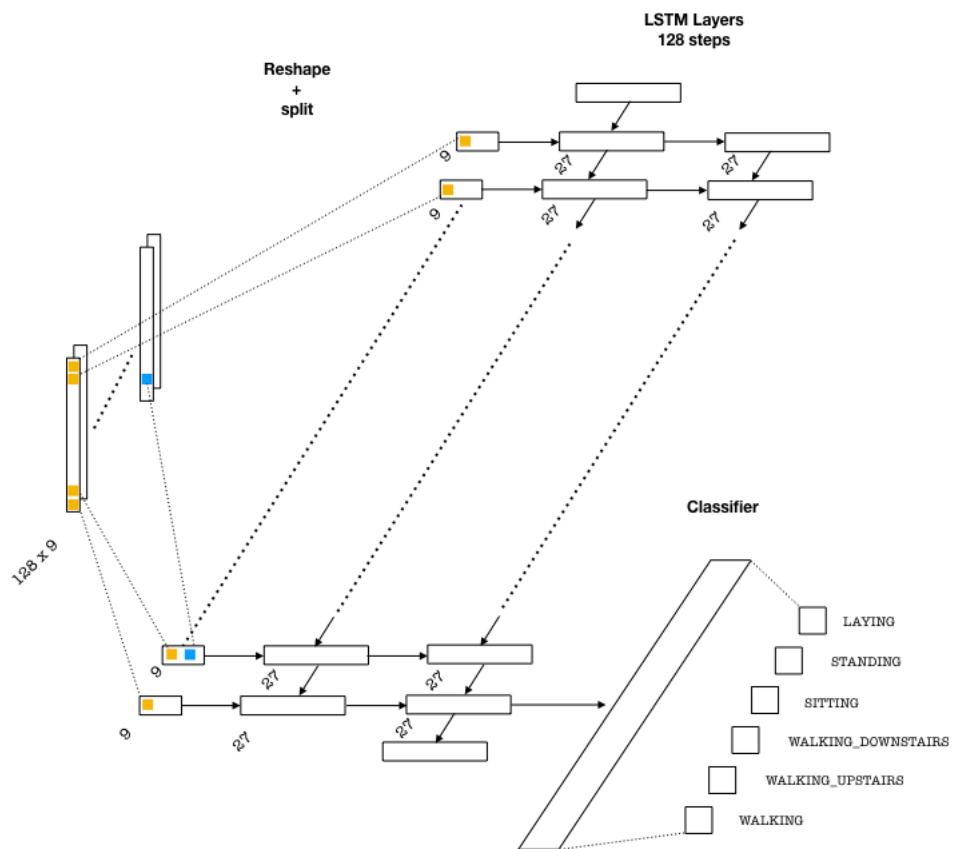


图 9.6: lstm 架构图

```

1 #Deepin 64bit Python3.6.1 TensorFlow1.3 CPU测试通过
2 import numpy as np
3 import tensorflow as tf
4 import os

```

```

5 from utilities import *
6 from sklearn.model_selection import train_test_split
7 import matplotlib.pyplot as plt
8 data_path = '/home/liushuai/Pictures/UCI HAR Dataset'
9 X_train, labels_train, list_ch_train = read_data(data_path=data_path, split="train")
10 X_test, labels_test, list_ch_test = read_data(data_path=data_path, split="test")
11 assert list_ch_train == list_ch_test, "Mismatch in channels!"
12 X_train, X_test = standardize(X_train, X_test)
13 X_tr, X_vld, lab_tr, lab_vld = train_test_split(X_train, labels_train,
14                                                 stratify=labels_train, test_size=0.2,
15                                                 random_state=123)
16
17 y_tr = one_hot(lab_tr)
18 y_vld = one_hot(lab_vld)
19 y_test = one_hot(labels_test)
20
21 lstm_size = 27           # 2 times the amount of channels
22 lstm_layers = 2          # Number of layers
23 batch_size = 600          # Batch size
24 seq_len = 128             # Number of steps
25 learning_rate = 0.0005   # Learning rate (default is 0.001)
26 epochs = 500
27
28 # Fixed
29 n_classes = 6
30 n_channels = 9
31
32 graph = tf.Graph()
33 with graph.as_default():
34     inputs_ = tf.placeholder(tf.float32, [None, seq_len, n_channels], name='inputs')
35     labels_ = tf.placeholder(tf.float32, [None, n_classes], name='labels')
36     keep_prob_ = tf.placeholder(tf.float32, name='keep')
37     learning_rate_ = tf.placeholder(tf.float32, name='learning_rate')
38
39 with graph.as_default():
40     # Construct the LSTM inputs and LSTM cells
41     lstm_in = tf.transpose(inputs_, [1, 0, 2]) # reshape into (seq_len, N,
42                                         channels)
43     lstm_in = tf.reshape(lstm_in, [-1, n_channels]) # Now (seq_len*N, n_channels)

```

```

44 # To cells
45 lstm_in = tf.layers.dense(lstm_in, lstm_size, activation=None) # or tf.nn.
46   relu, tf.nn.sigmoid, tf.nn.tanh?
47
48 # Open up the tensor into a list of seq_len pieces
49 lstm_in = tf.split(lstm_in, seq_len, 0)
50
51 # Add LSTM layers
52 lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
53 drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob_)
54 cell = tf.contrib.rnn.MultiRNNCell([drop] * lstm_layers)
55 initial_state = cell.zero_state(batch_size, tf.float32)
56 with graph.as_default():
57   outputs, final_state = tf.contrib.rnn.static_rnn(cell, lstm_in, dtype=tf.
58     float32,
59       initial_state=initial_state)
60   logits = tf.layers.dense(outputs[-1], n_classes, name='logits')
61   cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
62     labels=labels_))
63   train_op = tf.train.AdamOptimizer(learning_rate_)
64   gradients = train_op.compute_gradients(cost)
65   capped_gradients = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var in
66     gradients]
67   optimizer = train_op.apply_gradients(capped_gradients)
68   correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(labels_, 1))
69   accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32), name='accuracy')
70
71 if (os.path.exists('checkpoints-rnn') == False):
72   os.mkdir('checkpoints-rnn')
73 validation_acc = []
74 validation_loss = []
75 train_acc = []
76 train_loss = []
77 with graph.as_default():
78   saver = tf.train.Saver()
79 with tf.Session(graph=graph) as sess:
80   sess.run(tf.global_variables_initializer())
81   iteration = 1
82   for e in range(epochs):
83     state = sess.run(initial_state)
84     for x,y in get_batches(X_tr,y_tr,batch_size):
85       feed = {inputs_:x,labels_:y,keep_prob_:0.5,
86           initial_state:state,learning_rate_:learning_rate}

```

```

82     loss ,_,state ,acc = sess.run([ cost ,optimizer ,final_state ,accuracy] ,
83     feed_dict = feed)
84     train_acc.append(acc)
85     train_loss.append(loss)
86     if (iteration%10 ==0):
87         print("Epoch:{}/{})".format(e,epochs) ,
88             "Iteration:{:d})".format(iteration) ,
89             "Train loss:{:.6f})".format(loss) ,
90             "Train acc:{:.6f})".format(acc))
91     if (iteration%25 == 0):
92         val_state = sess.run( cell.zeros_state(batch_size ,tf.float32))
93         val_acc_ = []
94         val_loss_ = []
95         for x_v,y_v in get_batches(X_vld,y_vld,batch_size):
96             feed = {inputs_:x_v,labels_:y_v,keep_prob_:1.0,initial_state
97 :val_state}
98             loss_v ,state_val.acc_v = sess.run([ cost ,finnal_state ,
99 accuracy] ,feed_dict=feed)
100             val_acc_.append(acc_v)
101             val_loss_.append(loss_v)
102             print("Epoch:{}/{})".format(e,epochs) ,
103                 "Iteration:{:d})".format(iteration) ,
104                 "Validation loss:{:.6f})".format(np.mean(val_loss_)) ,
105                 "Validation acc:{:.6f})".format(np.mean(val_acc_)))
106             validation_acc.append(np.mean(val_acc_))
107             validation_loss.append(np.mean(val_loss_))
108             iteration += 1
109             saver.save(sess ,”checkpoints/eeg.ckpt”)
110
111
112 with tf.Session(graph=graph) as sess:
113     sess.run(tf.global_variables_initializer())
114     iteration = 1
115
116     for e in range(epochs):
117         # Initialize
118         state = sess.run(initial_state)
119
120         # Loop over batches
121         for x,y in get_batches(X_tr, y_tr, batch_size):
122
123             # Feed dictionary
124             feed = {inputs_ : x, labels_ : y, keep_prob_ : 0.5,
125                     initial_state : state, learning_rate_ : learning_rate}

```

```

122
123     loss , __ , state , acc = sess.run([ cost , optimizer , final_state ,
accuracy ] ,
124                                         feed_dict = feed)
125     train_acc.append(acc)
126     train_loss.append(loss)
127
128     # Print at each 5 iters
129     if (iteration % 50 == 0):
130         print("Epoch: {} / {} ".format(e, epochs),
131               "Iteration: {:d} ".format(iteration),
132               "Train loss: {:.6f} ".format(loss),
133               "Train acc: {:.6f} ".format(acc))
134
135     # Compute validation loss at every 25 iterations
136     if (iteration%100 == 0):
137
138         # Initiate for validation set
139         val_state = sess.run( cell.zero_state(batch_size , tf.float32) )
140
141         val_acc_ = []
142         val_loss_ = []
143         for x_v, y_v in get_batches(X_vld, y_vld, batch_size):
144             # Feed
145             feed = {inputs_ : x_v, labels_ : y_v, keep_prob_ : 1.0 ,
initial_state : val_state}
146
147             # Loss
148             loss_v, state_v, acc_v = sess.run([ cost , final_state ,
accuracy ] , feed_dict = feed)
149
150             val_acc_.append(acc_v)
151             val_loss_.append(loss_v)
152
153         # Print info
154         print("Epoch: {} / {} ".format(e, epochs),
155               "Iteration: {:d} ".format(iteration),
156               "Validation loss: {:.6f} ".format(np.mean(val_loss_)),
157               "Validation acc: {:.6f} ".format(np.mean(val_acc_)))
158
159         # Store
160         validation_acc.append(np.mean(val_acc_))
161         validation_loss.append(np.mean(val_loss_))

```

```

162
163     # Iterate
164     iteration += 1
165
166     saver.save(sess, "checkpoints/eeg.ckpt")
167 plt.plot(validation_acc, validation_loss, 'r')
168 plt.title('accuracy with loss')
169 plt.savefig('acc_loss.png', dpi=800)

```

## CNN 和 LSTM 结合

```

1 import numpy as np
2 import os
3 from utilities import *
4 from sklearn.model_selection import train_test_split
5 import tensorflow as tf
6 import matplotlib.pyplot as plt
7 data_path = './UCI HAR Dataset'
8 X_train, labels_train, list_ch_train = read_data(data_path=data_path, split='train')
9 X_test, labels_test, list_ch_test = read_data(data_path=data_path, split='test')
10 assert list_ch_train == list_ch_test, "Mismatch in channels!"
11 X_tr, X_vld, lab_tr, lab_vld = train_test_split(X_train, labels_train, test_size =
12     0.2,
13     stratify = labels_train, random_state = 123)
14 y_tr = one_hot(lab_tr)
15 y_vld = one_hot(lab_vld)
16 y_test = one_hot(labels_test)
17 lstm_size = 36
18 lstm_layers = 2
19 batch_size = 600
20 seq_len = 128
21 learning_rate = 0.0005
22 n_classes = 6
23 n_channels = 9
24 epochs = 500
25
26 graph = tf.Graph()
27 with graph.as_default():
28     inputs_ = tf.placeholder(tf.float32, [None, seq_len, n_channels], name='inputs')
29     labels_ = tf.placeholder(tf.float32, [None, n_classes], name = 'labels')
30     keep_prob_ = tf.placeholder(tf.float32, name = 'keep')
31     learning_rate_ = tf.placeholder(tf.float32, name = 'learning_rate')
32 with graph.as_default():
33     conv1 = tf.layers.conv1d(inputs = inputs_, filters=18,kernel_size=2,strides

```

```

=1,
33         padding='same', activation = tf.nn.relu)
34 n_ch = n_channels*2
35 with graph.as_default():
36     lstm_in = tf.transpose(conv1,[1,0,2])
37     lstm_in = tf.reshape(lstm_in,[-1,n_ch])
38     lstm_in = tf.layers.dense(lstm_in,lstm_size,activation = tf.nn.relu)
39     lstm_in = tf.split(lstm_in,seq_len,0)
40     lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
41     drop = tf.contrib.rnn.DropoutWrapper(lstm,output_keep_prob = keep_prob_)
42     cell = tf.contrib.rnn.MultiRNNCell([drop]*lstm_layers)
43     initial_state = cell.zero_state(batch_size,tf.float32)
44
45 with graph.as_default():
46     outputs,final_state = tf.contrib.rnn.static_rnn(cell,lstm_in,dtype = tf.
47     float32,initial_state = initial_state)
48     logits = tf.layers.dense(outputs[-1],n_classes,name='logits')
49     cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
50     labels=labels_))
51     train_op = tf.train.AdamOptimizer(learning_rate_)
52     gradients = train_op.compute_gradients(cost)
53     capped_gradients = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var in
54     gradients]
55     optimizer = train_op.apply_gradients(capped_gradients)
56     correct_pred = tf.equal(tf.argmax(logits,1),tf.argmax(labels_,1))
57     accuracy = tf.reduce_mean(tf.cast(correct_pred,tf.float32),name = 'accuracy')
58
59 if (os.path.exists('checkpoints-crnn') == False):
60     os.mkdir('checkpoints-crnn')
61 validation_acc = []
62 validation_loss = []
63 train_acc = []
64 train_loss = []
65 with graph.as_default():
66     saver = tf.train.Saver()
67 #with tf.Session(graph=graph) as sess:
68 #    sess.run(tf.global_variables_initializer())
69 #    iteration = 1
70 #    for e in range(epochs):
71 #        state = sess.run(initial_state)
72 #        for x,y in get_batches(X_train,y_tr,batch_size):
73 #            feed = {inputs_:x,labels_:y,keep_prob_:0.5,

```

```

71 #           initial_state:state , learning_rate_:learning_rate}
72 #           loss ,_, state , acc = sess.run([ cost , optimizer , final_state ,
73 #           accuracy] , feed_dict = feed)
74 #           train_acc.append(acc)
75 #           train_loss.append(loss)
76 #           if (iteration%5 == 0):
77 #               print ('Epoch:{} / {}'.format(e,epochs) ,
78 #                     'Iteration:{} :d'.format(iteration) ,
79 #                     'Train loss:{} :6 f'.format(loss) ,
80 #                     'Train acc:{} :.6 f'.format(acc))
81 #           if (iteration%25 == 0):
82 #               val_state = sess.run(cell.zeros_state(batch_size,tf.float32))
83 #               val_acc_ = []
84 #               val_loss_ = []
85 #               for x_v,y_v in get_batches(X_vld,y_vld,batch_size):
86 #                   feed = {input_:x_v,labels_:y_v,keep_prb_:1.,
87 #                           initial_state:val_state}
88 #                   val_acc_.append(acc_v)
89 #                   val_loss_.append(loss_v)
90 #                   print ('Epoch:{} / {}'.format(e,epochs) ,
91 #                         "Iteration:{} :d".format(iteration) ,
92 #                         'Validation loss:{} :6 f'.format(np.mean(val_loss_)) ,
93 #                         'Validation acc:{} :.6 f'.format(np.mean(val_acc_)))
94 #                   validation_acc_.append(np.mean(val_acc_))
95 #                   validation_loss_.append(np.mean(val_loss_))
96 #               iteration+=1
97 #
98 #           saver.save(sess,"checkpoints-crnn/har.ckpt")
99 with tf.Session(graph=graph) as sess:
100     sess.run(tf.global_variables_initializer())
101     iteration = 1
102     for e in range(epochs):
103         # Initialize
104         state = sess.run(initial_state)
105         # Loop over batches
106         for x,y in get_batches(X_tr, y_tr, batch_size):
107             # Feed dictionary
108             feed = {inputs_ : x, labels_ : y, keep_prob_ : 0.5,
109                     initial_state : state , learning_rate_ : learning_rate}
110             loss , _, state , acc = sess.run([ cost , optimizer , final_state ,
111             accuracy] , feed_dict = feed)
112             train_acc.append(acc)
113             train_loss.append(loss)

```

```

113     # Print at each 5 iters
114     if (iteration % 5 == 0):
115         print("Epoch: {} / {}".format(e, epochs),
116               "Iteration: {}".format(iteration),
117               "Train loss: {:.6f}".format(loss),
118               "Train acc: {:.6f}".format(acc))
119     # Compute validation loss at every 25 iterations
120     if (iteration%25 == 0):
121         # Initiate for validation set
122         val_state = sess.run(cell.zero_state(batch_size, tf.float32))
123         val_acc_ = []
124         val_loss_ = []
125         for x_v, y_v in get_batches(X_vld, y_vld, batch_size):
126             # Feed
127             feed = {inputs_ : x_v, labels_ : y_v, keep_prob_ : 1.0,
128                     initial_state : val_state}
129             # Loss
130             loss_v, state_v, acc_v = sess.run([cost, final_state,
131                                              accuracy], feed_dict = feed)
132             val_acc_.append(acc_v)
133             val_loss_.append(loss_v)
134             # Print info
135             print("Epoch: {} / {}".format(e, epochs),
136                   "Iteration: {}".format(iteration),
137                   "Validation loss: {:.6f}".format(np.mean(val_loss_)),
138                   "Validation acc: {:.6f}".format(np.mean(val_acc_)))
139             # Store
140             validation_acc.append(np.mean(val_acc_))
141             validation_loss.append(np.mean(val_loss_))
142             iteration += 1
143             saver.save(sess, "checkpoints-crnn/har.ckpt")
144         with tf.Session(graph=graph) as sess:
145             sess.run(tf.global_variables_initializer())
146             iteration = 1
147             for e in range(epochs):
148                 # Initialize
149                 state = sess.run(initial_state)
150                 # Loop over batches
151                 for x,y in get_batches(X_tr, y_tr, batch_size):
152                     # Feed dictionary
153                     feed = {inputs_ : x, labels_ : y, keep_prob_ : 0.5,
154                             initial_state : state, learning_rate_ : learning_rate}
155                     loss, _, state, acc = sess.run([cost, optimizer, final_state,
156

```

```

accuracy] , feed_dict = feed)
    train_acc.append(acc)
    train_loss.append(loss)
    # Print at each 5 iters
    if (iteration % 5 == 0):
        print("Epoch: {} / {}" .format(e, epochs),
              "Iteration: {:d}" .format(iteration),
              "Train loss: {:.6f}" .format(loss),
              "Train acc: {:.6f}" .format(acc))
    # Compute validation loss at every 25 iterations
    if (iteration%25 == 0):
        # Initiate for validation set
        val_state = sess.run(cell.zero_state(batch_size, tf.float32))
        val_acc_ = []
        val_loss_ = []
        for x_v, y_v in get_batches(X_vld, y_vld, batch_size):
            # Feed
            feed = {inputs_ : x_v, labels_ : y_v, keep_prob_ : 1.0,
initial_state : val_state}
            # Loss
            loss_v, state_v, acc_v = sess.run([cost, final_state,
accuracy], feed_dict = feed)
            val_acc_.append(acc_v)
            val_loss_.append(loss_v)
        # Print info
        print("Epoch: {} / {}" .format(e, epochs),
              "Iteration: {:d}" .format(iteration),
              "Validation loss: {:.6f}" .format(np.mean(val_loss_)),
              "Validation acc: {:.6f}" .format(np.mean(val_acc_)))
        # Store
        validation_acc.append(np.mean(val_acc_))
        validation_loss.append(np.mean(val_loss_))
    # Iterate
    iteration += 1
saver.save(sess, "checkpoints-crnn/har.ckpt")

```



# Chapter 10

## Tensorflow 技巧

### 10.1 文 件 读 取

#### 10.1.1 批 量 读 取 压 缩 图 片 为 指 定 格 式

```
1 # TensorFlow1.3 GPU ubuntu14.0.4 64bit
2 # 文件和处理的图像放在一个目录
3 import tensorflow as tf
4 import cv2
5 from os import listdir
6 # 将图片压缩， 默认压缩大小为[224,224]
7 def parse_function(filename,label):
8     image_string = tf.read_file(filename)
9     image_decoded = tf.image.decode_png(image_string,3)
10    image_resized = tf.image.resize_images(image_decoded,[224,224])
11    return image_resized,label
12 # 过滤filepath路径中非ext格式的文件， 默认过滤保留png格式图片
13 def filter_file(filepath,ext='png'):
14     fileall = listdir(filepath)
15     filtered_list = []
16     for i in fileall:
17         if (len(i.split('.'))==2) and (i.split('.')[1]==ext):
18             filtered_list.append(i)
19     return filtered_list
20 # 压缩保存
21 def clip_pic(filenames,save_path):
22     num_files = len(filenames)
23     labels = tf.constant([0]*num_files)
24     dataset = tf.contrib.data.Dataset.from_tensor_slices((tf.constant(filenames),labels))
```

```
25     dataset = dataset.map(parse_function)
26     iterator = dataset.make_one_shot_iterator()
27     next_element = iterator.get_next()
28     sess = tf.Session()
29     count = 0
30     for i in range(num_files):
31         img = sess.run(next_element)[0]
32         cv2.imwrite(save_path+filenames[i],img)
33         count+=1
34     print('成功压缩'+str(count)+'张图片')
35 def main():
36     filepath = '/home/bleedingfight/Pylon/test/testma/'
37     save_path = '/home/bleedingfight/test1/'
38     filenames = filter_file(filepath)
39     clip_pic(filenames,save_path)
40
41 if __name__ == '__main__':
42     main()
```

# Chapter 11

## Tensorflow API

### 11.1 tf.check\_numerics

check\_numerics(tensor,message,none): 检查一个 tensor 是 NaN 或者 Inf 值。运行的时候如果 tensor 有任何值不是 NaN 或者 Inf 报告一个 InvalidArgument 错误，否则传递 tensor 的值。

- tensor: 一个 Tensor，可以使下面的类型:half,float32,float64
- message: 一个 string，错误的前缀
- name: 操作的名字

Return : 返回一个和 tensor 同样类型的 Tensor

### 11.2 tf.clip\_by\_value

clip\_by\_value(t,clip\_value\_min,clip\_value\_max,name=None): 剪切值为最大值和最小值，给定一个 tensor t，操作返回一个和 t 同样类型，同样大小的值到 clip\_value\_min, 和 clip\_value\_max. 任何值小于 clip\_value\_min 被设置为 clip\_value\_min, 任何大于 clip\_value\_max 的值被设置为 clip\_value\_max。参数:

- t: 一个 tensor
- clip\_value\_min: 一个 0 维 Tensor 或者和 t 相同形状的 tensor，取代的最小值
- clip\_value\_max: 一个 0 维 Tensor 或者和 t 有相同形状的 Tensor，取代的最大值
- name: 操作的名字

Returns : 一个取代后的 tensor。

Raises : 如果剪切的 tensor 将触发数组的广播操作将安徽比输入更大的返回 tensor。

## 11.3 tf.app.flags

### 11.3.1 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。

### 11.3.2 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。

### 11.3.3 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。

### 11.3.4 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。

### 11.3.5 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。

### 11.3.6 tf.convert\_to\_tensor

```

1 convert_to_tensor(
2     value,
3     dtype=None,
4     name=None,
5     preferred_dtype=None
6 )

```

转换给定的 value 为一个 Tensor。这个函数转转 Python 对象为 Tensor 对象，接受的 Tensor 对象有 numpy array,Python list,Python 标量。例如

```

1 import numpy as np
2
3 def my_func(arg):
4     arg = tf.convert_to_tensor(arg, dtype=tf.float32)
5     return tf.matmul(arg, arg) + arg
6
7 # The following calls are equivalent.
8 value_1 = my_func(tf.constant([[1.0, 2.0], [3.0, 4.0]]))
9 value_2 = my_func([[1.0, 2.0], [3.0, 4.0]])
10 value_3 = my_func(np.array([[1.0, 2.0], [3.0, 4.0]]), dtype=np.float32))

```

这个函数在组合新的操作时很有用，如上慢的 my\_func。所有的标注的 Python 操作构造体应用这个函数到每个输入的 Tensor 值，允许这些操作接受 numpy 数组，Python 列表，标量和 Tensor 对象。

- value: 支持转化为 Tensor 的数据类型
- dtype: 返回 tensor 的元素类型。如果确实，类型从 value 推断
- name: 如果新的 Tensor 被创建，这个参数决定 Tensor 的名字
- preferred\_dtype: 返回 tensor 的元素类型选项，当 dtype 为 None 时。在一些情况下，调用器也许没有类型转化为一个 tensor，因此 preferred\_dtype 可能用作软偏好。如果转化为 preferred\_dtype 不可能，这个参数将不起作用

Returns 一个基于 value 的输出

Raises :TypeError: 如果没有转换对象适用与 value。RuntimeError: 如果转化对象的值不可用

### 11.3.7 tf.gather

```

1 gather(
2     params,
3     indices,
4     validate_indices=None,
5     name=None,
6     axis=0
7 )

```

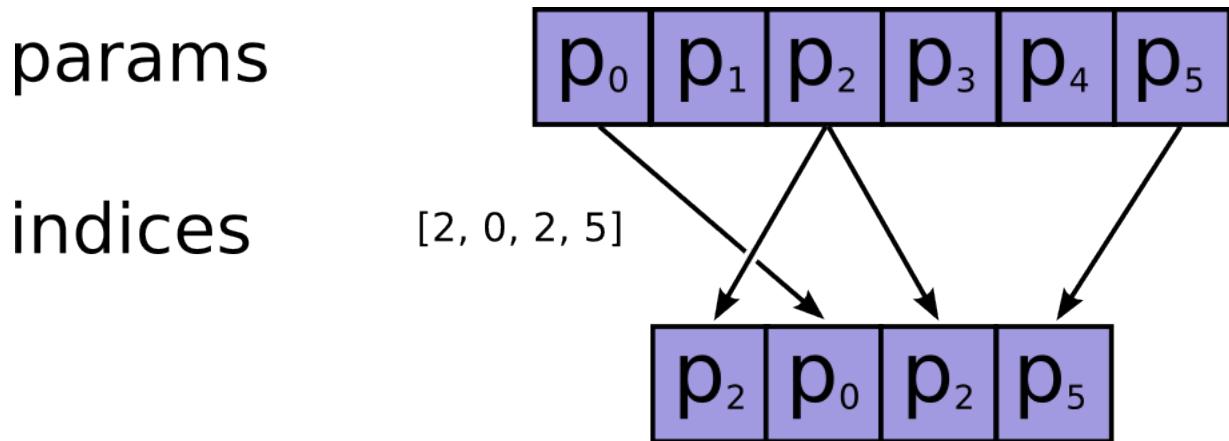
根据 indices 从 params 和 axis 轴获取 slices。indices 必须是任何维度 (通常是 0 维或者一维) 的一个整数 tensor 生成输出 tensor 的形状

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]

```



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]]

```

### 11.3.8 tf.one\_hot

```

1 one_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 位置的值由 on\_value 表示, 其它位置的值由 off\_value 表示。on\_value 和 off\_value 必须匹配数据类型, 如果 dtype 被提供他们必须和 dtype 制定的类型相同。

如果 on\_value 没有没提供, 他讲默认为 dtype 指定的 0。off\_value 没有提供, 他讲默认设置为 dtype 类型的 0。

如果设计日 indices rank 为 N, 输出 rank 将为 N+1, 新的轴将在 axis 创建 (默认新的轴被添加到末尾)

如果 indices 是一个标量, 输出形状将是一个长度为 depth 的向量

如果 indices 是一个长度为 features 的向量, 输出形状将为:

```
1 features x depth if 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 没有提供, 将尝试假设 on\_value 和 off\_value 的类型, 如果一个或者两个被传递进去。如果 on\_value, off\_value 是空或者 dtype 被提供, dtype 将默认为 tf.float32。

如果一个非数值类型输出被妖气, on\_value 和 off\_value 必须提供给 one\_hot。

例如:

```
1 indices = [0, 1, 2]
2 depth = 3
3 tf.one_hot(indices, depth) # output: [3 x 3]
4 # [[1., 0., 0.],
5 #  [0., 1., 0.],
6 #  [0., 0., 1.]]
7
8 indices = [0, 2, -1, 1]
9 depth = 3
10 tf.one_hot(indices, depth,
11             on_value=5.0, off_value=0.0,
12             axis=-1) # output: [4 x 3]
13 # [[5.0, 0.0, 0.0], # one_hot(0)
14 #  [0.0, 0.0, 5.0], # one_hot(2)
15 #  [0.0, 0.0, 0.0], # one_hot(-1)
16 #  [0.0, 5.0, 0.0]] # one_hot(1)
17
18 indices = [[0, 2], [1, -1]]
19 depth = 3
20 tf.one_hot(indices, depth,
21             on_value=1.0, off_value=0.0,
22             axis=-1) # output: [2 x 2 x 3]
23 # [[[1.0, 0.0, 0.0], # one_hot(0)
24 #   [0.0, 0.0, 1.0]], # one_hot(2)
25 #   [[0.0, 1.0, 0.0], # one_hot(1)
```

```
26 # [0.0, 0.0, 0.0]] # one_hot(-1)
```

参数:

- indices: 一个索引 Tensor
- depth: 一个定义 onehot 维度的标量
- on\_value: 一个定义填充在输出的标量 indices[j]=i(默认为 1)
- off\_value: 一个定义填充输出的标量 indices[j]!i(默认为 0)
- axis: 一个填充的轴, 默认为-1
- dtype: 输出 tensor 的数据类型

Returns output:one-hottensor

Raises :TypeError: 如果 on\_value 或者 off\_value 和 dtype 不匹配, 或者 on\_value 和 off\_value 不匹配

### 11.3.9 tf.placeholder

```
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。它的值必须通过 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。

### 11.3.10 tf.py\_func

`py_func(func, inp, Tout, stateful=True, name=None)` 打包一个 python 函数将它用作一个 TensorFlow 操作，给一个 Python 函数‘func’，接受 numpy 数组作为它的输入返回 numpy 数组作为输出，转化函数作为一个 TensorFlow 图上的操作。下面的代码段构造一个简单的 TensorFlow 图利用 ‘`np.sinh()`’ Numpy 函数作为一个图上的操作：

```

1 def my_func(x):
2     # x will be a numpy array with the contents of the placeholder below
3     return np.sinh(x)
4
5     inp = tf.placeholder(tf.float32)
6     y = tf.py_func(my_func, [inp], tf.float32)

```

`tf.py_func()` 操作有如下已知的限制：

- 在 GraphDef 中函数将不被序列化，因此，如果你需要序列化你的模型和从不同的模型中恢复它，你不能用这个函数
- 这个操作必须运行在和 Python 程序相同的地址空间，如果你用分布式的 TensorFlow，你必须在痛痛的进程运行 `tf.train.Saver` 作为程序调用。
- `tf.py_func()` 和你必须添加创建的操作到服务器的设备上

参数：

- 一个 Python 函数接受一个和 ‘`inp`’ 对应的 `tf.Tensor` 对象元素类型匹配的 NumPy ndarray 对象列表，返回一个 ‘`adarray`’ 对象（或者一个 `ndarray`），它和 ‘`Tout`’ 中对应的值有相同的类型。
- `inp`: 一个 ‘`Tensor`’ 对象列表
- `Tout`: 一个 tensorflow 数据类型的列表或者元祖，或者如果仅有一个值的话为一个单个的 tensorflow 数据
- `statefull`: bool 行，如果为 True，函数并盖被认为 stateful 的，如果是 stateless 的，当给定相同的输入将返回相同的输出没有可见的副作用。优化像常见的子表达式减少仅仅在 stateless 上操作
- `name`: 操作的名字

返回 一个 Tensor 列表或者一个 ‘`func`’ 计算的 Tensor。

### 11.3.11 tf.read\_file

读取输出输入文件 `filename` 的整个内容 参数：

- filename: 一个 ‘string‘类型的 ‘Tensor‘
- name: 操作的名字

返回值 一个 ‘string‘类型的 ‘Tensor‘

#### 11.3.12 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]
```

#### 11.3.13 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 collectiobn 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。

#### 11.3.14 tf.split

split(value,num\_or\_size\_splits,axis=0,num=None,name='split'): 分割一个 Tensor, 如果 num\_or\_size\_split 是一个整数类型, num\_split 然后沿着 axis 分割 value 为 num\_split 更小的 tensors。要求 num\_split 均匀分割为 value.shape[axis]。如果 num\_or\_size\_splits 不是一个整数类型, 它被将定位一个 Tensor size\_splits, 然后分割值为 len(size\_splits) 段, 第 i 个形状有相同的大小作为 value 除了沿着维度 axis, 这里的 size 是 size\_splits[i]。

```

1 #value是一个形状为[5,30]的tensor
2 #分割沿着1轴分割'value'为3个tensor, 形状为【4,15,11】
3 split0 ,split1 ,split2 = tf.split (value ,[4,15,11] ,1)
4 tf.shape (split0) ==>[5,4]
5 tf.shape (split1) ==>[5,15]
6 tf.shape (split2) ==>[5,11]
7 #沿着1轴分割‘value’为三个tensor
8 split0 ,split1 ,split2 = tf.split (value ,num_or_size_split=3, axis=1)
9 tf.shape (split0) ==>[5,10]
```

参数:

- value: 要分割的 Tensor
- num\_or\_size\_splits: 是一个 0 维整数 Tensor, 表示沿着 split\_dim 或者包含每个分割长度的 1 维整数 tensor 分割多少份。如果是标量将被均匀分割为 value.shape[axis]。否者沿着分割维度求和必须和 value 匹配。
- axis: 0 维 int32 tensor, 沿着分割的维度, 必须在 [-rank(value),rank(value)], 默认为 0。
- num: 选项, 不能推断出时用于指定 size\_split 输出数
- name: 操作的名字

Returns : 如果 num\_or\_size\_split 是一个标量返回 num\_or\_size\_split 是一个一维 Tensor  
返回 value 分割的 num\_or\_size\_splits.get\_shape[0] Tensor 对象

Raises : 如果 num 不被指定也不能被推断出。将产生 ValueError

### 11.3.15 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)
```

### 11.3.16 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)

```

### 11.3.17 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。

## 11.3.18 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

```

## 11.3.19 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。

## 11.3.20 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()`

### 11.3.21 tf.random\_poisson

```

1 random_poisson(
2     lam,
3     shape,
4     dtype=tf.float32,
5     seed=None,
6     name=None
7 )

```

从 Poisson 分布中获取制定形状的样本，`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
   [:, 0] and [:, 1] 代表不同分布的数据
2
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 $7\times5$从两个分布的采样输出。
```

### 11.3.22 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。

### 11.3.23 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。

### 11.3.24 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

### 11.3.25 tf.transpose

tf.transpose(a,perm=None,name='transpose'): 根据 perm 改变 a 的维度，返回一个 tensor，它的维度 i 和 perm[i] 对应，如果 perm 没有给定，它被设置为 (n-1,...,0),z 这是 n 是输入 tensor 的 rank，输入是 2 维 tensor 是执行常规的转置操作:

```

1 # 'x' is [[1 2 3],[4 5 6]]
2 tf.transpose(x)==>[[1 4]
3 [2 5]
4 [3 6]]
5 #等价于
6 tf.transpose(x,[1,0])==>[[1 4]
7 [2 5]
8 [3 6]]
9 # perm对于n为tensor很有用， n>2
10 # x  [[1,2,3]
11 #      [4,5,6]]
12 #      [[7 8 9]
13 #      [10,11,12]]]
14 tf.transpose(x,perm=[0,2,1])==>[[[1 4]
15 [2 5]
16 [3,6]]
17
18      [[7 10]
19      [8 11]
20      [9 12]]]

```

参数:

- a: 一个 Tensor
- perm: 一个 a 的维度的乱序
- name: 操作的名字

Return 一个转置后的 Tensor

### 11.3.26 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，默認為 dtype 下的 1。off\_value 不提供，默認值為 0，如果輸入 Incices rank 是 N，輸出 rank 將是 N+1。新的 axis 將在 axis 建立。如果 indices 是一個標量，輸出形狀將是一個 depth 長度的向量，如果 indices 是億個長度為 features 向量輸出形狀將為：

```
1 feature x depth ifn axis == -1
2 depth x features if axis == 0
```

如果 indices 是一個矩陣，形狀為 [batch,features]，輸出形狀將為：

```
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 沒有被提供，它將假設 on\_value 或 off\_value 的數據類型，如果一個或多個的值被傳遞，，如果沒有 on\_value,off\_value，或者 dtype 被提供，dtype 將默認 tf.float32。

例子：

假設：

```
1 indices = [0, 2, -1, 1]
2 depth = 3
3 on_value = 5.0
4 off_value = 0.0
5 axis = -1
```

輸出形狀為  $[4 \times 3]$  輸出：

```
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)
```

假設：

```
1 indices = [[0, 2], [1, -1]]
2 depth = 3
3 on_value = 1.0
4 off_value = 0.0
5 axis = -1
```

輸出

```
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。

### 11.3.27 tf.tile

```

1 tile(
2   input ,
3   multiples ,
4   name=None
5 )

```

用给定的 tensor 铺出一个 tensor, 通过复制 input tensor multiples 次。输出 tensor 的第 i 维有 input.dims(i)\*multiples[i] 个元素, 输入值被沿着 i 维复制 multiples[i] 次。例如 [a b,c,d] 乘上 [2] 生成 [a b c d a b c d] 参数:

- input: 一个 Tensor, 一维或者更高维度
- multiples: 一个 Tensor。必须是下面的类型:int32,int64。一维长度必须和输出维度相同。
- name: 操作的名字 (可选)

Returns : 一个 Tensor, 和 input 有相同的数据类型

### 11.3.28 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。

通过沿着 axis 维度从 value 剪切 num 个 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 个 tensor 是切片 value[i,:,:,:], output 的每个 tensor 形状为 (B,C,D). (注意沿着维度 unstack) 如果 axis == 1 输出的第 i 个 tensor 是切片的值 [:,i,:,:] 和每个 output 的每个输出将有形状 (A,C,D)。tf.ustack(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)。

## 11.3.29 tf.contrib.rnn

## AttentionCellWrapper

继承自 RNNCell，是一个基本的 attention cell 包装器。

graph	__init__
losses	__call__
non_trainable_variables	__deepcopy__
non_trainable_weights	add_loss
output_size	add_update
scope_size	add_variable
state_size	apply
trainable_variable	build
trainable_weights	call
updates	get_losses_for
vaiables	get_updates_for
weights	zero_state

## 11.4 tf.Vairable

### 11.4.1 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()
```

```

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

## 11.4.2 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\_VARIABLE。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\_VARIABLE 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\_VARIABLE。
- 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: 数值类型的坟墓 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。

### \_\_\_\_invet\_\_\_\_

```

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。

### 11.4.3 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:

```

# 2-D tensor 'a'  

1 a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3]) => [[1. 2. 3.]  

2                                         [4. 5. 6.]]  

3  

# 2-D tensor 'b'  

4 b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2]) => [[7. 8.]  

5                                         [9. 10.]  

6                                         [11. 12.]]  

7  

8 c = tf.matmul(a, b) => [[58 64]  

9                                         [139 154]]  

10  

# 3-D tensor 'a'  

11 a = tf.constant(np.arange(1, 13, dtype=np.int32),  

12                         shape=[2, 2, 3]) => [[[1. 2. 3.]  

13                                         [4. 5. 6.]],  

14                                         [[7. 8. 9.]  

15                                         [10. 11. 12.]]]  

16  

17  

# 3-D tensor 'b'  

18 b = tf.constant(np.arange(13, 25, dtype=np.int32),  

19                         shape=[2, 3, 2]) => [[[13. 14.]  

20                                         [15. 16.]  

21                                         [17. 18.]],  

22                                         [[19. 20.]  

23                                         [21. 22.]  

24                                         [23. 24.]]]  

25  

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\_\_, \_\_ror\_\_, \_\_rpow\_\_ 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。

## 11.5 tf.image

### 11.5.1 adjust\_brightness

```
1 adjust_brightness(
2   image,
3   delta)
```

调整灰度或者 RGB 图像的亮度。

这是一个方便的转换 RGB 到浮点表达，调整亮度，然后转换它回原始的数据类型。如果一些调整是限制的它适当的最小化多余转化。delta 的值被增加到 tensor 类型的 image 的所有组件，不经 image 和 delta 在添加（image 被放大，如果它是固定点表达的化）前被转化为 float。对于通常的图像，delta 应该在 [0,1) 之间，正如浮点表达式被添加到图像上，这里像素值在 [0,1) 之间。

参数:

- image: 一个 tensor
- delta: 一个标量，添加上的像素值

Returns : 和 image 类型和形状相同的亮度调整

### 11.5.2 adjust\_contrast

adjust\_contrast(images,contrast\_factor) 调整 RGB 或者灰度图像的对比度。

这是一个方便的转换 RGB 图像到浮点表达，调整他的对比度然后转化回原始的数据类型。如果一些调整是限制的它明智的最小化多余转化，image 是一个至少 3 维的 tensor。三维解释为 [height,width,channels]，另一个维度表示是一个图像集合，像 [batch,height,width,channels]

对比度调整依赖于每个图像的通道，对每个通道 Op 计算通道中图像像素的均值然后调整每个组件 x 为  $(x - mean) * contrast factor + mean$  参数：

- images: 需要调整的图像，至少是 3 维
- contrast\_factor: 一个调整对比度的浮点数

Returns : 调整对比度后的一张或者多张图片

### 11.5.3 adjust\_gamma

```

1 adjust_gamma(
2     image,
3     gamma=1,
4     gain=1)

```

在输入图片上执行 Gamma 修正，广为人知的 Power Law Transform，函数根据方程  
 $Out = In^{\gamma} \cdot gain$  变换输入图像像素之后缩放像素到 0-1 之间。

- 参数:image: 一个 Tensor。ganna: 一个标量，非负实数，gain: 一个标量，常量乘子。

返回 : 一个 Tensor，Gamma 修正后的输出。

Raises : ValueError: 如果 gamma 是负数

注意：如果 gamma 大于 1，直方图将向左移，输出图像将比输入图像暗。如果 gamma 小于 1，直方图将向右移，输出图像将比输入图像亮。参考[Gamma\\_correction](#)

#### 11.5.4 adjust\_hug

```

1 adjust_hue(
2     image,
3     delta,
4     name=None
5 )

```

调整 RGB 图像的色调。

这是一个很方便的转化 RBG 图像为浮点表达的方法，转化它为 HSV，添加一些偏移到 hue 通道，转化回 RGB 变回原来的数据类型。如果一些调整被限制它智能的最小化冗余转化。

image 是一个 RGB 图像。图像的色调通过转化图像为 HSV 通过 delta 旋转色调通道 H。  
 图像然后转化为 RGB。delta 必须是在 [-1,1]

参数:

- image: 单张或者多张 RGB 图像，形状的最后一个元素必须是 3
- delta: 一个浮点数，如歌添加到色调通道
- name: 操作的名字

Returns : 调整后的一张或者多张图像，和 image 有相同的 Dtype

### 11.5.5 adjust\_saturation

```

1 adjust_saturation(
2     image,
3     saturation_factor,
4     name=None
5 )

```

调整 RGB 图像的饱和度。

这是一个很方便的转化 RGB 图像为浮点表达的方法，转化为 HSV，添加一个偏移到 S 通道，转化回 RGB 然后变回原来的数据类型。如果多个调整被限制它智能的最小化冗余转化的数量。

image 是一个 RGB 图像，图像饱和度通过转化图像为 HSV 饭后 S 通道乘上 saturation\_factor 然后接企鹅。图像然后被转换为 RGB。

参数:

- image: 一张或者多张 RGB 图像，形状最后一维的值必须是 3
- saturation\_factor: 浮点数，饱和度乘上的因子
- name: 一个操作的名字

Returns : 调整后的一张或者多张形状和 Dtype 与 image 相同的的图像

### 11.5.6 central\_crop

```

1 central_crop(
2     image,
3     central_fraction
4 )

```

剪裁图像的中心区域

移除图像的部分外围保留每个维度的中心区域，如果我们制定 central\_fraction=0.5, 这个函数返回一个如相面图中 X 表示区域

```

1 _____
2 | |
3 | XXXX |
4 | XXXX |
5 |         | where "X" is the central 50% of the image.
6 _____

```

参数:

- image: 三维浮点数, Tensor 的形状 [height, width, depth]

- central\_fraction: float(0,1], 建材区域的比例

Raises Value: 如果 central\_crop\_fraction 不在 (0,1] 范围内。 )

Returns :3 维浮点 Tensor

#### 11.5.7 decode\_bmp

```

1 decode_bmp(
2   contents,
3   channels=None,
4   name
5 )

```

解码 BMP 编码的图像为 uint8 的 tensor, 属性 channels 预示着解码后的图像的通道 接受下面值:

- 0: 用 BMP 编码图像的通道
- 3: 输出 RGB 图像
- 4: 输出 RGBA 图像

参数:

- contents: 一个 string 类型的 Tensor。0 维。BMO 编码图像
- channels: 一个 int 选项, 默认为 0
- name: 操作的名字

输出 : 类型为 uint8 类型的 Tensor, 形状为 [height, width, channels],RGB 顺序。

#### 11.5.8 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。

---

### 11.5.9 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。
- ratio: 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]

### 11.5.10 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。

#### 11.5.11 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: 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。

#### 11.5.12 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。

---

### 11.5.13 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。

### 11.5.14 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 或 3 维 Tensor, 形状为 [height,width,channels]
- size: 一维 int32 整型 Tensor 元素: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]

## 11.6 tf.feature\_column

接收表示特征的工具。

### 11.6.1 bucketized\_column

```
1 bucketized_column(
2     source_column,
3     boundaries
4 )
```

代表离散化的稠密输入，bucket 包括左边界不包括右边界，也就是说 boundaries=[0.,1.,2.]  
生成 buckets(-ing,0.],[0.,1.],[1.,2.),[2,inf)。例如如果输入是：

```
1 boundaries = [0, 10, 100]
2 input tensor = [[-5, 10000]
3                 [150,    10]
4                 [5,      100]]
```

输出是：

```
1 output = [[0, 3]
2           [3, 2]
3           [1, 3]]
```

例如：

```
1 price = numeric_column('price')
2 bucketized_price = bucketized_column(price, boundaries=[...])
3 columns = [bucketized_price, ...]
4 features = tf.parse_example(..., features=make_parse_example_spec(columns))
5 linear_prediction = linear_model(features, columns)
6
7 # or
8 columns = [bucketized_price, ...]
9 features = tf.parse_example(..., features=make_parse_example_spec(columns))
10 dense_tensor = input_layer(features, columns)
```

bucketized\_column 也能被用 crossed\_column 传递：

```
1 price = numeric_column('price')
2 # bucketized_column converts numerical feature to a categorical one.
3 bucketized_price = bucketized_column(price, boundaries=[...])
4 # 'keywords' is a string feature.
5 price_x_keywords = crossed_column([bucketized_price, 'keywords'], 50K)
6 columns = [price_x_keywords, ...]
```

```

7 features = tf.parse_example(..., features=make_parse_example_spec(columns))
8 linear_prediction = linear_model(features, columns)

```

参数:

- source\_column: numeric\_column 生成的一维稠密列
- boundaries: 指定边界的一个浮点数列表或元组

Returns :一个 \_BucketizedColumn

Raises :

- ValueError: 如果 source\_column 不是一个数值列，或者不是一维。
- ValueError: 如果 boundaries 不是一个列表或者元组。

### 11.6.2 categorical\_column\_with\_hash\_bucket

```

1 categorical_column_with_hash_bucket(
2     key,
3     hash_bucket_size,
4     dtype=tf.string
5 )

```

当你的稀疏特征是字符串或者整数格式的时候你想布置你的输入进入一个有限的散列的 bucket, output\_id=Hash(input\_feature\_string)%bucket\_size 输入不字典 features,features[key] 可以是 Tensor 或者 SparseTensor。如果是 Tensor 整数缺少值用-1 表示，“字符串用”表示。注意这些值独立于参数 default\_value 参数.

```

1 keywords = categorical_column_with_hash_bucket("keywords", 10K)
2 columns = [keywords, ...]
3 features = tf.parse_example(..., features=make_parse_example_spec(columns))
4 linear_prediction = linear_model(features, columns)
5
6 # or
7 keywords_embedded = embedding_column(keywords, 16)
8 columns = [keywords_embedded, ...]
9 features = tf.parse_example(..., features=make_parse_example_spec(columns))
10 dense_tensor = input_layer(features, columns)

```

参数:

- key: 一个用来识别输入的独一无二的字符串。它用做列的名字和特征的 key。
- hash\_bucket\_size: 一个大于 1 的整数，bucket 的树木。

- dtype:feature 的类型。仅仅字符串和整数型支持、

Returns 一个 \_HashedCategoricalColumn

Raises :

- ValueError:hash\_bucket\_size 不大于 1。
- ValueError:dtype 不是字符串或者整数。

### 11.6.3 categorical\_column\_with\_identity

```

1 categorical_column_with_identity(
2     key,
3     num_buckets,
4     default_value=None
5 )

```

一个返回 identity 值的 \_categoricalColumn, 当你的输入是 [0,num\_buckets) 之间的整数, 你想用哪个输入作为分类的 ID, 值在范围外的为默认值是指定, 否则将不能指定。通常被用于整数索引的连续范围, 但是它不是必须的如归哦一些 ID 没有使用这就会很低效考虑使用 categorical\_column\_with\_hash\_bucket, 对于输入字典 features,features[key] 可以是 Tensor 或者 SparseTensor, 如果是 Tensor, 用-1 表示整数类型的缺失, ”表示字符串类型的缺失, 注意值独立于 default\_value 参数, 下面的例子中每个输入在 [0,100000) 中每指定相同的值, 另一个输入赋值 default\_value 0。

线性模型:

```

1 video_id = categorical_column_with_identity(
2     key='video_id', num_buckets=1000000, default_value=0)
3 columns = [video_id, ...]
4 features = tf.parse_example(..., features=make_parse_example_spec(columns))
5 linear_prediction, _, _ = linear_model(features, columns)

```

迁入 DNN 模型:

```

1 columns = [embedding_column(video_id, 9), ...]
2 features = tf.parse_example(..., features=make_parse_example_spec(columns))
3 dense_tensor = input_layer(features, columns)

```

参数:

- 一个独一无二的字符串识别输入特征。被用作特征解析配置列表名字和字典的键, 特征对象和特征列。
- num\_bucket 范围为输入和输出 [0,num\_buckets)

- default\_value: 如果为 None, 列的图操作将因为草畜输入范围而失败, 因此它的值必须在 [0,num\_buckets)

Returns :一个返回识别值的 \_CategoricalColumn

Raises :

- ValueError: 如果 num\_buckets 小于 1。
- ValueError: 如果 default\_value 不在 [0,num\_buckets)

#### 11.6.4 categorical\_column\_with\_vocabulary\_file

```

1 categorical_column_with_vocabulary_file(
2     key,
3     vocabulary_file,
4     vocabulary_size,
5     num_oov_buckets=0,
6     default_value=None,
7     dtype=tf.string
8 )

```

一个包含吃会文件的 \_CategoricalColumn, 当你的输入是字符串或者整数格式时使用, 你有一个词汇文件映射每个值到一个整数 ID, 模型, 草果范围的值被忽略, num\_oov\_buckets 和 default\_value 指定如何包含超出词汇表范围的值, 对于输入字典 features,features[key] 可以是 Tensor 或者 SparseTensor, 如果是 Tensor, 整数型缺少值-1 表示, 字符串缺少值用”表示, 注意这些值是独立于参数 default\_value 参数, 样本 num\_oov\_buckets: 文件'us/states.txt' 包含 50 行每行美国州的所写的两个字符, 所有文件中的输入值被负值为 ID0-49, 对应的行数, 所有其他得知被打散赋值 ID 50-54。

```

1 states = categorical_column_with_vocabulary_file(
2     key='states', vocabulary_file='/us/states.txt', vocabulary_size=50,
3     num_oov_buckets=5)
4 columns = [states, ...]
5 features = tf.parse_example(..., features=make_parse_example_spec(columns))
6 linear_prediction = linear_model(features, columns)

```

例如 default\_value: 文件'us/state.txt' 包含 51 行, 第一行是'XX', 其它的 50 行有两个美国州的缩写, 'XX' 和其他文件中缺失的值被赋值为 ID0, 所有其它值在 1-50。

```

1 states = categorical_column_with_vocabulary_file(
2     key='states', vocabulary_file='/us/states.txt', vocabulary_size=51,
3     default_value=0)
4 columns = [states, ...]

```

---

```
5 features = tf.parse_example(..., features=make_parse_example_spec(columns))
6 linear_prediction, _, _ = linear_model(features, columns)
```

增加一个 embedding:

```
1 columns = [embedding_column(states, 3), ...]
2 features = tf.parse_example(..., features=make_parse_example_spec(columns))
3 dense_tensor = input_layer(features, columns)
```

参数:

- key: 一个独一无二的识别输入特征的字符串，他被用作列的名字。字典的用于特征解析配置，特征 Tensor 对象和其它列的键。
- vocabulary\_file: 词汇表的文件名。
- vocabulary\_size: 词汇表中的元素的个数，必须不大于 vocabulary\_file 的长度，如果小于长度之后的值将被忽略。
- num\_oov\_buckets: 一个非负整数超过词汇 bucket 的数量，所有超过词汇表的输入将被赋值为在基于散列输入 [vocabulary,vocabulary\_size+num\_oov\_buckets) 范围的 ID，一个正的 num\_oov\_buckets 不能被 default\_value 指定。
- default\_value: 超过特征值范围的整数 ID，默认为-1，这可能不被一个正 num\_oov\_buckets 指定。
- dtype: 特征的类型，仅仅字符串和整数支持

Returns 一个词汇表文件 \_CategoricalColumn

Raise :ValueError:Vocabulary\_file 文件缺失，vocabulary\_size 缺失或者小于 1,num\_oov\_buckets 是一个负整数，num\_oov\_buckets 和 default\_value 被同时指定，dtype 不是整数或者字符串。

### 11.6.5 categorical\_column\_with\_vocabulary\_list

```
1 categorical_column_with_vocabulary_list(
2     key,
3     vocabulary_list,
4     dtype=None,
5     default_value=-1,
6     num_oov_buckets=0
7 )
```

一个存储词汇表 `_CategoricalColumn`, 当你的输入是一个字符串或者整数格式时使用, 你有一个存储字徽标映射, 每个值到一个整数 ID., 默认超过词汇表范围的值被忽略, 用 `num_oov_buckets` 和 `default_value` 指定如何包含操作词汇表的值, 对于字典 `features[features[key]]` 既可以是 `Tensor` 也可以是 `SparseTensor`, 如果是 `Tensor`, 整型缺失值用-1 表示, “字符串缺失值” 表示注意这些值独立于 `default_value` 参数, `num_oov_buckets` 样本: 在下面的例子中, 每个 `vocabulary_list` 中的输入被赋值 ID0-3 对应他的索引, 所有的其他输入被打散赋值为 4-5.

```

1 colors = categorical_column_with_vocabulary_list(
2     key='colors', vocabulary_list=( 'R', 'G', 'B', 'Y'),
3     num_oov_buckets=2)
4 columns = [colors, ...]
5 features = tf.parse_example(..., features=make_parse_example_spec(columns))
6 linear_prediction, _, _ = linear_model(features, columns)
```

样本 `default_value`: 下面的例子每个 `vocabulary_list` 的输入被赋值一个 ID0-4 对应他的索引, 所有其它的输入被复制 `defaulr_vale` 0.

```

1 colors = categorical_column_with_vocabulary_list(
2     key='colors', vocabulary_list=( 'X', 'R', 'G', 'B', 'Y'), default_value=0)
3 columns = [colors, ...]
4 features = tf.parse_example(..., features=make_parse_example_spec(columns))
5 linear_prediction, _, _ = linear_model(features, columns)
```

做一个 embedding:

```

1 columns = [embedding_column(colors, 3), ...]
2 features = tf.parse_example(..., features=make_parse_example_spec(columns))
3 dense_tensor = input_layer(features, columns)
```

参数:

- `key`: 一个独一无二的字符串识别输入特征。它被用作列的名字和特征解析配置的字典的 `key`, 特征 `Tensor` 对象和特征列。
- `vocabulary_list`: 一个定义可顺序列带的词汇表, 每个特征被映射到它在 `vocabulary_list` 的值的索引必须是可转换的类型。
- `dtype`: 特征的类型, 仅仅支持字符串和整数类型。如果为 `None` 它将从 `vocabulary_list` 推理。
- `default_value`: 超过词汇表返回的整数 ID 值, 默认为-1, 不能被指定为一个正的 `num_oov_buckets`.

- num\_oov\_buckets: 非负整数, 超过词汇表 bucket 的数量, 所有的处处词汇表的输入将被赋值 ID[len(vocabulary\_list),len(vocabulary\_list)+num\_oov\_buckets) 基于山裂输入值, 一个正的 num\_oov\_buckets 不能被 default\_value 指定。

Returns : 一个存储的词汇表 \_CategoricalColumn。

Raise :ValueError: 如果 vocabulary\_list,num\_oov\_buckets 是负整数, num\_oov\_buckets 和 default\_value 同时被指定, dtype 不是整数或者字符串。

#### 11.6.6 cross\_column

```

1 crossed_column(
2     keys,
3     hash_bucket_size,
4     hash_key=None
5 )

```

返回执行交叉分类特征的一列, 交叉特征被 hash\_bucket\_size 打散, 变换可能被 Hash%hash\_bucket\_size 打散, 例如输入特征是:

- 第一个键值访问的 SparseTensor。

```

1   shape = [2, 2]
2 {
3     [0, 0]: "a"
4     [1, 0]: "b"
5     [1, 1]: "c"
6 }
7 \end{itemize}
8 \item 被第二个键值访问的 SparseTensor:
9 \begin{itemize}
10   shape = [2, 1]
11 {
12     [0, 0]: "d"
13     [1, 0]: "e"
14 }
15

```

交叉特征:

```

1   shape = [2, 2]
2 {
3     [0, 0]: Hash64("d", Hash64("a")) % hash_bucket_size
4     [1, 0]: Hash64("e", Hash64("b")) % hash_bucket_size

```

```

5     [1, 1]: Hash64("e", Hash64("c")) % hash_bucket_size
6 }
```

通过交叉字符串特征创建一个线性模型:

```

1 keywords = categorical_column_with_vocabulary_file(
2     'keywords', '/path/to/vocabulary/file', vocabulary_size=1K)
3 keywords_x_doc_terms = crossed_column([keywords, 'doc_terms'], 50K)
4 columns = [keywords_x_doc_terms, ...]
5 features = tf.parse_example(..., features=make_parse_example_spec(columns))
6 linear_prediction = linear_model(features, columns)
```

如果输入特征是数值类型, 你可以用 categorical\_column\_with\_identity 或者 bucketized\_column:

```

1 # vertical_id is an integer categorical feature.
2 vertical_id = categorical_column_with_identity('vertical_id', 10K)
3 price = numeric_column('price')
4 # bucketized_column converts numerical feature to a categorical one.
5 bucketized_price = bucketized_column(price, boundaries=[...])
6 vertical_id_x_price = crossed_column([vertical_id, bucketized_price], 50K)
7 columns = [vertical_id_x_price, ...]
8 features = tf.parse_example(..., features=make_parse_example_spec(columns))
9 linear_prediction = linear_model(features, columns)
```

为了在 DNN 模型中使用交叉列, 你需要增加它到一个嵌入列:

```

1 vertical_id_x_price = crossed_column([vertical_id, bucketized_price], 50K)
2 vertical_id_x_price_embedded = embedding_column(vertical_id_x_price, 10)
3 dense_tensor = input_layer(features, [vertical_id_x_price_embedded, ...])
```

参数:

- keys: 一个可以迭代的识别被交叉的特征, 每个元素可以是字符串 (用相关的特征必须是字符串类型), \_CategoricalColumn: 将用变形的列通过 tensor 生成, 不支持散列的绝对的列。
- hash\_bucket\_size: 一个大于 1 的整数, buckets 的数量。
- hash\_key: 指定将被 FingerprintCat64 使用的 hash\_key 结合 SparseCrossOp fingerprint 交叉。

Returns : 一个 \_CrossedColumn

Raises :ValueError: 如果 len(keys)<2, 如果任何 keys 既不是字符串也不是 \_CategoricalColumn, 如果任何 keys 是 \_ashedCategoricalColumn, 如果 hash\_buckets\_size<1

### 11.6.7 embedding\_column

```

1 embedding_column(
2     categorical_column ,
3     dimension ,
4     combiner='mean' ,
5     initializer=None ,
6     ckpt_to_load_from=None ,
7     tensor_name_in_ckpt=None ,
8     max_norm=None ,
9     trainable=True
10 )

```

从稀疏绝对输转换 `_CategoricalColumn`, 当你的输入稀疏但是你想转换他们为一个稠密的表达时使用, 输入必须被任何 `categorical_column_*` 函数创建的一个 `_CategoricalColumn`, 下面的雷子是一个识别 DNN 模型列的例子:

```

1 video_id = categorical_column_with_identity(
2     key='video_id', num_buckets=1000000, default_value=0)
3 columns = [embedding_column(video_id, 9), ...]
4 features = tf.parse_example(..., features=make_parse_example_spec(columns))
5 dense_tensor = input_layer(features, columns)

```

参数:

```

1 \item categorical\_column:一个 categorical\_column\_with\_* 函数创建的 _CategoricalColumn, 这个列生成代表嵌入的查找的稀疏ID。
2 \item dimension:一个整数指定嵌入的维度, 必须大于0.
3 \item combiner:一个字符串指定如果有多个输入在一行时如何减少, 当前的 'mean', 'sqrt' 和 'sum' 被支持, 'mean' 默认, 'sqrt' 经常得到好的精度, 特别是词袋模型, 每一个可以被认为每列作为样本级别的正规化。对于更多信息, 查看 tf.embedding\_lookup\_sparse.
4 \item initializer:一个变量初始化器函数用于embedding变量的初始化, 如果没有在一定默认为 tf.truncated_normal_initializer, 均值为0标准差为 $1/\sqrt{\text{dimension}}$ .
5 \item ckpt\_to\_load\_from: string 代表从保存列的权重checkpoint名字、样例, 如果 tensor\_name\_load\_from 不是None
6 \item max\_norm:如果不为None, 嵌入值用l2中正规化
7 \item trainable:驶入嵌入能被训练, 默认为true
8 \item [Returns]:从稀疏输入转化的 _DenseColumn
9 \item [Raises]: ValueError, 如果 dimension 不大于0, 如果一个 chpt\_to\_load\_from 和 tensor\_name\_in\_ckpt 被指定, 如果 initializer 被指定不被调用。

```

### 11.6.8 indicator\_column

代表给定绝对的 mult-hot, 用于打包任何 categorical\_column\_\* 用 embedding\_column  
如果输入是稀疏的:

```

1 name = indicator_column(categorical_column_with_vocabulary_list(
2     'name', [ 'bob' , 'george' , 'wanda' ])
3 columns = [name, ...]
4 features = tf.parse_example(..., features=make_parse_example_spec(columns))
5 dense_tensor = input_layer(features, columns)
6
7 dense_tensor == [[1, 0, 0]] # If "name" bytes_list is ["bob"]
8 dense_tensor == [[1, 0, 1]] # If "name" bytes_list is ["bob", "wanda"]
9 dense_tensor == [[2, 0, 0]] # If "name" bytes_list is ["bob", "bob"]

```

参数: categorical\_column: 一个由 categorical\_column\_with\* 或者 crossed\_column 创建  
的 \_CategoricalColumn, 返回一个 \_IndicatorColumn。

### 11.6.9 input\_layer

```

1 input_layer(
2     features,
3     feature_columns,
4     weight_collections=None,
5     trainable=True
6 )

```

基于给定的 feature\_column 返回一个出来的 Tensor 作为输入层, 通常在训练数据上被  
FeatureColumns 描述的见得样本, 在模型的第一层列的方向数据应该被转化为一个  
Tensor。例如:

```

1 price = numeric_column('price')
2 keywords_embedded = embedding_column(
3     categorical_column_with_hash_bucket("keywords", 10K), dimensions=16)
4 columns = [price, keywords_embedded, ...]
5 features = tf.parse_example(..., features=make_parse_example_spec(columns))
6 dense_tensor = input_layer(features, columns)
7 for units in [128, 64, 32]:
8     dense_tensor = tf.layers.dense(dense_tensor, units, tf.nn.relu)
9 prediction = tf.layers.dense(dense_tensor, 1)

```

参数:

- features: 一个从 key 映射到 tensor。\_FeatureColumn 通过通过这些 key 查找, 例如 numeric\_column('price') 将查看字典中的'price', 值可以是 SparseTensor 护着 Tensor

取决于 \_FeatureColumn。

- feature\_columns: 一个可迭代的包含 FeatureColumn 用作你的模型的输入所有的这些都应该是继承自 \_DenseColumn 的实例, 像 numeric\_column, embedding\_column, bucketized\_column, 如果你有绝对的特征, 你可以用 embedding\_column 或者 indicator\_column 打包他们。
- weight\_collections: 一个将被添加到集合列表的名字, 注意变量也被增加到集合 tf.GraphKeys.GLOBAL 和 ops.GraphKeys.MODEL\_VARIABLES.
- trainable: 如果为 True 添加变量到图集合 GraphKeys.TRAINABLE\_VARIABLE.

Returns : 一个代表模型输入层 Tensor, 他的形状是 (batch\_size, first\_layer\_dimension) 数据类型是 float32, first\_layer\_demension 倍 feature\_columns 决定。

Raises : 如果一个 item 在 feature\_columns 不是一个 \_DenseColumn。

### 11.6.10 linear\_model

```

1 linear_model(
2     features ,
3     feature_columns ,
4     units=1,
5     sparse_combiner='sum',
6     weight_collections=None,
7     trainable=True
8 )

```

返回一个基于 feature\_columns 给定的线性预测 Tensor, 返回一个基于给定 feature\_column 线性预测, 这个函数生成基于输出维度单位的权重和, 权重和在分类问题上涉及分类问题。它涉及线性回归模型问题的预测, 注意不支持列: linear\_model 对待绝对列作为 indicator\_column 当 input\_layer 明确要求用一个 embedding\_column 或者 indicatorcolumn 打包他们。例如:

```

1 price = numeric_column('price')
2 price_buckets = bucketized_column(price, boundaries=[0., 10., 100., 1000.])
3 keywords = categorical_column_with_hash_bucket("keywords", 10K)
4 keywords_price = crossed_column('keywords', price_buckets, ...)
5 columns = [price_buckets, keywords, keywords_price ...]
6 features = tf.parse_example(..., features=make_parse_example_spec(columns))
7 prediction = linear_model(features, columns)

```

参数:

- features: 一个从 key 到 tensor 的映射, \_FeatureColumn 查找这些键, 例如 numeric\_column('price') 将查找字典中的'price', Value 是 Tensor 或者 SparseTensor 取决于 \_FeatureColumn.
- feature\_columns: 一个可迭代的 FeatureColumn 用做模型的输入, 所有的项目都应该是继承自 \_FeatureColumn 的实体。
- units: 一个整数, 输出空间的维度, 默认值为 1。
- sparse\_combiner: 一个字符串指定如果稀疏列是多值时如何减少的字符串, 当前"mean","sqrt" 和"sum" 支持, sum 默认是 sqrt 经常获得好的精度。特别的在字列的词袋模型, 它结合每个列:
  - sum: 没有正规化列中的特征
  - mean: 在列的特征上做 l1 正则化
  - sqrt: 在猎德特征上做 l2 正则化。
- weight\_collections: 一个每个变量将被添加到集合的名字, 注意变量将被添加到集合 tf.GraphKeys.GLOBAL\_VARIABLE 和 ops.Graphkeys.MODEL\_VARIABLE
- trainable: 如果为 True 将增加变量到图集合 Graphkeys.TRAINABLE\_VARIABLE

Returns :一个代表预测线性模型的预测的 Tensor, 它的形状 (batch\_size,units) 和它的 dtype 为 float32

Raises :ValueError, 如果在 feature\_columns 中的项目既不是一个 \_DenseColumn 也不是 \_CategoricalColumn

### 11.6.11 make\_sparse\_example\_spec

从输入特征列创建稀疏空间字典, 返回字典可以被用作 tf.parse\_example 的 features 参数。

```

1 # Define features and transformations
2 feature_b = numeric_column(...)
3 feature_c_bucketized = bucketized_column(numeric_column("feature_c"), ...)
4 feature_a_x_feature_c = crossed_column(
5     columns=["feature_a", feature_c_bucketized], ...)
6
7 feature_columns = set(
8     [feature_b, feature_c_bucketized, feature_a_x_feature_c])
9 features = tf.parse_example(
10    serialized=serialized_examples,
11    features=make_parse_example_spec(feature_columns))

```

下面的例子，make\_sparse\_example\_spec 将返回字典：

```

1 {
2     "feature_a": parsing_ops.VarLenFeature(tf.string),
3     "feature_b": parsing_ops.FixedLenFeature([1], dtype=tf.float32),
4     "feature_c": parsing_ops.FixedLenFeature([1], dtype=tf.float32)
5 }
```

参数:feature\_columns: 一个包含所有特征列的迭代器，所有的项目应该是继承自  
\_FeatureColumn 的实体。返回一个映射每个特征键到 FixedlenFeature 或者  
VarLenFeature 值，异常 ValueError: 如果任何给定的 feature\_columns 不包含一个  
\_FeatureColumn 实体。

### 11.6.12 numeric\_column

```

1 numeric_column(
2     key,
3     shape=(1,),
4     default_value=None,
5     dtype=tf.float32,
6     normalizer_fn=None
7 )
```

表示真实值或者数值特征，例如：

```

1 price = numeric_column('price')
2 columns = [price, ...]
3 features = tf.parse_example(..., features=make_parse_example_spec(columns))
4 dense_tensor = input_layer(features, columns)
5
6 # or
7 bucketized_price = bucketized_column(price, boundaries=[...])
8 columns = [bucketized_price, ...]
9 features = tf.parse_example(..., features=make_parse_example_spec(columns))
10 linear_prediction = linear_model(features, columns)
```

参数：

- key: 一个独一无二的字符串识别输入特征，它被用作特征解析配置，特征 Tensor 对象和特征列的名字
- shape: 一个指定 Tensor 形状的迭代器整数，一个整数可以给单一为，一个整数意味着一个 Tensor 的宽，Tensor 代表将有形状 [batch\_size]+shape

- dtype: 定义值的数据类型， 默认是 tf.float32, 必须是一个没有量化的， 真实的整数或者浮点数类型。
- normalizer\_fn: 如果不为 None, 一个函数可以在 default\_value 被用于解析后被正则化。睁着花旗函数接受输入 Tensor 作为参数返回输出 Tensor, 请注意尽管多数情况用这个函数证这话， 他可以没用于多种 Tensorflow 转换。

Return \_NumericColumn

Raise : 如果任何维度不是证书有, default\_value 是一个迭代不但是形状不匹配,default\_value 不兼容的数据类型报 TypeError, 如果任何维度不是整数或者不能转化为 tf.float32 的 dtype 将报 ValueError。

### 11.6.13 weighted\_categorical\_column

```

1 weighted_categorical_column(
2     categorical_column,
3     weight_feature_key,
4     dtype=tf.float32
5 )

```

应用权重到 \_CategoricalColumn, 让输入是一维有值时使用， 例如如果你表示文本文件作为一个子频率的集合， 你可以提供两个平行的稀疏输入特征， 输入 tf.Example 对象:

```

1 [
2   features {
3     feature {
4       key: "terms"
5       value {bytes_list {value: "very" value: "model"}}
6     }
7     feature {
8       key: "frequencies"
9       value {float_list {value: 0.3 value: 0.1}}
10    }
11  },
12  features {
13    feature {
14      key: "terms"
15      value {bytes_list {value: "when" value: "course" value: "human"}}
16    }
17    feature {
18      key: "frequencies"
19      value {float_list {value: 0.4 value: 0.1 value: 0.2}}
}

```

```

20     }
21   }
22 ]

```

---

```

1 categorical_column = categorical_column_with_hash_bucket(
2     column_name='terms', hash_bucket_size=1000)
3 weighted_column = weighted_categorical_column(
4     categorical_column=categorical_column, weight_feature_key='frequencies')
5 columns = [weighted_column, ...]
6 features = tf.parse_example(..., features=make_parse_example_spec(columns))
7 linear_prediction, _, _ = linear_model(features, columns)

```

假设输入字典包含一个 key ‘terms’ 的 SparseTensor，SparseTensor 键‘frequencies’，两个 Tensor 必须有相同的索引和 dense 形状。

参数：

- categorical\_column: 一个 catrgorical\_column\_with函数创建的 \_CategoricalColumn
- weight\_feature\_key: 权重值的字符串键。
- dtype: 权重的类型，像 tf.float32, 仅仅浮点数和整数支持

Returns 一个两个稀疏响亮的组合的 \_CategoricalColumn: 一个代表 id 另一个代表 id 特征的权重值。

Raises :ValueError: 如果 dtype 不能转化为 float32

## 11.7 layer

### 11.7.1 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。

### 11.7.2 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。

### 11.7.3 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。

#### 11.7.4 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短到 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。

### 11.7.5 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。

### 11.7.6 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。

### 11.7.7 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。

#### 11.7.8 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。

### 11.7.9 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。

#### 11.7.10 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。

### 11.7.11 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。

#### 11.7.12 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。

#### 11.7.13 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。

#### 11.7.14 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。

#### 11.7.15 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 channel 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。

## 11.8 tf.train

提供了训练模型的类和函数。

### 11.8.1 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 () he compute\_gradients() 接受一个 gate\_gradients 参数控制 fradient 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。

### 11.8.2 tf.train.slice\_input\_producer

```

1 slice_input_producer(
2     tensor_list ,
3     num_epochs=None ,
4     shuffle=True ,
5     seed=None ,
6     capacity=32 ,
7     shared_name=None ,
8     name=None
9 )

```

生成一个在 tensor\_list 的 Tensor 切片。用队列实现，一个 QueueRunner 被添加到当前的 Graph ‘s 的 QUEUE\_RUNNER 集合。参数:

- tensor\_list: 一个 Tensor 对象的列表，每个 tensor\_list 中的 Tensor 在第一个维度上必须有相同的大小。
- num\_epochs: 一个整数 (可选) 如果指定，slice\_input\_producer 每 num\_epochs 产生一个切片，直到出现OutOfRange 错误。如果不指定，slice\_input\_producer 可以无限次数的循环切片
- shuffle:Boolean。如果为 true，整数在每个 epoch 被随机打乱
- seed: 一个整数 (可选)，如果 shuffle == True 用一个种子
- Capacity: 一个整数，设置队列的容量
- shared\_name:(可选), 如果设置，队列将被在多个会话中通过名字共享
- name: 可选，操作的名字

Returns :一个 tensor 列表，元素是 tensor\_list 中的元素。如果在 tensor\_list 中的 tensor 形状为 [N,a,b,...,z]，对应的输出维度的形状为 [a,b,...,z]

ValueError :如果 slice\_input\_producer 从 tensor\_list 什么都没有产生

### 11.8.3 tf.train.shuffle\_batch

```

1 shuffle_batch(
2     tensors ,
3     batch_size ,
4     capacity ,
5     min_after_dequeue ,

```

```

6     num_threads=1,
7     seed=None,
8     enqueue_many=False,
9     shapes=None,
10    allow_smaller_final_batch=False,
11    shared_name=None,
12    name=None
13 )

```

通过随机大暖 tensor 创建 batches。这个函数添加如下到 Graph:

- 从 tensors 出兑打乱队列
- dequeue\_many 从 queue 创建 batches
- 从 tensor 出队一个 QueueRunner 到 QUEUE\_RUNNER 集合

如果 enqueue\_many 是 False, tensors 假设代表一个单个样本。一个输入 tensor 形状 [x,y,z] 将成为 [batch\_size,x,y,z] 的输出

如果 enqueue\_manny 是 True, tensors 假设表示一批样本, 这里的第一位是样本的索引。tensors 的所有成员应该在第一维度上有相同的代销。如果输入 tensor 有形状 [\* ,x,y,z], 输出将有形状 [batch\_size,x,y,z]

capacity: 参数控制允许队列有多长 返回操作是一个出队操作, 如果输入队列耗尽将抛出 tf.errors.OutOfRangeError。如果这操作被田东另一个输入队列, 他的队列将捕获异常, 然而, 如果这个操作被用在你的主线程中你将响应, 自己捕获。

例如:

```

1 # Creates batches of 32 images and 32 labels.
2 image_batch, label_batch = tf.train.shuffle_batch(
3     [single_image, single_label],
4     batch_size=32,
5     num_threads=4,
6     capacity=50000,
7     min_after_dequeue=10000)

```

你必须确保 (1)shapes 参数被传递, (2) 所有在 tensors 的 tensor 必须有完整定义的形状。ValueError 将在两者之中有一个不满足时出现。如果 allow\_maller\_final\_batch 是 True, 没有足够的元素填充进 batch 时一个比 batch\_size 小的值被返回, 否则未处理的元素江北丢弃/另外, 所有输出 tensor 的静态 shape, 正如通过 shae 参数访问将有一个值为 None 的 Dimension, 依赖于 fixed\_batch\_size 的操作将失败。

参数:

1 \item tensors: 一个入队的列表或者字典

```
2 \item batch\_size:一个从queue获取的批大小
3 \item capacity:一个整数, 队列的最大元素
4 \item min\_after\_dequeue:出队后在队列中最数目, 用于确保一个混合元素的等级
5 \item num\_threads:一个些线程确保入伏你tensor\_list
6 \item seed:一个在queue中的随机种子
7 \item enqueue\_many:是否在tensor\_list中的每个tensor数一个样本
8 \item shape:(可选)每个样本的形状。默认从tensor\_list推断
9 \item allow\_smaller\_final\_batch:(可选)Boolean, 如果为True, 如果队列中没有足够的
    元素允许最后的batch变小
10 \item shared\_name:(可选)如果设置, 队列将在给定名字下在多个会话上共享。
11 \item name: (可选) 操作的名字
12 \item [Returns]:一个tensors类型的列表或tensors字典
13 \item [Raises]:ValueError:如果shape没有指定, 而且不能通过tensors的元素推断
```

@book{Goodfellow-et-al-2016, title=Deep Learning, author=Ian Goodfellow and Yoshua Bengio and Aaron Courville, publisher=MIT Press, note=<http://www.deeplearningbook.org>, year=2016}