

Glimpse

On

Deep Learning

--TensorFlow



TensorFlow 初 了解

By: 程枫
justry2014@outlook.com

深度学习工具对比

库名称	开发语言	平台	特性
Caffe Caffe->Caffe2	C++/python/Matlab	Linux / Mac OS / Windows	在CV领域应用最为广泛
Theano	python	Linux / Mac OS / Windows	发布早，文档丰富 适用于学术领域
PyTorch Torch ->PyTorch	Python (Torch开发语言为Lua/C)	Linux / Mac OS	轻量级、灵活
Tensorflow	C++ / Python	Linux / Mac OS / Windows Android / iOS	功能齐全、适合企业级开发

Keras 一个用于快速构建深度学习原型的高级库，可以用 Tensorflow 和 Theano 的backend，优点是高度的模块化

安装 (Ubuntu 16.04 64bit)

1、安装Anaconda

- (1) 官网下载对应版本
- (2) 在下载目录下执行：`bash Anaconda3-4.4.0-Linux-x86_64.sh`
- (3) 安装Spyder，`sudo apt install spyder`

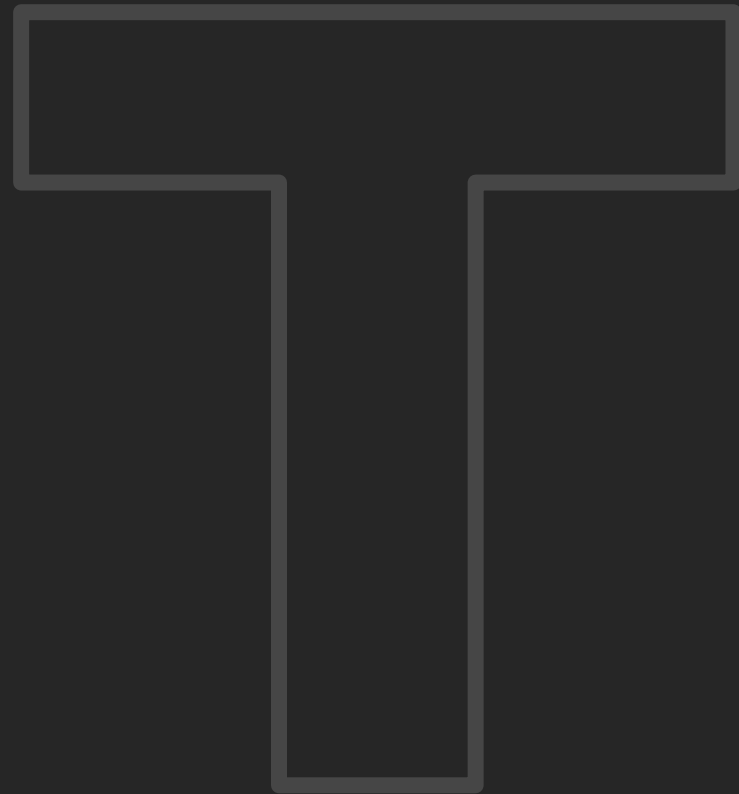
2、安装TensorFlow CPU版本

- (1) 方法一：`pip install tensorflow`
- (2) 方法二：下载源码安装（可对配置进行修改）

第一行代码

```
env python2.7  
import tensorflow as tf  
matrix1 = tf.constant([[3., 3.]])  
matrix2 = tf.constant([[2.],[2.]])  
product = tf.matmul(matrix1, matrix2)  
sess = tf.Session()  
result = sess.run(product)  
print result  
sess.close() #关闭会话
```

运行结果: `[[12.]]`



基本概念

TensorFlow 的 数据模型 为 **tensor（张量）**，可简单理解为类型化的多维数组，0 维张量是一个数字，也成为标量，1 维张量称为“向量”，2 维张量称为矩阵…… 但和 Numpy 中的数组不同是，一个张量中主要保留了三个属性：**name, shape, dtype**

```
#env = python 2.7
import tensorflow as tf
a = tf.constant([1,2,3])
b = tf.constant([2,3,5])
result = tf.add(a,b, name = "add")
print result
```

运行结果 **tensor("add:0", shape=(3,), dtype=int32)**

add:0 表示result这个张量是计算节点“add”输出的第一个结果

shape = (3,) 表示张量是一个一维数组，这个数组的长度是3

dtype=int32 是数据类型，TensorFlow会对参与运算的所有张量进行类型的检查，当类型不匹配时会报错。

基本概念

TensorFlow 的 运行模型 为 **session（会话）**，会话 拥有和管理 TensorFlow 程序运行时的 所有 资源，当计算完成后，需要 **关闭会话** 来帮助系统回收资源，否则就可能出现 资源泄露 的问题，TensorFlow 使用会话的方式主要有如下两种模式：

```
#env = python 2.7
sess = tf.Session()#创建一个会话
sess.run(result)    #运行此会话
Sess.close()        #运行完毕关闭此会话
```

```
#env = python 2.7
with tf.Session() as sess:
    sess.run(result)
#通过python的上下文管理器来使用会话，当
上下文退出时会话管理和资源释放也自动完成
```

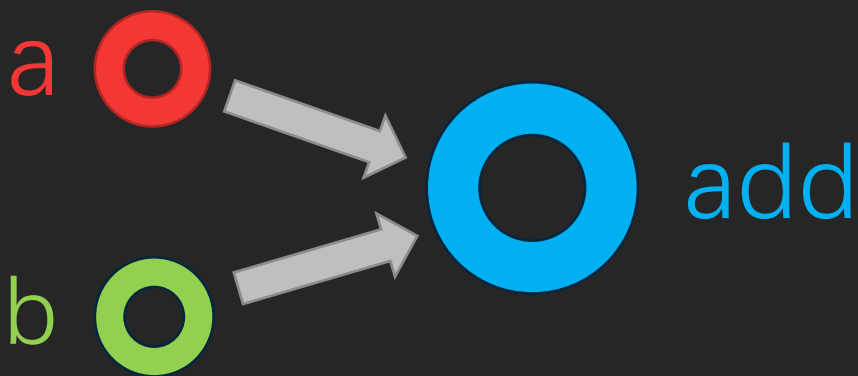
运行结果

```
array([3,5,8], dtype=int32)
```

当使用第一种模式，程序因为异常而退出时，关闭会话的函数就可能不会执行从而导致资源泄露，因此，推荐使用第二种模式

基本概念

TensorFlow 的 计算模型 为 **graph** (图)，一个 TensorFlow 图描述了计算的过程。为了进行计算, 图必须在 **会话** 里被启动. 会话 将图的 **op** (节点) 分发到诸如 CPU 或 GPU 之类的设备上, 同时提供执行 op 的方法. 这些方法执行后, 将产生的 **tensor** 返回. 在 Python 语言中, 返回的 tensor 是 numpy ndarray 对象; 在 C /C++ 语言中, 返回的 tensor 是 tensorflow::Tensor 实例.



TensorFlow 的Tensor 表明了它的数据结构, 而Flow 则直观地表现出张量之间通过计算相互转化的过程。TensorFlow 中的每一个计算都是图上的一个节点, 而节点之间的边描述了计算之间的依赖关系, a, b 为常量, 不依赖任何计算, 而add 计算则依赖读取两个常量的取值。

基本概念

在实现上, TensorFlow 将图形定义转换成 **分布式执行** 的操作。一般不需要显式指定使用 CPU 还是 GPU, TensorFlow 能 **自动检测**。如果检测到 GPU, TensorFlow 会尽可能地利用找到的第一个 GPU 来执行操作。如果机器上有超过一个可用的 GPU, **除第一个外的其它 GPU 默认是不参与计算的**。为了让 TensorFlow 使用这些 GPU, 你必须将 op 明确指派给它们执行。with...Device 语句用来指派特定的 CPU 或 GPU 执行操作

```
with tf.Session() as sess:
    with tf.device("/gpu:1"):
        matrix1 = tf.constant([[3., 3.]])
        matrix2 = tf.constant([[2.],[2.]])
        product = tf.matmul
```

为了获取你的 operations 和 Tensor 被指派到哪个设备上运行, 用 log_device_placement 新建一个 session, 并设置为 True.

```
Sess=tf.Session(config=tf.ConfigProto(log_device_placement=True))
```


变量 Variable

当训练模型时，用变量来 **存储和更新** 参数。变量包含张量 (Tensor) 存放于内存的缓存区。建模时它们需要被明确地 **初始化**，模型训练后它们必须被存储到磁盘。这些变量的值可在之后模型训练和分析是被加载。

当创建一个变量时，你将一个 张量 作为初始值传入构造函数 `Variable()`。TensorFlow 提供了一系列操作符来初始化张量，初始值是 **常量** 或是 **随机值**。在初始化时需要指定张量的 shape，变量的 shape 通常是固定的，但 TensorFlow 提供了高级的机制来重新调整。

```
weights = tf.Variable(tf.random_normal([2,3], stddev = 2), name="weights")
```

这里生成 2×3 的矩阵，元素的均值为0，标准差为2（可用 mean 指定均值，默认值为0）。

随机数生成函数：
`tf.truncated_normal()` 正太分布，偏离平均值超过两个偏差，重新选择
`tf.random_uniform (最小值, 最大值)` 平均分布（控制字符中可加入种子，`seed = n`）

常数生成函数：
全零数组 `tf.zeros([2,3],int32)` 全一数组 `tf.ones()` 全为给定的某数(例如9)：`tf.fill([2,3],9)`
产生给定值得常量 `tf.constant([1,2,3])`

变量 Variable

除了tf.Variable函数，TensorFlow 还提供了tf.get_variable 函数来创建或获取变量。它们的区别在于，对于tf.Variable 函数，变量名称是个可选的参数，通过 name = 'v' 给出，但对于 tf.get_variable 函数，变量名称是必填的选项，它会首先根据变量名试图创建一个参数，如果遇到同名的参数就会报错。

```
>>> import tensorflow as tf
>>> v = tf.Variable(tf.constant(1.0,shape = [1]), name = 'v')
>>> v1 = tf.Variable(tf.constant(1.0,shape = [1]), name = 'v')
>>> print (v==v1)
False
>>> print(v.name)
v:0
>>> print(v1.name)
v_1:0

>>> n = tf.get_variable('n', shape=[1], initializer = tf.constant_initializer(1.0)) # 初始化
>>> n1 = tf.get_variable('n', shape=[1], initializer = tf.constant_initializer(1.0))
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
.....
```

如果需要通过 tf.get_variable 获取一个已经创建的变量。需要通过 tf.variable_scope 函数来生成一个上下文管理器，当 tf.variable_scope 函数使用参数 reuse = True 时（默认值为False），这个上下文管理器内所有 tf.get_variable 函数会直接获取已经创建的变量，如果变量不存在，就会报错。tf.variable_scope 函数是可嵌套的，当不指定值时，reuse 的值将会和外面一层保持一致。

在生成上下文管理器时，也会创建一个命名空间，在命名空间内创建的变量名称，都会带上这个命名空间名作为前缀。可以之间通过带命名空间的变量名来获取其他命名空间下的变量。

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1], initializer=tf.constant_initializer(1.0))
#with tf.variable_scope("foo"):
#    v = tf.get_variable("v", [1]) # 报错
with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v", [1])
print (v == v1) # 运行结果: True
with tf.variable_scope("", reuse=True):
    v5 = tf.get_variable("foo/bar/v", [1]) # 获取 foo/bar 中创建的变量

>>> print(v1.name)
foo/v2:0
```

变量 · 实例

```
# 创建一个 op，其作用是使 state 增加 1
state = tf.Variable(0, name="counter")
one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)
init_op = tf.initialize_all_variables()
with tf.Session() as sess:
    sess.run(init_op)
    print sess.run(state)
    for _ in range(3):
        sess.run(update)
        print sess.run(state)
```

代码中 `assign()` 操作是图所描绘的表达式的一部分，正如 `add()` 操作一样。所以在调用 `run()` 执行表达式之前，它并不会真正执行赋值操作。

通常会将一个统计模型中的参数表示为一组变量。例如，你可以将一个神经网络的 **权重** 作为某个变量存储在一个 `tensor` 中。在训练过程中，通过重复运行训练图，**更新** 这个 `tensor`。

注：`assign(ref, value, validate_shape=None, use_locking=None, name=None)`

函数功能 Update 'ref' by assigning 'value' to it.

变量 Variable

有时候会需要用另一个变量的初始化值给当前变量初始化。`tf.initialize_all_variables()`是并行地初始化所有变量。使用其它变量的值初始化一个新的变量时，可以直接把已初始化的值作为新变量的初始值，或者把它当做tensor计算得到一个值赋予新变量。

```
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),name="weights")
w2 = tf.Variable(weights.initialized_value(), name="w2")
w_twice = tf.Variable(weights.initialized_value() * 0.2, name="w_twice")
```

保存和恢复变量

```
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
init_op = tf.initialize_all_variables()
saver = tf.train.Saver()
with tf.Session() as sess:
    sess.run(init_op)
    save_path = saver.save(sess, "/tmp/model.ckpt")
    print "Model saved in file: ", save_path
```

```
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
saver = tf.train.Saver()
with tf.Session() as sess:
    saver.restore(sess, "/tmp/model.ckpt")
    print "Model restored."
```

用同一个Saver对象来恢复变量。当从文件中恢复变量时，不需要对它们做初始化。`tf.train.Saver()` 默认保存和加载计算图上定义的全部变量，如果只需加载模型的部分变量，例如可以使用`tf.train.Saver([v1])` 构建，这样只有变量 v1 会被加载进来。

如果要对变量名进行修改，可通过字典将模型保存时的变量名和需要加载的变量联系起来。

```
v1 = tf.Variable(tf.constant(1.0, shape=[1]), name = "other-v1")
v2 = tf.Variable(tf.constant(2.0, shape=[1]), name = "other-v2")
saver = tf.train.Saver({"v1": v1, "v2": v2})
```

Xaiver initialization

如果深度学习模型的权重初始化得太小，那信号将在每层间传递时逐渐缩小而难以产生作用；如果权重初始化得太大，那信号将在每层间传递时逐渐放大并导致发散和失效。

Xaiver initialization 由 Xavier Glorot 和 Yoshua Bengio 在 2010年提出，Xavier 让权重满足均值为0，前向传播和反向传播时每一层的方差一致： $\forall i, n_i Var[W^i] = 1$ $\forall i, n_{i+1} Var[W^i] = 1$

但是，实际当中输入与输出的个数往往不相等，于是为了均衡考量，最终我们的权重方差应满足

$$\forall i, Var[W^i] = \frac{2}{n_i + n_{i+1}}$$

因此，Xavier初始化的实现就是：

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

根据均匀分布的性质，它的方差正好是 $\frac{2}{n_i + n_{i+1}}$ ，均值为0。这里实现的是均匀分布（也可以是高斯分布）。

代码中，fan_in就是输入节点的数量，fan_out是输出节点的数量。

Python代码：

```
def xavier_init(fan_in, fan_out, constant = 1):  
    low = -constant * np.sqrt(6.0 / (fan_in + fan_out))  
    high = constant * np.sqrt(6.0 / (fan_in + fan_out))  
    return tf.random_uniform((fan_in, fan_out),  
                             minval=low, maxval=high, dtype=tf.float32)
```

具体推导过程参考论文： *Understanding the difficulty of training deep feedforward neural networks*

Fetch & Feed

为了取回操作的输出内容, 可以在使用 Session 对象的 run() 调用执行图时, 传入一些 tensor, 来取回结果。当获取的多个 tensor 值时, 是在 op 的一次运行中一起获得而不是逐个去获取的。

feed 使用一个 tensor 值临时替换一个操作的输出结果. 你可以提供 feed 数据作为 run() 调用的参数。feed 只在调用它的方法内有效, 方法结束, feed 就会消失。最常见的用例是将某些特殊的操作指定为“feed”操作, 标记的方法是使用 tf.placeholder() 为这些操作创建占位符。

```
input1 = tf.constant(3.0)
input2 = tf.constant(2.0)
input3 = tf.constant(5.0)
intermed = tf.add(input2, input3)
mul = tf.multiply(input1, intermed)
with tf.Session() as sess:
    result = sess.run([mul, intermed])
    print result
```

运行结果 : [21.0, 7.0]

```
input1 = tf.placeholder(tf.float32) #占位符
input2 = tf.placeholder(tf.float32)
output = tf.multiply(input1, input2)
with tf.Session() as sess:
    print sess.run([output], feed_dict={input1:[7.], input2:[2.]})
```

运行结果 : [array([14.], dtype=float32)]

TensorFlow训练神经网络

- 三个步骤：
- (1) 定义神经网络的结构和前向传播的输出结果；
 - (2) 定义损失函数以及选择反向传播优化的算法；
 - (3) 生成会话并且在训练数据上反复运行反向传播优化算法。

完整实例程序

```
import tensorflow as tf
from numpy.random import RandomState
batch_size = 8          #定义训练数据batch大小
w1= tf.Variable(tf.random_normal([2, 3], stddev=1, seed=1))
w2= tf.Variable(tf.random_normal([3, 1], stddev=1, seed=1))
x = tf.placeholder(tf.float32, shape=(None, 2), name="x-input")
y_ = tf.placeholder(tf.float32, shape=(None, 1), name='y-input')
a = tf.matmul(x, w1)      #前向传播过程
y = tf.matmul(a, w2)
#定义损失函数和反向传播的算法
cross_entropy = -tf.reduce_mean(y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0)))
train_step = tf.train.AdamOptimizer(0.001).minimize(cross_entropy)
rdm = RandomState(1)      #通过随机数生成一个模拟数据集
X = rdm.rand(128,2)
Y = [[int(x1+x2 < 1)] for (x1, x2) in X]
```

`tf.clip_by_value(t, clip_value_min, clip_value_max, name=None)` :对tensor数据进行截断操作，应对梯度爆发或者梯度消失的情况

TensorFlow训练神经网络

```
with tf.Session() as sess:
    init_op = tf.global_variables_initializer()
    sess.run(init_op)

    # 输出目前 (未经训练) 的参数取值
    print "w1:", sess.run(w1)
    print "w2:", sess.run(w2)
    print "\n"
    # 训练模型
    STEPS = 5000
    for i in range(STEPS):
        start = (i*batch_size) % 128
        end = (i*batch_size) % 128 + batch_size
        sess.run(train_step, feed_dict={x: X[start:end], y_: Y[start:end]})
        if i % 1000 == 0:
            total_cross_entropy = sess.run(cross_entropy, feed_dict={x: X, y_: Y})
            print("After %d step(s), cross entropy on all data is %g" % (i, total_cross_entropy))

    # 输出训练后的参数取值, 运行结果如右图
    print "\n"
    print "w1:", sess.run(w1)
    print "w2:", sess.run(w2)
```

训练前

```
w1: [[-0.81131822  1.48459876  0.06532937]
      [-2.44270396  0.0992484   0.59122431]]
```

```
w2: [[-0.81131822] [ 1.48459876] [ 0.06532937]]
```

训练后

```
w1: [[-1.9618274  2.58235407  1.68203783]
      [-3.4681716   1.06982327  2.11788988]]
```

```
w2: [[-1.8247149 ] [ 2.68546653] [ 1.41819501]]
```

```
After 0 step(s), cross entropy on all data is 0.0674925
After 1000 step(s), cross entropy on all data is 0.0163385
After 2000 step(s), cross entropy on all data is 0.00907547
After 3000 step(s), cross entropy on all data is 0.00714436
After 4000 step(s), cross entropy on all data is 0.00578471
```


神经网络 · 激活函数

Definition : Use a cascade of many layers of nonlinear processing units for feature extraction and transformation.

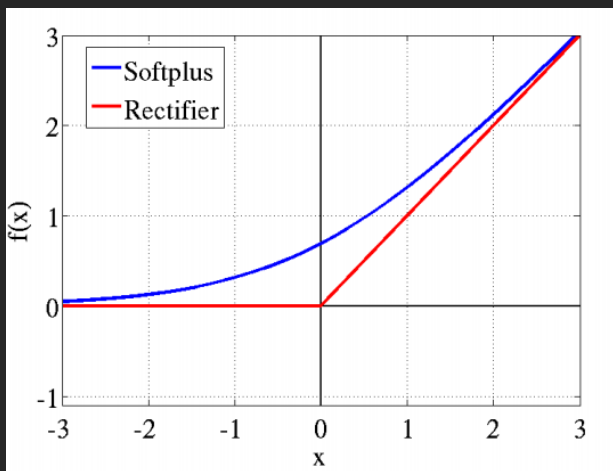
Each successive layer uses the output from the previous layer as input.

-- Wikipedia

线性模型的最大特点是任意线性模型的组合仍然是线性模型，只通过线性变换，任意层的全连接神经网络和单层神经网络没有任何区别，因此非线性是深度学习的重要特性。

目前TensorFlow提供了 7 种不同的非线性激活函数，常见的有：tf.nn.relu、tf.sigmoid 和 tf.tanh

ReLU(Rectified Linear Units)函数

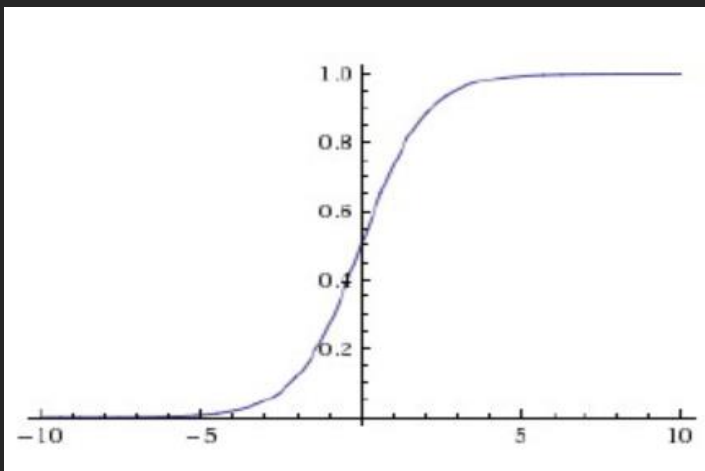


Hard ReLU: $g(x) = \max(0, x)$

Noise ReLU: $\max(0, x + N(0, \sigma(x)))$

该函数的导函数： $g(x)' = 0$ 或 1

S形函数 (Sigmoid Function)

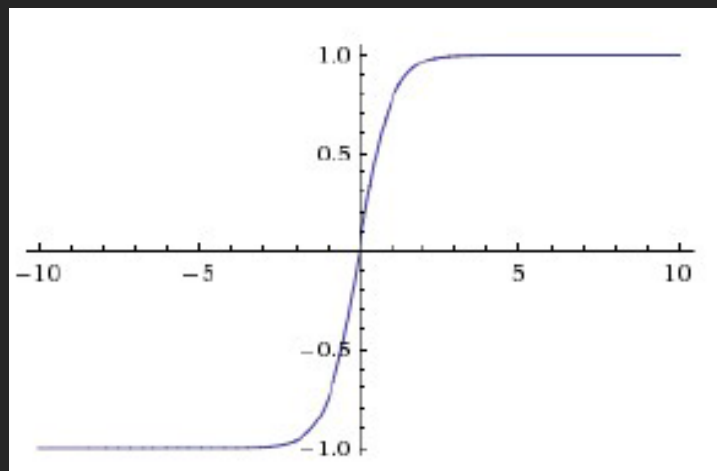


函数表达式和导函数：

$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = \frac{\alpha e^{-\alpha x}}{(1 + e^{-\alpha x})^2} = \alpha f(x)[1 - f(x)]$$

双曲正切函数 (tanh)



$\tanh(x) = 2\text{sigmoid}(2x) - 1$ ，但tanh 是0均值的。

函数表达式：

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

神经网络 · softmax

sigmoid、tanh 等将一个 real value 映射到 (0,1) 的区间 (也可以是 (-1,1))，这样可以用来做二分类。

当遇到多分类问题时，我们可以使用 softmax 函数，假设有一个数组 Z ， Z_j 表示 Z 中的第 j 个元素，那么这个

元素的 Softmax 值就是：
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \dots, K.$$
 它的优点是 定义符合要求，计算十分方便

在神经网络的计算当中，经常需要计算按照神经网络的正向传播计算的分数 S_1 ，和按照正确标注计算的分数 S_2 ，之间的差距，计算 Loss，才能应用反向传播。Loss 定义为交叉熵，
$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^j}\right)$$
 它占的比重越大，这个样本的 Loss 也就越小，这种定义符合我们的要求。

当我们对分类的 Loss 进行改进的时候，我们要通过梯度下降，每次优化一个 step 大小的梯度，这个时候我们就要求 Loss 对 **每个权重矩阵的偏导**，然后应用链式法则。那么这个过程的第一步，就是求 Loss 对 score 的偏

$$\frac{\partial L_i}{\partial f_{y_i}} = -\ln\left(\frac{e^{f_{y_i}}}{\sum_j e^j}\right)' = -(1 - P_{f_{y_i}}) = P_{f_{y_i}} - 1$$
 只要 将算出来的概率的向量对应的真正结果的那一维减1 就可以

举个例子，通过若干层的计算，得到的某个训练样本的向量是 $[1, 5, 3]$ ，那么概率分别就是 $[0.015, 0.866, 0.117]$ ，如果这个样本正确的分类是第二个的话，那么计算出来的偏导就是 $[0.015, 0.866 - 1, 0.117] = [0.015, -0.134, 0.117]$ ，然后再根据这个进行 back propagation 就可以了。

Softmax 回归 vs. 多个二元分类器

当类别数 $k = 2$ 时，softmax 回归退化为 logistic 回归。这表明 softmax 回归是 logistic 回归的一般形式。当所分的类别互斥时，更适于选择 softmax 回归分类器，当不互斥时，建立多个独立的 logistic 回归分类器更加合适。

Softmax 代码实现

`softmax(x)` 函数应该返回一个形状和x相同的NumPy array类型。例如，当输入为一个列表或者一维矩阵时：

```
scores = [1.0, 2.0, 3.0]
```

应该返回一个同样长度（即3个元素）的一维矩阵：

```
print softmax(scores)
```

```
[ 0.09003057  0.24472847  0.66524096 ]
```

对于一个二维矩阵，如以下（列向量表示单个样本），例如：

```
scores = np.array([[1, 2, 3, 6],  
                  [2, 4, 5, 6],  
                  [3, 8, 7, 6]])
```

该函数应该返回一个同样大小(3,4)的二维矩阵，如下：

```
[[ 0.09003057  0.00242826  0.01587624  0.33333333]  
 [ 0.24472847  0.01794253  0.11731043  0.33333333]  
 [ 0.66524096  0.97962921  0.86681333  0.33333333]]
```

每个样本（列向量）中的概率加起来应当等于 1

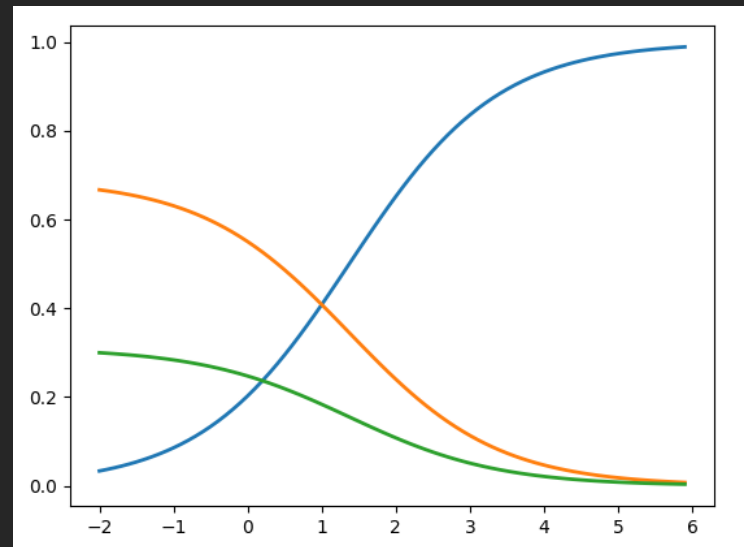
```
"""Softmax."""
```

```
scores = [3.0, 1.0, 0.2]  
import numpy as np
```

```
def softmax(x):  
    return np.exp(x)/np.sum(np.exp(x),axis=0)  
    """Compute softmax values for each sets of scores in x."""  
    pass # TODO: Compute and return softmax(x)
```

```
print(softmax(scores))  
# Plot softmax curves  
import matplotlib.pyplot as plt  
x = np.arange(-2.0, 6.0, 0.1)  
scores = np.vstack([x, np.ones_like(x), 0.2 * np.ones_like(x)])  
# vstack:Stack arrays in sequence vertically (row wise)  
plt.plot(x, softmax(scores).T, linewidth=2)  
plt.show()
```

运行结果：



神经网络 · 代价函数

神经网络模型的效果和优化的目标是通过**代价函数** (lost function) , 也称 **损失函数** 来定义的。代价函数常用均方差代价函数, 对于一个神经元 (单输入单输出, sigmoid函数) ,定义其代价函数为: $C = \frac{(y - a)^2}{2}$ 其中y是期望的输出, a为神经元的实际输出。

训练神经网络通过**反向传播代价**, 以减少代价为导向, 调整参数。参数主要有: 神经元之间的**连接权重w**, 以及每个神经元的偏置b。调参的方式是采用**梯度下降算法** (Gradient descent) , 沿着梯度方向调整参数大小。

w和b的梯度推导:
$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x$$
$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z)$$
 为sigmoid函数, 和b更新非常慢。

σ 表示激活函数, z 表示神经元的输入。神经网络模型常用的激活函数因为sigmoid函数的性质, 导致 $\sigma'(z)$ 在 z 取大部分值时会很小, 会使得w

sigmoid 函数 (还有tanh) 有很多优点, 非常适合用来做激活函数, 要解决这个问题, 就可以更改代价函数, 使用 **交叉熵** (cross entropy) 代价函数, $H(p, q) = - \sum_x p(x) \log q(x).$ 其中y为期望的输出, a为神经元实际输出。交叉熵刻画的是 **两个概率分布之间的距离**。交叉熵代价函数同样有两个性质: (1) **非负性**, 所以我们的目标就是最小化代价函数; (2) 当**真实输出a与期望输出y接近的时候**, **代价函数接近于0**。

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j(\sigma(z) - y).$$
$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y).$$

观察它的导数, 没有 $\sigma'(z)$ 这一项, 权重的更新是受 $\sigma(z)-y$ 这一项影响, 即受误差的影响。当误差大的时候, 权重更新就快, 当误差小的时候, 权重的更新就慢。这是很好的性质。

因为交叉熵一般会与 softmax 函数一起使用, 所以TensorFlow 对这两个功能进行了 **统一封装**, 并提供了 `tf.nn.softmax_cross_entropy_with_logits` 函数。

自定义代价函数

TensorFlow 不仅支持经典的代价函数，还可以优化任意的自定义代价函数。例如，在销售一个商品时，每个商品的成本为1元，利润是10元，那么少预测一个就少赚10元，多预测一个才赔1元，当时用均方误差时，神经网络模型很可能无法最大化预期利润。这时，就需要自定义代价函数：这里 $a = 10$, $b=1$; y_i 为第 i 个数据的正

确答案， y'_i 为预测值：
$$\text{Loss}(y, y') = \sum_{i=1}^n f(y_i, y'_i) \quad f(x, y) = \begin{cases} a(x - y) & x > y \\ b(y - x) & x \leq y \end{cases}$$

该代价函数代码：`loss = tf.reduce_sum(tf.select(tf.greater(v1, v2), (v1-v2)*a, (v2-v1)*b))`

`tf.greater` 的输入是两个张量，比较这两个输入张量的大小，并返回比较结果 (`True, False`)，维度不同时，会进行广播 (`broadcasting`) 操作。`tf.select` 函数有三个参数，第一个为选择条件依据，当为 `True` 时，会选择第2个参数中的值，否则使用第3个参数中的值。（注意：`window` 中用 `tf.where` 代替）

实例完整代码：

```
import tensorflow as tf
from numpy.random import RandomState
#定义神经网络的相关参数和变量
batch_size = 8
x = tf.placeholder(tf.float32, shape=(None, 2), name="x-input")
y_ = tf.placeholder(tf.float32, shape=(None, 1), name='y-input')
w1= tf.Variable(tf.random_normal([2, 1], stddev=1, seed=1))
y = tf.matmul(x, w1)
# 定义损失函数使得预测少了的损失大，于是模型应该偏向多的方向预测。
loss_less = 10
loss_more = 1
loss = tf.reduce_sum(tf.where(tf.greater(y, y_), (y - y_) * loss_more, (y_ - y) * loss_less))
train_step = tf.train.AdamOptimizer(0.001).minimize(loss)
```

自定义代价函数

```
#生成模拟数据集
rdm = RandomState(1)
X = rdm.rand(128,2)
Y = [[x1+x2+rdm.rand()/10.0-0.05] for (x1, x2) in X] #加入随机噪声
# 这里设置的训练目标 Y 为 x1+x2 +均值为0的随机噪声。
# 训练模型
with tf.Session() as sess:
    init_op = tf.global_variables_initializer()
    sess.run(init_op)
    STEPS = 5000
    for i in range(STEPS):
        start = (i*batch_size) % 128
        end = (i*batch_size) % 128 + batch_size
        sess.run(train_step, feed_dict={x: X[start:end], y_: Y[start:end]})
        if i % 1000 == 0:
            print("After %d training step(s), w1 is: " % (i))
            print (sess.run(w1), "\n")
    print ("Final w1 is: \n", sess.run(w1))
```

```
After 0 training step(s), w1 is:
[[-0.81031823]
 [ 1.4855988 ]]

After 1000 training step(s), w1 is:
[[ 0.01247112]
 [ 2.1385448 ]]

After 2000 training step(s), w1 is:
[[ 0.45567414]
 [ 2.17060661]]

After 3000 training step(s), w1 is:
[[ 0.69968724]
 [ 1.8465308 ]]

After 4000 training step(s), w1 is:
[[ 0.89886665]
 [ 1.29736018]]

Final w1 is:
[[ 1.01934695]
 [ 1.04280889]]
```

运行后得到的 W1 的值为 $1.02x_1 + 1.04x_2$ 比设定的 $x_1 + x_2$ 要大，这就是因为代价函数中 $\text{loss_less} > \text{loss_more}$ 。如果将 loss_less 的值设为1， loss_more 的值设为10，预测的 W1 结果将会是 $\begin{bmatrix} 0.95525807 \\ 0.9813394 \end{bmatrix}$ ，也就是说这样的设置下，模型会更加偏向预测的更小。

如果使用均方误差作为代价函数： $\text{loss} = \text{tf.contrib.losses.mean_squared_error}(y, y_)$ ，预测的 W1 结果将会是 $\begin{bmatrix} 0.97437561 \\ 1.0243336 \end{bmatrix}$ ，会尽量让预测值离标准答案更近，但不一定能使利益最大化。由此可见，不同的代价函数会对训练得到的模型产生重要影响。

神经网络 · 梯度下降

神经网络的优化算法可以分为两个阶段，第一阶段先通过前向传播算法计算得到预测值，并将预测值和真实值做对比，得出两者之间的差距。然后在第二阶段通过反向传播算法计算损失函数对每一个参数的梯度，再根据梯度和学习率使用梯度下降算法更新每一个参数。

只有当代价函数为凸函数时，梯度下降算法才能保证被优化的函数达到全局最优解。除此之外，梯度下降算法另一个问题就是计算时间过长，根据统计，如果一个操作需要 n 次浮点运算，那么计算这个梯度则需要三倍计算量。在海量训练数据下，要计算所有训练数据的损失函数是非常耗时间的，为了加速训练过程，可以使用随机梯度下降算法（Stochastic Gradient Descent, SGD），它优化的并不是在全部训练数据上的损失函数，而是在每一轮迭代中，随机优化某一条训练数据上的损失函数。这样每一轮参数更新的速度大大加快，也不会陷入局部最优。但某一条数据上损失函数更小并不代表在全部数据上损失函数更小，因此，使用随机梯度下降优化得到的神经网络需要更多的步数，有时甚至很难达到局部最优。

为了综合梯度下降算法和随机梯度下降算法的优缺点，在实际应用中一般采取两个算法的折中，每一计算一小部分训练数据（称为一个batch）的损失函数，也称 mini-batch 梯度下降，通常batch的取值为 2 到 100。

在随机梯度下降中，还可以引入冲量（Momentum），将上一步的梯度方向保留一部分在下一步中，它的思想是，若当前梯度方向与上一次相同，那么，此次的速度增强，否则，应该相应减弱。通常保留的系数在初始学习稳定之前，取0.5，之后取0.9或更大。

神经网络 · 学习率

神经网络的学习率决定了参数每次更新的幅度，如果学习率过小，需要更多轮的迭代才能达到一个比较理想的优化效果，如果过大，可能导致参数在极优值的两侧来回移动。TensorFlow提供了一种灵活的学习率设置方法——**指数衰减法**。

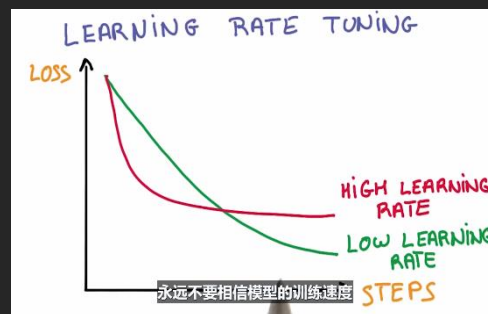
`tf.train.exponential_decay` (`learning_rate`, `global_step`, `decay_steps`, `decay_rate`, `staircase=False`, `name=None`)

通过这个函数，可以先使用较大的学习率来快速得到一个比较优的解，然后随着迭代的继续逐步减小学习率。

它实现了以下代码的功能：
$$\text{decayed_learning_rate} = \text{learning_rate} * \text{decay_rate} ^ (\text{global_step} / \text{decay_steps})$$

其中 `learning_rate` 为事先设定的初始学习率，`decay_rate` 为衰减系数（取值0到1），`decay_steps` 为衰减速度，`global_step` 为全局步数，系统会自动更新这个参数的值，从1开始，每迭代一次加1。这样随着迭代步数的增加，`decayed_learning_rate` 就会逐渐减小。除此之外，`staircase` 默认值为False，代表 `global_step / decay_steps` 为浮点数，梯度的下降是连续衰减的。当设置为 True 时，`global_step / decay_steps` 会被转化为整数，使得学习率成为一个**阶梯函数**（staircase function），这时，`decay_steps` 通常设置为完整训练一个 batch 所需要的迭代轮数，这样一个 batch 中的每个数据对模型训练有相等的作用，而每完整的训练一个 batch，学习率就减小一次。

一般来说，初始学习率、衰减系数和衰减速度都是根据经验设置的，而且代价函数下降的速度和迭代结束之后总代价的大小没有必然的联系，也就是说并不能通过前几轮代价函数下降的速度来比较不同神经网络的效果。



神经网络 · 学习率

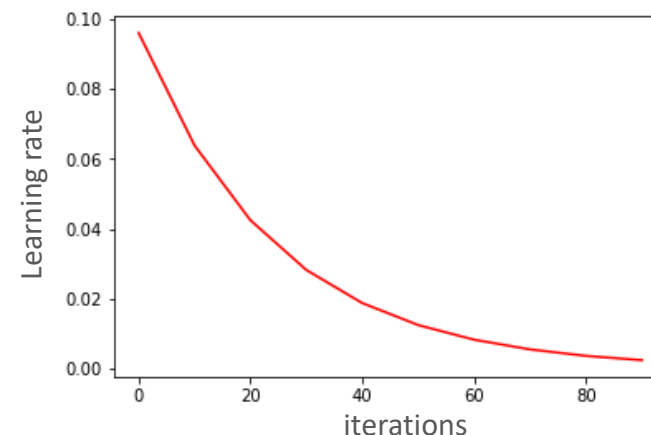
实例代码

```
TRAINING_STEPS = 100
global_step = tf.Variable(0)
LEARNING_RATE = tf.train.exponential_decay(0.1, global_step, 1, 0.96, staircase=True)

x = tf.Variable(tf.constant(5, dtype=tf.float32), name="x")
y = tf.square(x)
train_op = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(y, global_step=global_step)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(TRAINING_STEPS):
        sess.run(train_op)
        if i % 10 == 0:
            LEARNING_RATE_value = sess.run(LEARNING_RATE)
            x_value = sess.run(x)
            print "After %s iteration(s): x%s is %f, learning rate is %f."%(i+1, i+1, x_value, LEARNING_RATE_value)
```

```
After 1 iteration(s): x1 is 4.000000, learning rate is 0.096000.
After 11 iteration(s): x11 is 0.690561, learning rate is 0.063824.
After 21 iteration(s): x21 is 0.222583, learning rate is 0.042432.
After 31 iteration(s): x31 is 0.106405, learning rate is 0.028210.
After 41 iteration(s): x41 is 0.065548, learning rate is 0.018755.
After 51 iteration(s): x51 is 0.047625, learning rate is 0.012469.
After 61 iteration(s): x61 is 0.038558, learning rate is 0.008290.
After 71 iteration(s): x71 is 0.033523, learning rate is 0.005511.
After 81 iteration(s): x81 is 0.030553, learning rate is 0.003664.
After 91 iteration(s): x91 is 0.028727, learning rate is 0.002436.
```



神经网络 · 正则化

神经网络在训练数据不够多时，或者overtraining时，常常会导致**过拟合**（overfitting）。随着训练过程，网络在training data上的error渐渐减小，但是在验证集上的error却反而渐渐增大。为了避免过拟合问题，一个常用的方法就是**正则化**（Regularization），主要有 L1 正则化（L1-norm）和L2正则化（L2-norm）。

L1正则化在原始的代价函数后面加上一个L1正则化项，即所有权重w的**绝对值的和**，乘以 λ/n 。L1正则化也被称为“稀疏规则算子”（Lasso Regularization）

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|.$$

L2正则化在原始的代价函数后面加上的是 所有权重w的 **平方的和**，再乘以 $\lambda/2n$ （方便求导）。在回归里面，有人把有它的回归叫“岭回归”（Ridge Regression）

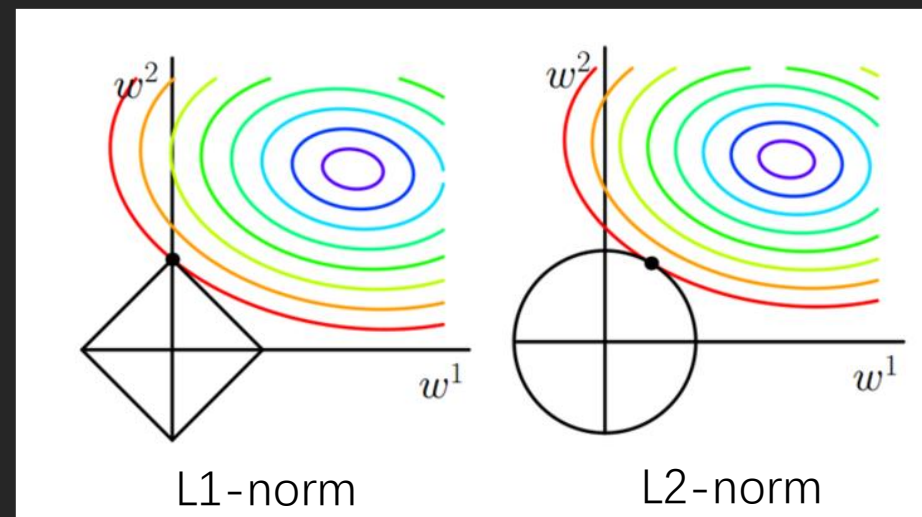
$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

一般都会在正则化项之前添加一个系数用 α 或 λ 表示，这个系数需要用户指定。

L1正则化产生的权值矩阵更加**稀疏**，可以用于**特征选择**。同时由于计算公式不可导，优化时更加复杂。

而L2正则化代价函数的优更加简洁，可以获得值很小的参数，能**更好的防止过拟合**，二维平面下L2正则化的函数图形是个圆，与方形相比，被磨去了棱角。因此J0与L相交时使得w1或w2等于零的机率小了许多，这就是为什么L2正则化**不具有稀疏性**的原因。

在实践中，也可以将它们同时使用。



TensorFlow 正则化

TensorFlow 提供了 `tf.contrib.layers.l1_regularizer` 函数，计算一个给定参数的L1正则化项的值（l1可以换成l2）

```
>>> import tensorflow as tf
>>> weights = tf.constant([[1.0, -2.0], [-3.0, 4.0]])
>>> with tf.Session() as sess:
    print(sess.run(tf.contrib.layers.l1_regularizer(.5)(weights)))
    print(sess.run(tf.contrib.layers.l2_regularizer(.5)(weights)))
```

运行结果：5.0 7.5

其中L2的计算自动除以2 以使求导更加简洁

加入正则项后，代价函数为：`loss = tf.reduce_mean(tf.square(y_ - y)) + tf.contrib.layers.l1_regularizer(lambda)(w)`

当网络的参数增多以后，这样的书写方式会导致 代价函数的定义很长，可读性差。除此之外，当网络结构复杂以后定义网络结构的部分和计算代价函数的部分可能不在同一个函数中，为了解决这个问题，可以使用TensorFlow中提供的 **集合（collection）** 来管理。collection提供了一种“零存整取”的思路：在任意位置，任意层次都可以创造对象，存入相应collection中；创造完成后，统一从一个collection中取出一类变量，施加相应操作

`tf.add_to_collection` 函数可以将资源加入一个或多个集合中，然后通过 `tf.get_collection` 获取一个集合里面的所有资源。

实例代码

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
data = []
label = []
np.random.seed(0)
```

```
# 以原点为圆心，半径为1的圆把散点划分成红蓝两部分，并加入随机噪音。
for i in range(150):
    x1 = np.random.uniform(-1,1)
    x2 = np.random.uniform(0,2)
    if x1**2 + x2**2 <= 1:
        data.append([np.random.normal(x1, 0.1), np.random.normal(x2, 0.1)])
        label.append(0)
    else:
        data.append([np.random.normal(x1, 0.1), np.random.normal(x2, 0.1)])
        label.append(1)
```

```
data = np.hstack(data).reshape(-1,2)
label = np.hstack(label).reshape(-1, 1)
plt.scatter(data[:,0], data[:,1], c=label,cmap="RdBu", vmin=-.2, vmax=1.2, edgecolor="white")
plt.show()    #绘制原始图形
```

#定义获取权重函数，并将正则项加入到损失的函数

```
def get_weight(shape, lambda1):
    var = tf.Variable(tf.random_normal(shape), dtype=tf.float32)
    tf.add_to_collection('losses', tf.contrib.layers.l2_regularizer(lambda1)(var))
    return var
```

定义神经网络

```
x = tf.placeholder(tf.float32, shape=(None, 2))
y_ = tf.placeholder(tf.float32, shape=(None, 1))
sample_size = len(data)
```

每层节点的个数

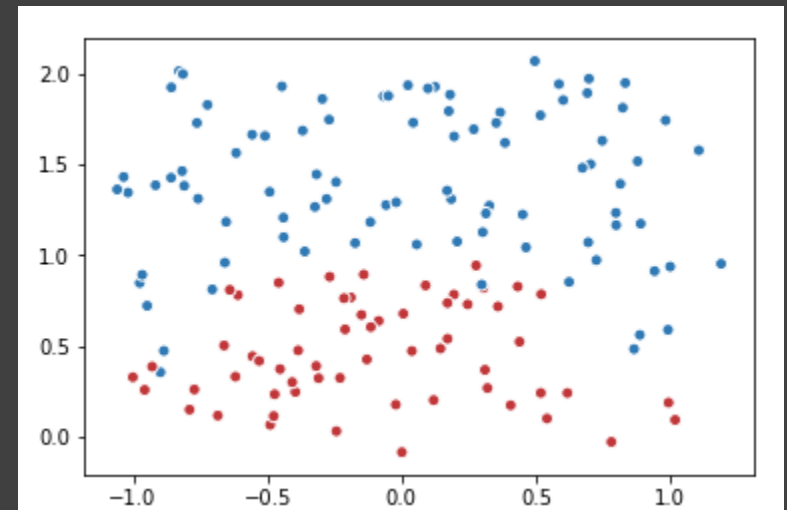
```
layer_dimension = [2,10,5,3,1]
n_layers = len(layer_dimension)
cur_layer = x
in_dimension = layer_dimension[0]
```

循环生成网络结构

```
for i in range(1, n_layers):
    out_dimension = layer_dimension[i]
    weight = get_weight([in_dimension, out_dimension], 0.003)
    bias = tf.Variable(tf.constant(0.1, shape=[out_dimension]))
    cur_layer = tf.nn.elu(tf.matmul(cur_layer, weight) + bias)
    in_dimension = layer_dimension[i]
y= cur_layer
```

损失函数的定义

```
mse_loss = tf.reduce_sum(tf.pow(y_ - y, 2)) / sample_size
tf.add_to_collection('losses', mse_loss)
loss = tf.add_n(tf.get_collection('losses'))
```



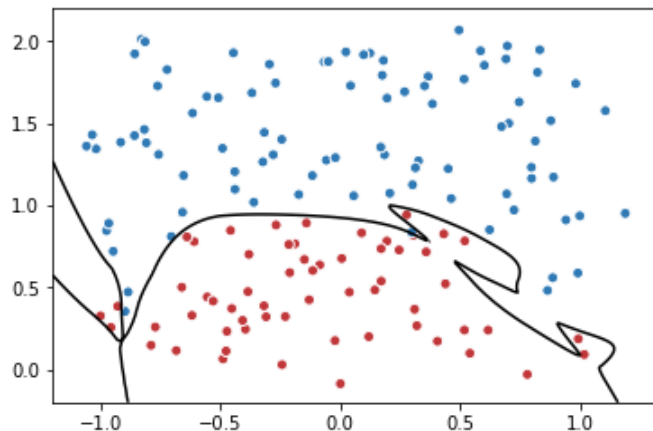
```
# 训练不带正则项的损失函数mse_loss
# 定义训练的目标函数mse_loss, 训练次数及训练模型
train_op = tf.train.AdamOptimizer(0.001).minimize(mse_loss)
TRAINING_STEPS = 40000
```

此处改为 loss训练带正则项的损失函数

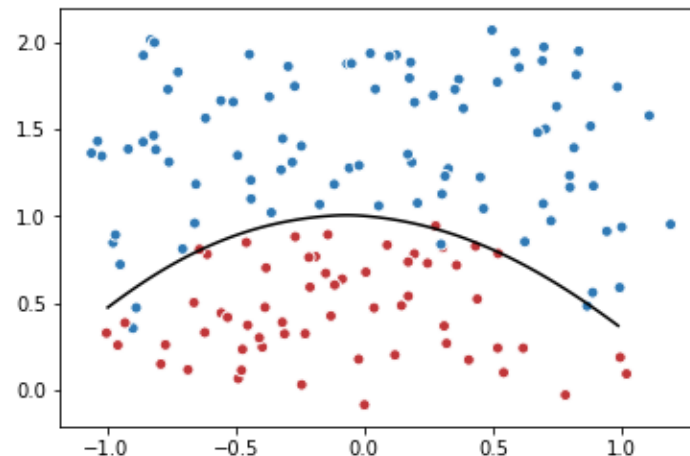
```
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for i in range(TRAINING_STEPS):
        sess.run(train_op, feed_dict={x: data, y_: label})
        if i % 2000 == 0:
            print("After %d steps, mse_loss: %f" % (i, sess.run(mse_loss, feed_dict={x: data, y_: label})))
# 画出训练后的分割曲线
xx, yy = np.mgrid[-1.2:1.2:.01, -0.2:2.2:.01]
grid = np.c_[xx.ravel(), yy.ravel()]
probs = sess.run(y, feed_dict={x: grid})
probs = probs.reshape(xx.shape)

plt.scatter(data[:,0], data[:,1], c=label, cmap="RdBu", vmin=-.2, vmax=1.2, edgecolor="white")
plt.contour(xx, yy, probs, levels=[.5], cmap="Greys", vmin=0, vmax=.1)
plt.show()
```

```
After 0 steps, mse_loss: 3.177866
After 2000 steps, mse_loss: 0.057275
After 4000 steps, mse_loss: 0.038920
After 6000 steps, mse_loss: 0.031493
After 8000 steps, mse_loss: 0.026564
After 10000 steps, mse_loss: 0.023217
After 12000 steps, mse_loss: 0.022093
After 14000 steps, mse_loss: 0.019196
After 16000 steps, mse_loss: 0.016351
After 18000 steps, mse_loss: 0.013316
After 20000 steps, mse_loss: 0.009571
After 22000 steps, mse_loss: 0.007185
After 24000 steps, mse_loss: 0.005867
After 26000 steps, mse_loss: 0.005189
After 28000 steps, mse_loss: 0.004776
After 30000 steps, mse_loss: 0.004467
After 32000 steps, mse_loss: 0.004204
After 34000 steps, mse_loss: 0.003976
After 36000 steps, mse_loss: 0.003775
After 38000 steps, mse_loss: 0.003559
```

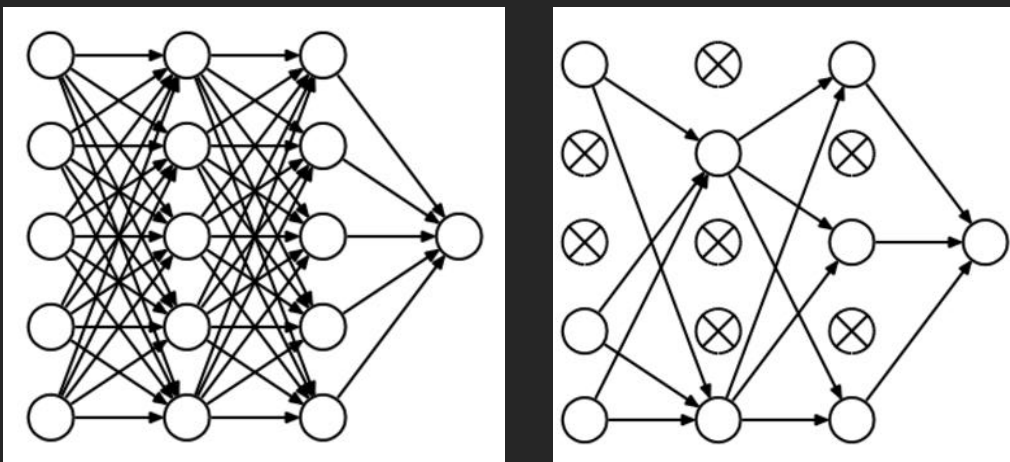


```
After 0 steps, loss: 1.018758
After 2000 steps, loss: 0.098835
After 4000 steps, loss: 0.066563
After 6000 steps, loss: 0.060748
After 8000 steps, loss: 0.059121
After 10000 steps, loss: 0.055136
After 12000 steps, loss: 0.054979
After 14000 steps, loss: 0.054980
After 16000 steps, loss: 0.054979
After 18000 steps, loss: 0.054979
After 20000 steps, loss: 0.054979
After 22000 steps, loss: 0.054979
After 24000 steps, loss: 0.054979
After 26000 steps, loss: 0.054980
After 28000 steps, loss: 0.054979
After 30000 steps, loss: 0.054979
After 32000 steps, loss: 0.054979
After 34000 steps, loss: 0.054979
After 36000 steps, loss: 0.054981
After 38000 steps, loss: 0.054979
```



dropout

L1、L2正则化是通过修改代价函数来实现防止过拟合的，而 Dropout 则是通过修改神经网络本身来实现的，它是在训练网络时用的一种技巧（trick）。在模型训练时按照一定的概率暂时让网络某些隐含层节点的权重不工作（也称丢弃），不工作的那些节点可以暂时认为不是网络结构的一部分，但是它的权重得保留下来，对于神经网络单元，按照一定的概率将其暂时从网络中丢弃。对于随机梯度下降来说，由于是随机丢弃，故而每一个mini-batch都在训练不同的网络。这样阻止特征检测器的共同作用，减弱了神经元节点间的联合适应性，增强了泛化能力，提高神经网络的性能。



TensorFlow中，可以用一个placeholder来代表一个神经元的输出在 dropout 中保持不变的概率。这样我们可以在训练过程中启用dropout，在测试过程中关闭dropout。TensorFlow的 `tf.nn.dropout` 操作除了可以屏蔽神经元的输出外，还会自动处理神经元输出值的 scale。所以用 dropout 的时候可以不用考虑 scale。

实例代码（片段）

```
..... # MNIST部分代码
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
.....
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

滑动平均模型

除了防止过拟合的方法之外，**滑动平均模型** 也能使模型在测试数据上更加健壮。TensorFlow 提供了 `tf.train.ExponentialMovingAverage` 函数来实现，用户需要提供一个衰减率（decay）来控制模型更新的速度，它的值越大模型越趋于稳定。实际运用中，decay 一般会设置为十分接近 1 的常数（0.99或0.999）。为了使得模型在训练的初始阶段更新得更快，ExponentialMovingAverage 还提供了 num_updates 参数（可使用神经网络的迭代轮数）来动态设置 decay 的大小：

$$\text{decay} = \min \left\{ \text{decay}, \frac{1 + \text{num_updates}}{10 + \text{num_updates}} \right\}$$

ExponentialMovingAverage 对每一个变量（variable）都会维护一个**影子变量（shadow variable）**。影子变量的初始值就是这个变量的初始值，而每次运行变量更新时，影子变量的值会更新为：

$$\text{shadow_variable} = \text{decay} \times \text{shadow_variable} + (1 - \text{decay}) \times \text{variable}$$

实例代码

```
import tensorflow as tf
v1 = tf.Variable(0, dtype=tf.float32) #初始值设为0
step = tf.Variable(0, trainable=False)
# step 模拟神经网络中的迭代轮数，来动态控制衰减率
ema = tf.train.ExponentialMovingAverage(0.99, step)
maintain_averages_op = ema.apply([v1])
# 定义一个更新滑动平均的操作
with tf.Session() as sess:
    # 初始化
    init_op = tf.global_variables_initializer()
    sess.run(init_op)
    print(sess.run([v1, ema.average(v1)]))
```

```
# 更新变量v1的取值
sess.run(tf.assign(v1, 5)) # 更新 v1 的值为5
sess.run(maintain_averages_op)
print (sess.run([v1, ema.average(v1)]))
# 通过ema.average(v1)获得滑动平均之后变量的取值
# 更新step和v1的取值
sess.run(tf.assign(step, 10000)) # 更新step的值为10000
sess.run(tf.assign(v1, 10)) #更新 v1 的值为 10
sess.run(maintain_averages_op)
print (sess.run([v1, ema.average(v1)]))
# 当前 v1 的值和 step 值保持不变，更新一次v1的滑动平均值
sess.run(maintain_averages_op)
print(sess.run([v1, ema.average(v1)]))
```

运行结果：
[0.0, 0.0]
[5.0, 4.5]
[10.0, 4.5549998]
[10.0, 4.6094499]

One-Hot 编码

One-Hot 编码，又称独热编码或一位有效编码。对于每一个特征，如果它有m个可能值，那么经过One-Hot编码后，就变成了m个二元特征，并且这些特征互斥，**每次只有一个激活**。因此，数据会变成稀疏的。使用one-hot编码来表示离散特征，将离散特征的取值扩展到了欧式空间，**距离计算更合理**。

例如自然状态码为：000,001,010,011,100,101

则One-Hot 编码为：000001,000010,000100,001000,010000,100000

例如，有一个离散型特征，代表工作类型，该离散型特征共有三个取值。不使用one-hot编码，其表示分别是 $x_1 = (1)$ ， $x_2 = (2)$ ， $x_3 = (3)$ 。两个工作之间的距离是， $(x_1, x_2) = 1$ ， $d(x_2, x_3) = 1$ ， $d(x_1, x_3) = 2$ 。那么 x_1 和 x_3 工作之间就越不相似吗？显然这样的表示，计算出来的特征的距离是不合理。那如果使用one-hot编码，则得到 $x_1 = (1, 0, 0)$ ， $x_2 = (0, 1, 0)$ ， $x_3 = (0, 0, 1)$ ，那么两个工作之间的距离就都是 $\sqrt{2}$ 。即每两个工作之间的距离是一样的，显得更合理。

Scikit-learn实例

```
>>> from sklearn import preprocessing
>>> enc = preprocessing.OneHotEncoder()
>>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
>>> enc.transform([[0, 1, 3]]).toarray()
array([[ 1.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  1.]])
```

TensorFlow 实例 (MNIST)

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

#A one-hot vector is a vector which is 0 in most dimensions, and 1 in a single dimension.
#For example, 3 would be [0,0,0,1,0,0,0,0,0].
#Consequently, mnist.train.labels is a [55000, 10] array of floats.
```


自编码器 (AutoEncoding)

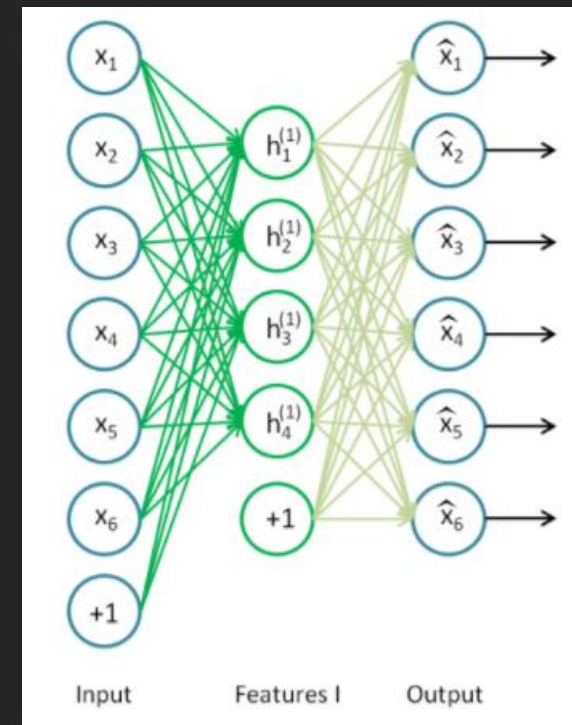
传统机器学习任务很大程度上依赖好的特征工程，而特征工程往往是非常耗时耗力的，尤其在视频、图像、语音等领域，深度学习则可以解决人工难以提取有效特征的问题。

深度学习在早期一度被理解作为一种无监督的特征学习，模仿人脑对特征逐层抽象提取的过程。一是无监督学习：无需对数据标注，对数据内容的组织形式的学习，提取的是频繁出现的特征；二是逐层抽象，从简单的微观特征开始，不断抽象特征的层级，逐渐往复杂的宏观特征转变。

早年人们研究稀疏编码 (sparse coding) 时，发现，几乎所有图像碎片 (16像素*16像素) 都可以有64种正交边组合得到，而且需要的边的数量是很少的 (稀疏的)，在声音领域，绝大多数声音都可以由20种基本声音结构线性组合得到。

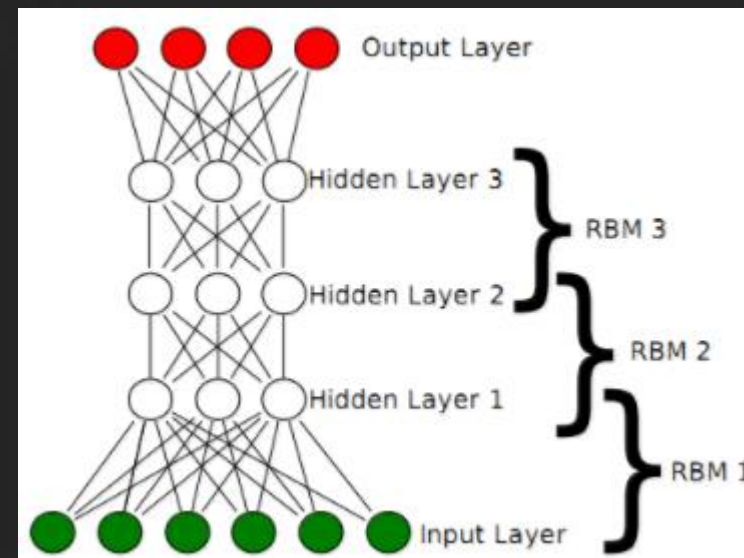
自编码器 (AutoEncoding) 就是借助稀疏编码的思想，使用稀疏的一些高阶特征来重构自己，它的特征：第一，期望输入\输出一致；第二，希望使用高阶特征重构自己。所以可以加入几种限制：

- (1) 限制中间隐含层的数量，让隐含层节点数量小于输入输出的节点数量，再给中间隐含层的权重加上正则项，根据惩罚系数控制隐含节点的系数程度，惩罚系数越大特征越稀疏。
- (2) 给数据加入噪声，称为 (Denoising AutoEncoder)，最常见的是加入加性高斯噪声 (Additive Gaussian Noise) 或随机遮挡噪声 (Masking Noise)。



深度信念网络 (Deep Belief Networks)

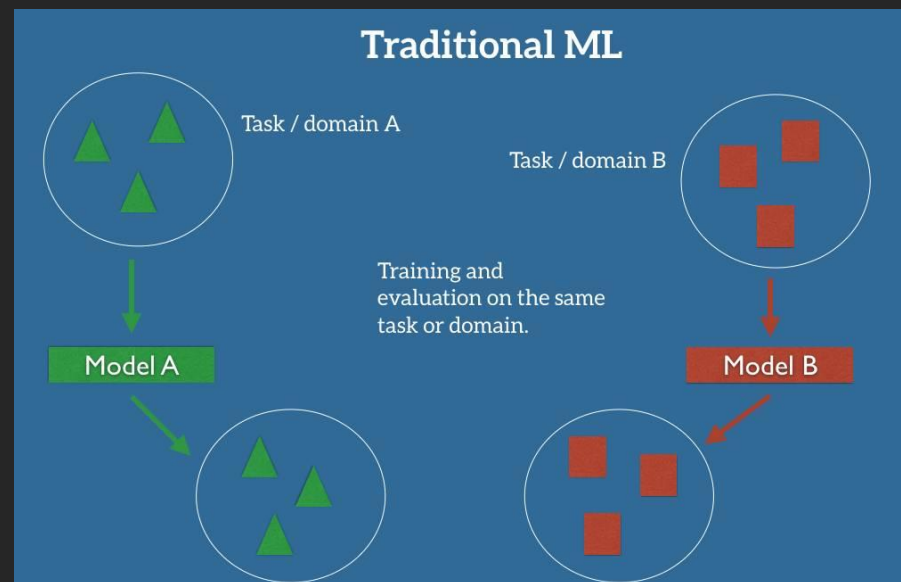
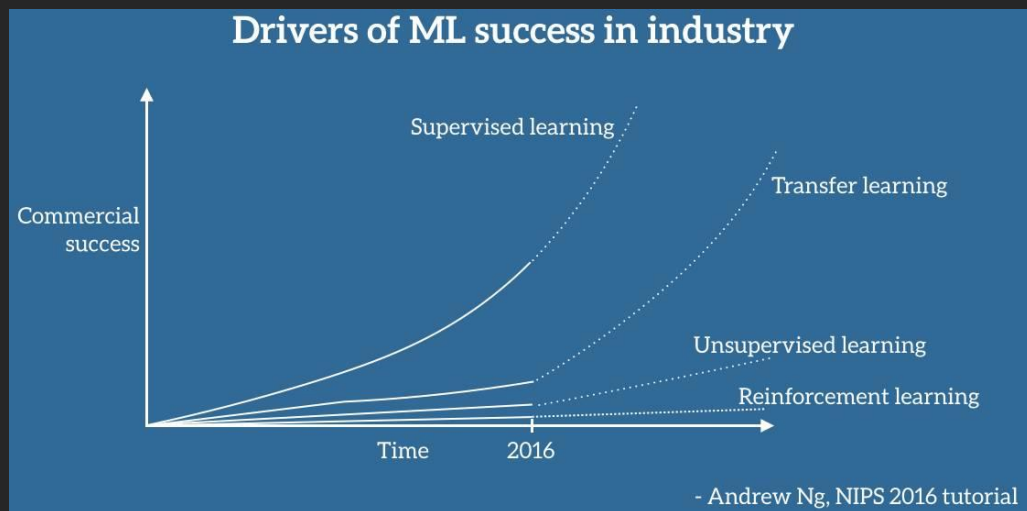
Hinton 教授在 *Science* 上发表文章 *Reducing the dimensionality of data with neural networks*, 讲解了使用自编码器对数据进行降维的方法, 同时提出了基于深度信念网络 (Deep Belief Networks) 可使用无监督的逐层训练的贪心算法。思路就是先用自编码器的方法进行无监督的预训练, 提取特征并初始化权重, 然后使用标注信息进行监督式的训练。



如果自编码器的隐含层只有一层, 其原理就类似于主成分分析 (PCA), Hinton提出的DBN模型有多个隐含层, 每个隐含层都是限制性玻尔兹曼机 (Restricted Boltzman Machine, RBM)。DBN训练时, 首先对每两层之间进行无监督的预训练 (相当于一个多层自编码器), 可以将整个网络的权重初始化到一个理想的分布, 最后通过反向传播算法调整模型权重, 这个步骤会使用经过标注的信息来做监督性的分类训练、解决了网络过深带来的梯度弥散 (Gradient Vanishment) 问题。

迁移学习

在传统的机器学习的框架下，学习的任务就是在给定充分训练数据的基础上来学习一个分类模型；然后利用这个学习到的模型来对测试文档进行分类与预测。然而一些新出现的领域中的大量训练数据非常难得到，传统的机器学习需要对每个领域都标定大量训练数据，这将会耗费大量的人力与物力。其次，传统的机器学习假设训练数据与测试数据服从相同的数据分布，然而在许多情况下，这种同分布假设并不满足。从另外一个角度上看，如果我们有了大量的、在不同分布下的训练数据，完全丢弃这些数据也是非常浪费的。如何合理的利用这些数据就是迁移学习主要解决的问题。迁移学习（Transfer Learning）可以从现有的数据中迁移知识，将从一个环境训练出的模型通过调整使其适用于一个新的问题，因此迁移学习不会像传统机器学习那样作同分布假设。斯坦福的教授吴恩达（Andrew Ng）在广受流传的 2016 年 NIPS 会议的教程中曾经说过：「迁移学习将会是继监督学习之后的下一个机器学习商业成功的驱动力」。



MNIST 数字识别实例

```
import tensorflow as tf
#Windows 10 python 3.6 TensorFlow 1.2.1
#定义神经网络结构相关的参数
INPUT_NODE = 784      #输入层节点数
OUTPUT_NODE = 10      #输出层节点数
LAYER1_NODE = 500     #隐藏层节点数

#通过tf.get_variable函数来获取变量
def get_weight_variable(shape, regularizer):
    weights = tf.get_variable("weights", shape, initializer=tf.truncated_normal_initializer(stddev=0.1))
    if regularizer != None: #正则项
        tf.add_to_collection('losses', regularizer(weights))
    return weights

#定义神经网络的前向传播过程
def inference(input_tensor, regularizer):
    with tf.variable_scope('layer1'):

        weights = get_weight_variable([INPUT_NODE, LAYER1_NODE], regularizer)
        biases = tf.get_variable("biases", [LAYER1_NODE], initializer=tf.constant_initializer(0.0))
        layer1 = tf.nn.relu(tf.matmul(input_tensor, weights) + biases)

    with tf.variable_scope('layer2'):
        weights = get_weight_variable([LAYER1_NODE, OUTPUT_NODE], regularizer)
        biases = tf.get_variable("biases", [OUTPUT_NODE], initializer=tf.constant_initializer(0.0))
        layer2 = tf.matmul(layer1, weights) + biases

    return layer2
```

定义神经网络中的参数和前向传播过程

MNIST 数字识别实例

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import mnist_inference
import os

# 定义神经网络结构相关的参数
BATCH_SIZE = 100
LEARNING_RATE_BASE = 0.8      #基础学习率
LEARNING_RATE_DECAY = 0.99    #学习率衰减率
REGULARIZATION_RATE = 0.0001  #正则化项在代价函数中的系数
TRAINING_STEPS = 30000        #训练轮数
MOVING_AVERAGE_DECAY = 0.99   #滑动平均衰减率
MODEL_SAVE_PATH = "MNIST_model/"
MODEL_NAME = "mnist_model"

def train(mnist):
    # 定义输入输出placeholder
    x = tf.placeholder(tf.float32, [None, mnist_inference.INPUT_NODE], name='x-input')
    y_ = tf.placeholder(tf.float32, [None, mnist_inference.OUTPUT_NODE], name='y-input')

    regularizer = tf.contrib.layers.l2_regularizer(REGULARIZATION_RATE)
    y = mnist_inference.inference(x, regularizer)
    global_step = tf.Variable(0, trainable=False)

    # 定义代价函数、学习率、滑动平均操作以及训练过程
    variable_averages = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY, global_step)
    variables_averages_op = variable_averages.apply(tf.trainable_variables())
    # 定义一个更新滑动平均的操作
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_)
    cross_entropy_mean = tf.reduce_mean(cross_entropy)
    loss = cross_entropy_mean + tf.add_n(tf.get_collection('losses'))
```

定义神经网络的训练过程

训练结果

```
After 1 training step(s), loss on training batch is 2.93368.
After 1001 training step(s), loss on training batch is 0.238723.
After 2001 training step(s), loss on training batch is 0.170218.
After 3001 training step(s), loss on training batch is 0.156278.
After 4001 training step(s), loss on training batch is 0.116478.
After 5001 training step(s), loss on training batch is 0.104817.
After 6001 training step(s), loss on training batch is 0.102203.
After 7001 training step(s), loss on training batch is 0.0890646.
After 8001 training step(s), loss on training batch is 0.0852515.
After 9001 training step(s), loss on training batch is 0.0748403.
After 10001 training step(s), loss on training batch is 0.0748737.
After 11001 training step(s), loss on training batch is 0.0618936.
After 12001 training step(s), loss on training batch is 0.0644133.
After 13001 training step(s), loss on training batch is 0.0574872.
After 14001 training step(s), loss on training batch is 0.0486513.
After 15001 training step(s), loss on training batch is 0.0491409.
After 16001 training step(s), loss on training batch is 0.0477048.
After 17001 training step(s), loss on training batch is 0.0535292.
After 18001 training step(s), loss on training batch is 0.0507448.
After 19001 training step(s), loss on training batch is 0.0459789.
After 20001 training step(s), loss on training batch is 0.0432968.
After 21001 training step(s), loss on training batch is 0.042615.
After 22001 training step(s), loss on training batch is 0.0445021.
After 23001 training step(s), loss on training batch is 0.0410245.
After 24001 training step(s), loss on training batch is 0.0367691.
After 25001 training step(s), loss on training batch is 0.0383737.
After 26001 training step(s), loss on training batch is 0.0341267.
After 27001 training step(s), loss on training batch is 0.0337505.
After 28001 training step(s), loss on training batch is 0.0333108.
After 29001 training step(s), loss on training batch is 0.0344076.
```


MNIST 数字识别实例

定义神经网络的训练过程（续）

```
learning_rate = tf.train.exponential_decay(
    LEARNING_RATE_BASE,
    global_step,
    mnist.train.num_examples / BATCH_SIZE, LEARNING_RATE_DECAY,
    staircase=True)
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss, global_step=global_step)
with tf.control_dependencies([train_step, variables_averages_op]):
    train_op = tf.no_op(name='train')

# 初始化TensorFlow持久化类
saver = tf.train.Saver()
with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(TRAINING_STEPS):
        xs, ys = mnist.train.next_batch(BATCH_SIZE)
        _, loss_value, step = sess.run([train_op, loss, global_step], feed_dict={x: xs, y_: ys})
        if i % 1000 == 0:
            print("After %d training step(s), loss on training batch is %g." % (step, loss_value))
            saver.save(sess, os.path.join(MODEL_SAVE_PATH, MODEL_NAME), global_step=global_step)

#主程序入口
def main(argv=None):
    mnist = input_data.read_data_sets("../../../datasets/MNIST_data", one_hot=True)
    train(mnist)

if __name__ == '__main__':
    main()
```

MNIST 数字识别实例

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import mnist_inference
import mnist_train
def evaluate(mnist):
    with tf.Graph().as_default() as g:
        x = tf.placeholder(tf.float32, [None, mnist_inference.INPUT_NODE], name='x-input')
        y_ = tf.placeholder(tf.float32, [None, mnist_inference.OUTPUT_NODE], name='y-input')
        validate_feed = {x: mnist.validation.images, y_: mnist.validation.labels}
        y = mnist_inference.inference(x, None)
        correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
        variable_averages = tf.train.ExponentialMovingAverage(mnist_train.MOVING_AVERAGE_DECAY)
        variables_to_restore = variable_averages.variables_to_restore()
        saver = tf.train.Saver(variables_to_restore)
        with tf.Session() as sess:
            ckpt = tf.train.get_checkpoint_state(mnist_train.MODEL_SAVE_PATH)
            if ckpt and ckpt.model_checkpoint_path:
                saver.restore(sess, ckpt.model_checkpoint_path)
                global_step = ckpt.model_checkpoint_path.split('/')[-1].split('-')[-1]
                accuracy_score = sess.run(accuracy, feed_dict=validate_feed)
                print("After %s training step(s), validation accuracy = %g" % (global_step, accuracy_score))
            else:
                print('No checkpoint file found')
    return

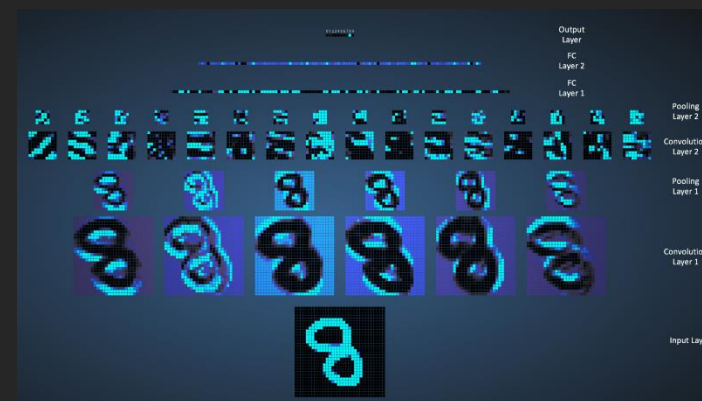
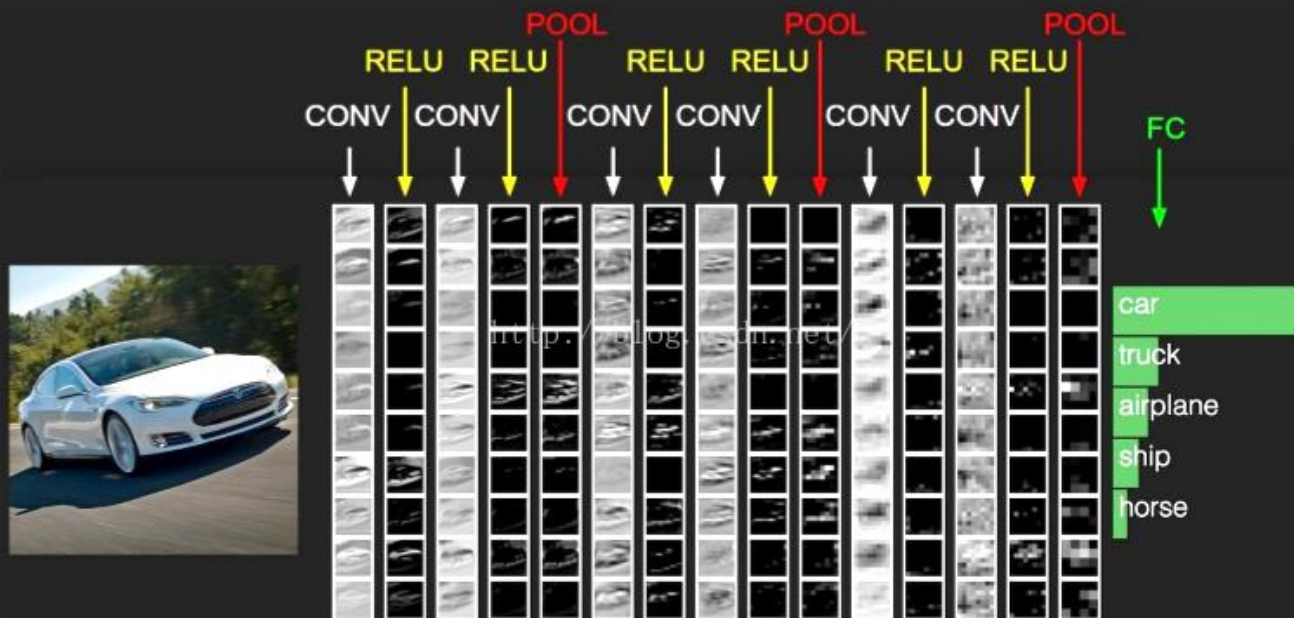
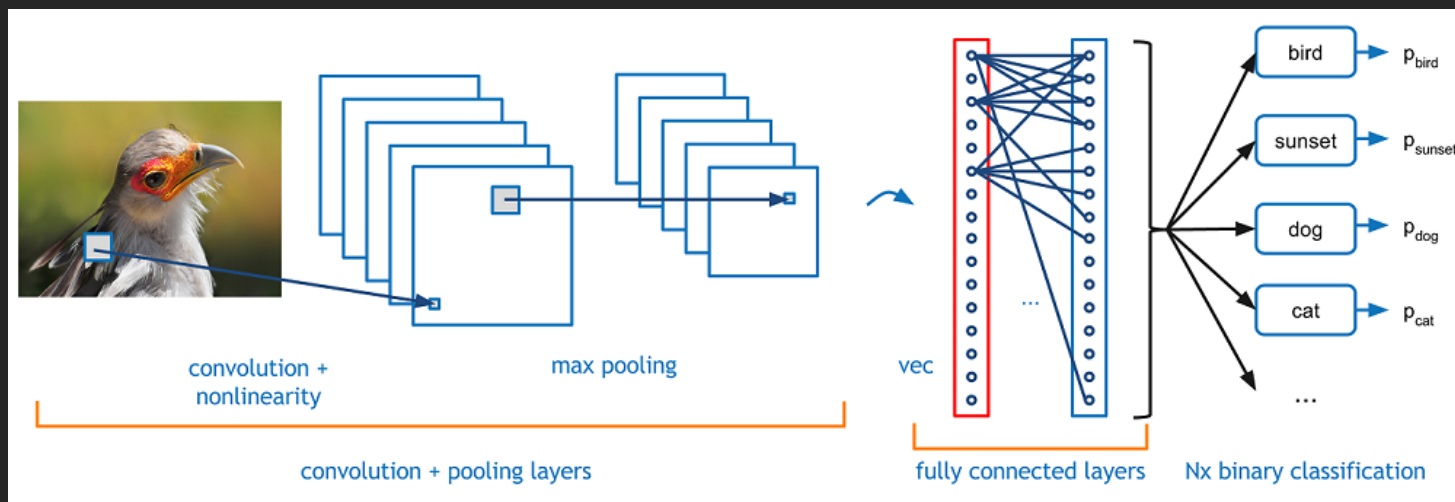
#主程序入口
def main(argv=None):
    mnist = input_data.read_data_sets("../../../datasets/MNIST_data", one_hot=True)
    evaluate(mnist)
if __name__ == '__main__':
    main()
```

定义神经网络的测试过程

测试结果 : INFO:tensorflow:Restoring parameters from MNIST_model/mnist_model-29001
After 29001 training step(s), validation accuracy = 0.985

卷积神经网络

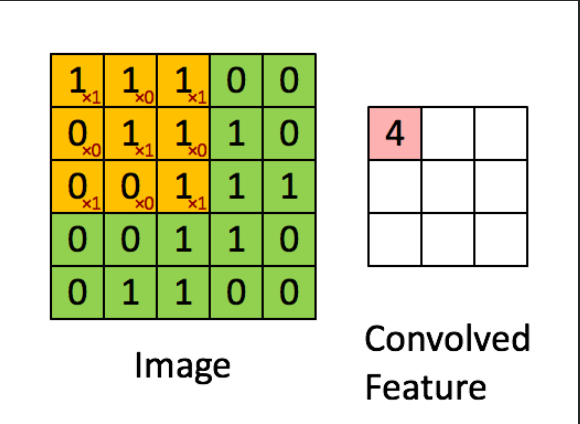
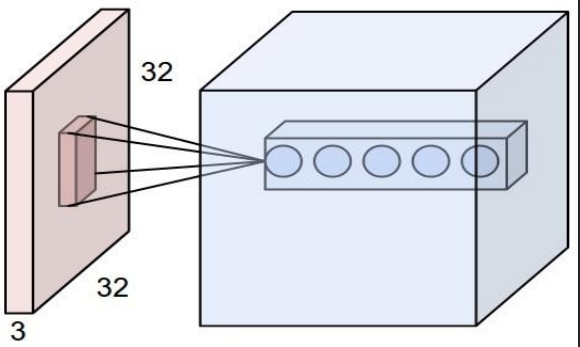
卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元。卷积神经网络由三部分构成。第一部分是输入层。第二部分由n个卷积层和池化层的组合组成。第三部分由一个全连结的多层感知机分类器构成。这一结构使得卷积神经网络能够利用输入数据的二维结构。卷积神经网络，就是会自动的对于一张图片学习出最好的卷积核以及这些卷积核的组合方式，然后来进行判断。



<http://scs.ryerson.ca/~aharley/vis/conv/>

卷积层

卷积神经网络的名字就来自于其中的卷积操作。卷积的主要目的是为了从输入图像中提取特征。卷积可以通过从输入的一小块数据中学到图像的特征，并可以保留像素间的空间关系。和传统的全连接层不同，卷积层中的每一个节点的输入只是上一层神经网络的一小块，这个小块常用的大小有 3×3 或者 5×5 。卷积层试图将神经网络中的每一小块进行更加深入地分析从而得到抽象程度更高的特征。一般来说，通过卷积层处理过的节点矩阵会变得更深，如下图，这个结构被称为卷积核（Kernel）或过滤器（Filter），过滤器可以将当前神经网络上的一个子节点矩阵转化为下一层神经网络上的一个单位节点矩阵（长宽为1，深度不限）。



在一个卷积层中，过滤器所处理的节点矩阵的长和宽都是由人工指定的，过滤器处理的矩阵深度和当前层神经网络节点矩阵的深度一致，所以过滤器的尺寸指的是输入节点矩阵的大小，而深度指的是输出单位节点的深度。将一个 $2 \times 2 \times 3$ 的节点矩阵转化为一个 $1 \times 1 \times 5$ 的单位节点矩阵，总共需要 $2 \times 2 \times 3 \times 5 + 5 = 65$ 个参数，其中最后5个为偏置项参数。当过滤器的大小不为 1×1 时，可以在当前层矩阵的边界上填充（padding），TensorFlow 提供 SAME 和 VALID 两种选择，其中 SAME 表示添加全 0 填充【过滤器某一维度的尺寸为偶数时，只填充该维度一侧(下，右)】，VALID 表示不添加填充。同时，还可以通过设置过滤器移动的步长来调节矩阵的大小，当不使用填充时，矩阵大小为：

$$\text{out_height} = \text{ceil}(\text{float}(\text{in_height} - \text{filter_height} + 1) / \text{float}(\text{strides}[1]))$$
$$\text{out_width} = \text{ceil}(\text{float}(\text{in_width} - \text{filter_width} + 1) / \text{float}(\text{strides}[2]))$$

卷积层

在卷积神经网络中，每一个卷积层中使用的过滤器的参数都是一样的，这是卷积神经网络的一个重要性质，从直观上来理解，**共享过滤器参数**可以使得图像上内容不受位置的影响。共享每个卷积层中过滤器的参数，可以巨幅减少神经网络上的参数，假设输入层矩阵的维度是 $32 \times 32 \times 3$ ，第一层卷积层使用尺寸为 5×5 ，深度为16的过滤器，那么这个卷积层的参数个数为 $5 \times 5 \times 3 \times 16 + 16 = 1216$ 个。

```
import tensorflow as tf
import numpy as np
M = np.array([
    [[1],[-1],[0]],
    [[-1],[2],[1]],
    [[0],[2],[-2]]
])
filter_weight = tf.get_variable('weights', [2, 2, 1, 1], initializer = tf.constant_initializer([
    [1, -1],
    [0, 2]]))

#前面两个维度为过滤器尺寸，第三个维度表示当前层的深度，第四个维度表示过滤器的深度
biases = tf.get_variable('biases', [1], initializer = tf.constant_initializer(1)) #[1]为过滤器深度
M = np.asarray(M, dtype='float32')
M = M.reshape(1, 3, 3, 1) #第一维对应一个输入batch，比如输入层中，input[0,:, :, :]表示第一张图片后三个维度对应一个节点矩阵
x = tf.placeholder('float32', [1, None, None, 1])
conv = tf.nn.conv2d(x, filter_weight, strides = [1, 2, 2, 1], padding = 'SAME') #第1个参数对应一个输入batch；第2个参数提供卷积层的权重；第3个参数为不同维度上的步长，其中第一维和最后一维一定为1，因为卷积层的步长只对矩阵的长和宽有效；最后一个为填充。
bias = tf.nn.bias_add(conv, biases)
activated_conv = tf.nn.relu(bias) #将计算结果通过ReLU激活函数去线性化
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    convoluted_M = sess.run(bias, feed_dict={x:M})
    print("convoluted_M: \n", convoluted_M)
```

convoluted_M:
[[[[7.]
[1.]]

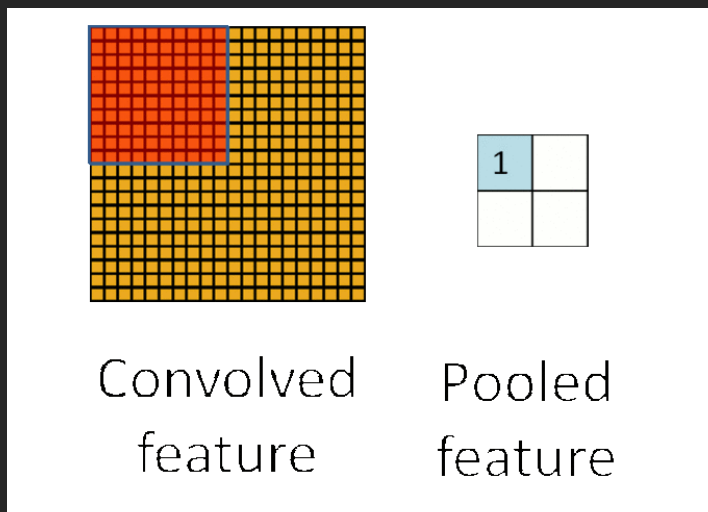
[[-1.]
[-1.]]]]

运行结果：

池化层

空间池化（Spatial Pooling）降低了各个特征图的维度，但可以保持大部分重要的信息。空间池化有下面几种方式：最大化、平均化、加和等等。对于**最大池化（Max Pooling）**，我们定义一个空间邻域（比如，2x2的窗口），并从窗口内的修正特征图中取出最大的元素。除了最大池化，也可以用平均值（Average Pooling）或者对窗口内的元素求和。在实际中，最大池化被证明效果更好一些。

池化函数可以逐渐降低输入表示的空间尺度。使输入表示（特征维度）变得更小，并且网络中的参数和计算的数量更加可控的减小，因此可以**控制过拟合**；同时使网络对于输入图像中更小的变化、冗余和变换变得**不变性**（输入的微小冗余将不会改变池化的输出——因为我们在局部邻域中使用了最大化/平均值的操作，这使得我们可以检测图像中的物体，无论它们位置在哪里。



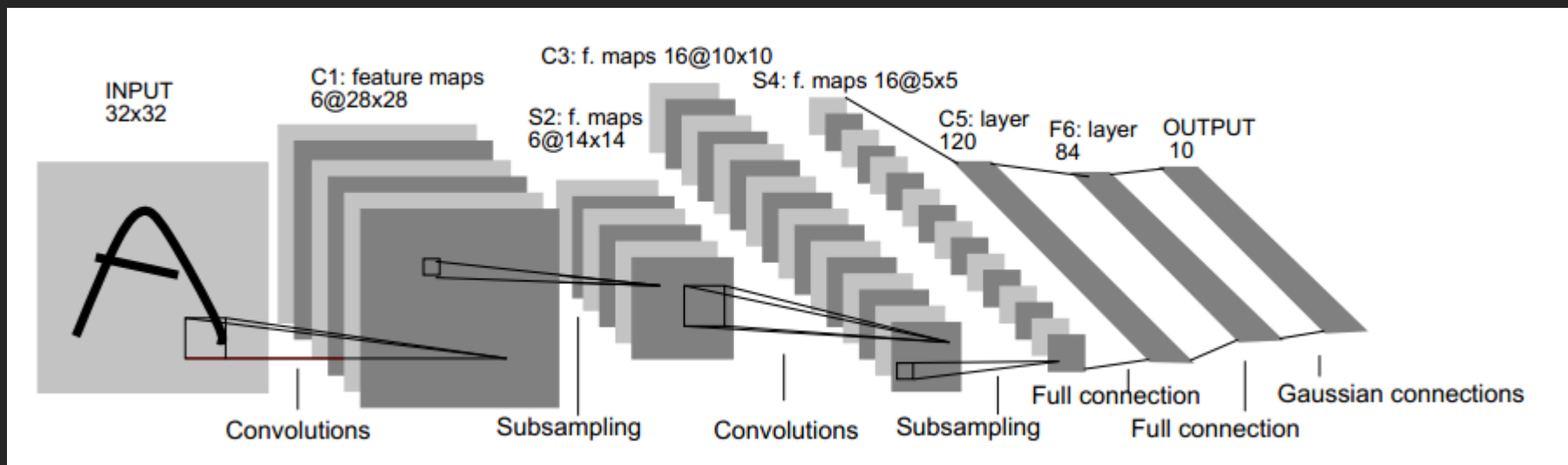
```
import tensorflow as tf
import numpy as np
M = np.array([
    [[1],[-1],[0]],
    [[-1],[2],[1]],
    [[0],[2],[-2]]
])
M = np.asarray(M, dtype='float32')
M = M.reshape(1, 3, 3, 1)
x = tf.placeholder('float32', [1, None, None, 1])
pool = tf.nn.avg_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
# 参数和卷积函数类似 最大池化: tf.nn.max_pool
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    pooled_M = sess.run(pool, feed_dict={x:M})
    print("pooled_M: \n", pooled_M)
```

```
pooled_M:
[[[ 0.25]
  [ 0.5 ]]]

[[ 1. ]
 [-2. ]]]
```

LeNet-5 模型

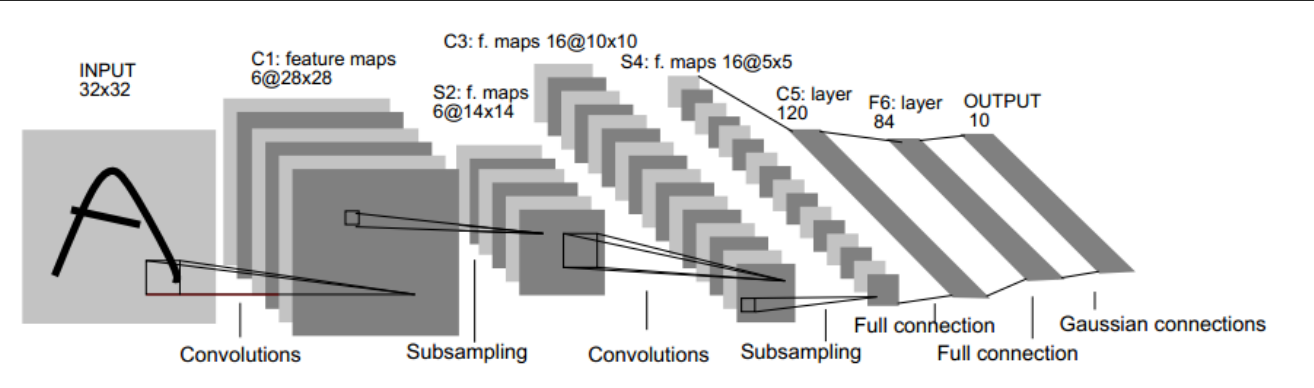
LeNet-5是Yann LeCun在1998年设计的用于手写数字识别的卷积神经网络，是早期卷积神经网络中最有代表性的实验系统之一。LeNet-5共有 7层（不包括输入层），每层都包含不同数量的训练参数。



1. 输入层是32x32像素的图片，比数据集28x28像素中最大的字符大很多，这样做的原因是能使潜在的特征比如边缘的端点、拐角能够出现在最高层次的特征解码器的接收域的中心。
2. C1层是卷积层，形成6个特征图谱。特征图谱中的每个单元与输入层的一个5x5的相邻区域相连，即卷积的输入区域大小是5x5，深度为6。每个特征图谱内参数共享，即每个特征图谱内只使用一个共同卷积核，卷积核有5x5个连接参数加上1个偏置共26个参数。卷积区域不使用0填充、步长为1，这样卷积层形成的特征图谱每个的大小是28x28。C1层共有 $26 \times 6 = 156$ 个训练参数，有 $(5 \times 5 + 1) \times 28 \times 28 \times 6 = 122304$ 个连接。

LeNet-5 模型

<http://scs.ryerson.ca/~aharley/vis/conv/>



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3			X	X	X		X	X	X	X			X		X	X
4				X	X	X		X	X	X	X		X	X		X
5					X	X	X		X	X	X	X		X	X	X

- 3. S2层是一个池化层（下采样层）。输入为第一层的输出，是一个28×28×6的节点矩阵，过滤器大小为2×2，长和宽的步长都为2。本层的输出矩阵大小为14×14×6，采样方式为4个输入相加，乘以一个可训练参数，加上一个可训练偏置，再通过sigmoid，共有2×6=12个训练参数（和的权+偏置），5×14×14×6=5880个连接。
- 4. C3层是一个卷积层，和C1不同的是C3的每个节点与S2中的多个图相连。C3层有16个10×10的图，每个图与S2层的连接的方式如上表所示。这种不对称的组合连接的方式有利于提取多种组合特征。改成有(5×5×3+1)×6 + (5×5×4 + 1) × 3 + (5×5×4 + 1)×6 + (5×5×6+1)×1 = 1516个训练参数，共有1516×10×10=151600个连接。
- 5. S4是一个池化层。C3层的16个10×10的图分别进行以2×2为单位的下抽样得到16个5×5的图。这一层有2×16共32个训练参数，5×5×5×16=2000个连接，连接的方式与S2层类似。
- 6. C5层是一个卷积层。由于S4层的16个图的大小为5×5，与卷积核的大小相同，所以卷积后形成的图的大小为1×1。这里形成120个输出节点。每个都与上一层的16个图相连。共有(5×5×16+1)×120 = 48120个训练参数，同样有48120个连接。

LeNet-5 模型

7. F6层是**全连接层**。F6层有84个节点，对应于一个7x12的比特图，-1表示白色，1表示黑色，这样每个符号的比特图的黑白色就对应于一个编码，**该层的训练参数和连接数是 $(120 + 1) \times 84 = 10164$** 。
8. Output层也是**全连接层**，共有10个节点，分别代表数字0到9，采用了**RBF函数**，即**径向欧式距离函数**，越接近于某数字的比特图编码，表示当前网络输入的识别结果是该字符，**该层的训练参数和连接数为 $84 \times 10 = 840$** 。

LeNet-5类似实例

```
import tensorflow as tf
INPUT_NODE = 784
OUTPUT_NODE = 10
IMAGE_SIZE = 28
NUM_CHANNELS = 1
NUM_LABELS = 10
CONV1_DEEP = 32
CONV1_SIZE = 5
CONV2_DEEP = 64
CONV2_SIZE = 5
FC_SIZE = 512
```

定义卷积神经网络中的参数和前向传播过程

```
def inference(input_tensor, train, regularizer): #train用于区分训练过程和测试过程，在这个程序中将用到dropout，且只在训练时使用
    with tf.variable_scope('layer1-conv1'): #第一层：卷积层
        conv1_weights = tf.get_variable(
            "weight", [CONV1_SIZE, CONV1_SIZE, NUM_CHANNELS, CONV1_DEEP],
            initializer=tf.truncated_normal_initializer(stddev=0.1)) #初始化，标准差为0.1
        conv1_biases = tf.get_variable("bias", [CONV1_DEEP], initializer=tf.constant_initializer(0.0))
        conv1 = tf.nn.conv2d(input_tensor, conv1_weights, strides=[1, 1, 1, 1], padding='SAME')
        relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_biases))
```

定义卷积神经网络中的参数和前向传播过程（续）

```
with tf.name_scope("layer2-pool1"): #第二层: 池化层
    pool1 = tf.nn.max_pool(relu1, ksize = [1,2,2,1],strides=[1,2,2,1],padding="SAME")

with tf.variable_scope("layer3-conv2"): #第三层: 卷积层
    conv2_weights = tf.get_variable(
        "weight", [CONV2_SIZE, CONV2_SIZE, CONV1_DEEP, CONV2_DEEP],
        initializer=tf.truncated_normal_initializer(stddev=0.1))
    conv2_biases = tf.get_variable("bias", [CONV2_DEEP], initializer=tf.constant_initializer(0.0))
    conv2 = tf.nn.conv2d(pool1, conv2_weights, strides=[1, 1, 1, 1], padding='SAME')
    relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_biases))

with tf.name_scope("layer4-pool2"): #第四层: 池化层
    pool2 = tf.nn.max_pool(relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
    pool_shape = pool2.get_shape().as_list()
    nodes = pool_shape[1] * pool_shape[2] * pool_shape[3] #第五层输入为向量, 这里需要将7×7×64矩阵转化为1维。
    reshaped = tf.reshape(pool2, [pool_shape[0], nodes])

with tf.variable_scope('layer5-fc1'): #第五层: 全连接层
    fc1_weights = tf.get_variable("weight", [nodes, FC_SIZE],
                                   initializer=tf.truncated_normal_initializer(stddev=0.1))
    if regularizer != None: tf.add_to_collection('losses', regularizer(fc1_weights)) #正则项
    fc1_biases = tf.get_variable("bias", [FC_SIZE], initializer=tf.constant_initializer(0.1))
    fc1 = tf.nn.relu(tf.matmul(reshaped, fc1_weights) + fc1_biases)
    if train: fc1 = tf.nn.dropout(fc1, 0.5) # dropout 参数

with tf.variable_scope('layer6-fc2'): #第六层: 全连接层
    fc2_weights = tf.get_variable("weight", [FC_SIZE, NUM_LABELS],
                                   initializer=tf.truncated_normal_initializer(stddev=0.1))
    if regularizer != None: tf.add_to_collection('losses', regularizer(fc2_weights))
    fc2_biases = tf.get_variable("bias", [NUM_LABELS], initializer=tf.constant_initializer(0.1))
    logits = tf.matmul(fc1, fc2_weights) + fc2_biases

return logits
```


定义卷积神经网络中的参数和训练过程

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import LeNet5_infernece
import os
import numpy as np
BATCH_SIZE = 100
LEARNING_RATE_BASE = 0.01
LEARNING_RATE_DECAY = 0.99
REGULARIZATION_RATE = 0.0001
TRAINING_STEPS = 30000
MOVING_AVERAGE_DECAY = 0.99
MODEL_SAVE_PATH = "MNIST_model/"
MODEL_NAME = "mnist_model"

def train(mnist):
    # 定义输出为4维矩阵的placeholder
    x = tf.placeholder(tf.float32, [
        BATCH_SIZE,
        LeNet5_infernece.IMAGE_SIZE,
        LeNet5_infernece.IMAGE_SIZE,
        LeNet5_infernece.NUM_CHANNELS],
        name='x-input')
    y_ = tf.placeholder(tf.float32, [None, LeNet5_infernece.OUTPUT_NODE], name='y-input')
    regularizer = tf.contrib.layers.l2_regularizer(REGULARIZATION_RATE) # L2正则
    y = LeNet5_infernece.inference(x, True, regularizer) # 训练过程, 使用dropout 和 正则项来减小过拟合
    global_step = tf.Variable(0, trainable=False)

    # 定义损失函数、学习率、滑动平均操作以及训练过程。
    variable_averages = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY, global_step) # 滑动平均模型
    variables_averages_op = variable_averages.apply(tf.trainable_variables()) # 定义一个更新滑动平均的操作
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_)
    cross_entropy_mean = tf.reduce_mean(cross_entropy)
    loss = cross_entropy_mean + tf.add_n(tf.get_collection('losses'))
```

定义卷积神经网络中的参数和训练过程（续）

```
learning_rate = tf.train.exponential_decay(
    LEARNING_RATE_BASE,
    global_step,
    mnist.train.num_examples / BATCH_SIZE, LEARNING_RATE_DECAY,
    staircase=True) # 学习率

train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss, global_step=global_step)
with tf.control_dependencies([train_step, variables_averages_op]):
    train_op = tf.no_op(name='train')

# 初始化TensorFlow持久化类。
saver = tf.train.Saver()
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for i in range(TRAINING_STEPS):
        xs, ys = mnist.train.next_batch(BATCH_SIZE)
        reshaped_xs = np.reshape(xs, (
            BATCH_SIZE,
            LeNet5_infernece.IMAGE_SIZE,
            LeNet5_infernece.IMAGE_SIZE,
            LeNet5_infernece.NUM_CHANNELS)) # 将输入的数据格式转化为该卷积神经网络的格式
        _, loss_value, step = sess.run([train_op, loss, global_step], feed_dict={x: reshaped_xs, y_: ys})

        if i % 1000 == 0:
            print("After %d training step(s), loss on training batch is %g." % (step, loss_value))
            saver.save(sess, os.path.join(MODEL_SAVE_PATH, MODEL_NAME), global_step=global_step)

def main(argv=None):
    mnist = input_data.read_data_sets(r"F:\Study\Python\Tensorflow\tensorflow-tutorial-
master\Deep_Learning_with_TensorFlow\datasets\MNIST_data", one_hot=True)
    train(mnist)

if __name__ == '__main__':
    main()
```

```

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import LeNet5_inference
import LeNet5_train
import os
import numpy as np
BATCH_SIZE = 100
MODEL_SAVE_PATH = "MNIST_model/"
MODEL_NAME = "mnist_model"
def evaluate(mnist):
    with tf.Graph().as_default() as g:
        x = tf.placeholder(tf.float32, [BATCH_SIZE,
            LeNet5_infernece.IMAGE_SIZE,
            LeNet5_infernece.IMAGE_SIZE,
            LeNet5_infernece.NUM_CHANNELS], name='x-input')
        y_ = tf.placeholder(tf.float32, [None, LeNet5_infernece.OUTPUT_NODE], name='y-input')
        validate_feed = {x: mnist.validation.images, y_: mnist.validation.labels}
        y = LeNet5_infernece.inference(x,False,None)
        correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
        variable_averages = tf.train.ExponentialMovingAverage(LeNet5_train.MOVING_AVERAGE_DECAY)
        variables_to_restore = variable_averages.variables_to_restore()
        saver = tf.train.Saver(variables_to_restore)
        with tf.Session() as sess:
            ckpt = tf.train.get_checkpoint_state(MODEL_SAVE_PATH)
            if ckpt and ckpt.model_checkpoint_path:
                saver.restore(sess, ckpt.model_checkpoint_path)
                global_step = ckpt.model_checkpoint_path.split('/')[-1].split('-')[-1]
                xs, ys = mnist.train.next_batch(BATCH_SIZE)
                reshaped_xs = np.reshape(xs, (
                    BATCH_SIZE,
                    LeNet5_infernece.IMAGE_SIZE,
                    LeNet5_infernece.IMAGE_SIZE,
                    LeNet5_infernece.NUM_CHANNELS))

```

```

accuracy_score = sess.run(accuracy, feed_dict={x: reshaped_xs, y_: ys})
        print("After %s training step(s),
validation accuracy = %g" % (global_step, accuracy_score))
    else:
        print('No checkpoint file found')

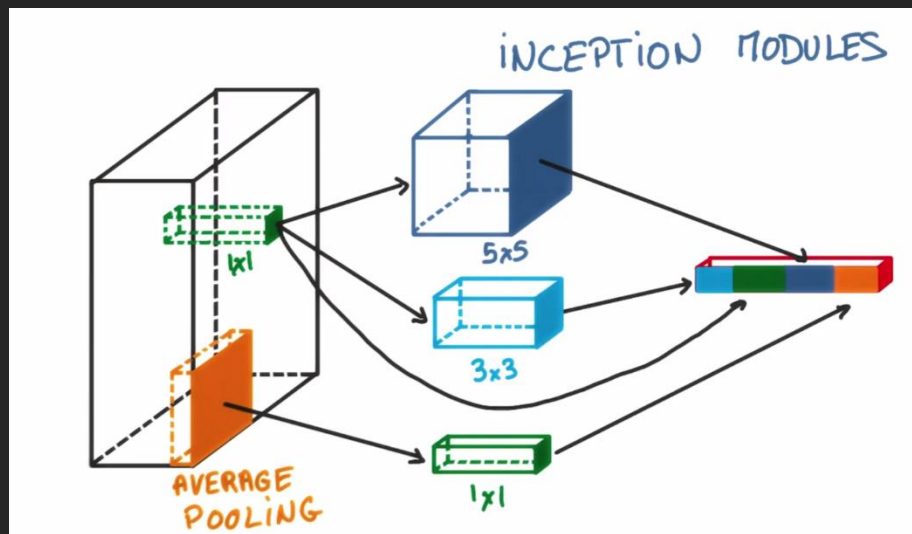
def main(argv=None):
    mnist =
input_data.read_data_sets(r"F:\Study\Python\Tensorflow\tensorflow-
tutorial-master\Deep_Learning_with_TensorFlow\datasets\MNIST_data",
one_hot=True)
    evaluate(mnist)
if __name__ == '__main__':
    main()

```

卷积神经网络测试过程

Inception 模型

在LeNet-5 模型中，不同卷积层通过**串联**的方式连接在一起，而 Inception 模型则是将不同的卷积层通过**并联**的方式结合在一起。将 1×1 ， 3×3 ， 5×5 的conv和 3×3 的pooling，堆叠在一起，一方面增加了网络的width，另一方面增加了网络对尺度的适应性。Inception -v3 是其中重要的改进，v3模型共有46层，由11个inception 模块组成，同时将 7×7 分解成两个一维的卷积（ $1\times 7, 7\times 1$ ）， 3×3 也是一样（ $1\times 3, 3\times 1$ ），这样的好处，既可以加速计算（多余的计算能力可以用来加深网络），又可以将1个conv拆成2个conv，使得网络深度进一步增加，增加了网络的非线性。



各版本相关论文

[v1] Going Deeper with Convolutions, 6.67% test error

<http://arxiv.org/abs/1409.4842>

[v2] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 4.8% test error

<http://arxiv.org/abs/1502.03167>

[v3] Rethinking the Inception Architecture for Computer Vision, 3.5% test error

<http://arxiv.org/abs/1512.00567>

[v4] Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, 3.08% test error

<http://arxiv.org/abs/1602.07261>

TF-slim

GitHub:<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim>

TF-slim是一个使构建，训练，评估神经网络变得简单的库。它可以消除原生tensorflow里面很多重复的模板性的代码，让代码更紧凑，更具备可读性。另外slim提供了很多计算机视觉方面的著名模型（VGG, AlexNet等），我们不仅可以直接使用，甚至能以各种方式进行扩展。

slim被放在tensorflow.contrib这个库下面，导入方法：`import tensorflow.contrib.slim as slim`

各子模块及功能介绍：

1. `arg_scope`: provides a new scope named `arg_scope` that allows a user to define default arguments for specific operations within that scope.
2. `data`: contains TF-slim's dataset definition, data providers, `parallel_reader`, and decoding utilities.
3. `evaluation`: contains routines for evaluating models.
4. `layers`: contains high level layers for building models using tensorflow.
5. `learning`: contains routines for training models.
6. `losses`: contains commonly used loss functions.
7. `metrics`: contains popular evaluation metrics.
8. `nets`: contains popular network definitions such as VGG and AlexNet models.
9. `queues`: provides a context manager for easily and safely starting and closing `QueueRunners`.
10. `regularizers`: contains weight regularizers.
11. `variables`: provides convenience wrappers for variable creation and manipulation.

实例：使用 slim 实现 inception-v3模型中的一个 inception 模块：

```
import tensorflow as tf
import tensorflow.contrib.slim as slim # slim被放在tensorflow.contrib这个库下面
net = 上一层输出节点矩阵
with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_pool2d], stride = 1, padding = 'SAME'):
    # slim.arg_scope 用于设置默认的参数取值，第一个参数为一个函数列表，在这个列表中的函数将使用后续中的值作为默认值。
    with tf.variable_scope('Mixed_7c'): # 为一个inception模块声明一个统一的变量命名空间
        with tf.variable_scope('Branch_0'): # 为此路径声明一个命名空间
            branch_0 = slim.conv2d(net, 320, [1, 1], scope='Conv2d_0a_1x1') # 边长为1，深度为320的卷积层

        with tf.variable_scope('Branch_1'):
            branch_1 = slim.conv2d(net, 384, [1, 1], scope='Conv2d_0a_1x1')
            branch_1 = tf.concat(3, [ # 将多个矩阵拼接起来，第一个参数“3”指定了拼接的维度
                slim.conv2d(branch_1, 384, [1, 3], scope='Conv2d_0b_1x3'),
                slim.conv2d(branch_1, 384, [3, 1], scope='Conv2d_0c_3x1')
            ])

        with tf.variable_scope('Branch_2'):
            branch_2 = slim.conv2d(net, 448, [1, 1], scope='Conv2d_0a_1x1')
            branch_2 = slim.conv2d(branch_2, 384, [3, 3], scope='Conv2d_0b_3x3')
            branch_2 = tf.concat(3, [
                slim.conv2d(branch_2, 384, [1, 3], scope='Conv2d_0c_1x3'),
                slim.conv2d(branch_2, 384, [3, 1], scope='Conv2d_0d_3x1')
            ])

        with tf.variable_scope('Branch_3'):
            branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
            branch_3 = slim.conv2d(branch_3, 192, [1, 1], scope='Conv2d_0b_1x1')

    net = tf.concat(3, [branch_0, branch_1, branch_2, branch_3]) #当前的最后输出由四个计算结果拼接得到
```

TFRecords

使用Tensorflow训练神经网络时，我们可以用多种方式来读取自己的数据。如果数据集比较小，而且内存足够大，可以选择直接将所有数据读进内存，然后每次取一个batch的数据出来。如果数据较多，可以每次直接从硬盘中进行读取，这种方式的读取效率比较低。Tensorflow官方推荐的一种较为高效的数据读取方式——TFRecords，TFRecords实际上是一种**二进制文件**，虽然它不如其他格式好理解，但是它能更好的利用内存，更方便复制和移动，并且不需要单独的标签文件。当数据量极大时，可以将数据分成多个 TFRecords 文件来提高效率。

TFRecords文件包含了`tf.train.Example` **协议内存块**(protocol buffer)(协议内存块包含了字段 Features)。可以将数据填入到Example协议内存块，将协议内存块序列化为一个字符串，并且通过`tf.python_io.TFRecordWriter` 写入到TFRecords文件。从TFRecords文件中读取数据，可以使用`tf.TFRecordReader`的`tf.parse_single_example`解析器将Example协议内存块(protocol buffer)解析为张量。

```
message Example {
  Features features = 1;
};

message Features{
  map<string,Feature> featrue = 1;
};

message Feature{
  oneof kind{
    BytesList bytes_list = 1;
    FloatList float_list = 2;
    Int64List int64_list = 3;
  }
};
```

`tf.train.Example`中包含了一个从属性名称到取值的字典，其中属性名称是一个字符串，属性的取值可以为**字符串 (BytesList)**，**实数列表 (FloatList)** 或者 **整数列表 (Int64List)**，例如我们可以将解码前的图片作为字符串，图像对应的类别标号作为整数列表。

实例：将MNIST输入数据转化为TFRecords 格式

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
# 生成整数型的属性
def _int64_feature(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
# 生成字符串型的属性
def _bytes_feature(value):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

# 读取mnist数据
mnist = input_data.read_data_sets("../..../datasets/MNIST_data", dtype=tf.uint8, one_hot=True)
images = mnist.train.images
labels = mnist.train.labels # 训练数据对应的正确答案
pixels = images.shape[1] # 训练数据的图像尺寸
num_examples = mnist.train.num_examples

# 输出TFRecord文件的地址
filename = "Records/output.tfrecords"
writer = tf.python_io.TFRecordWriter(filename)
for index in range(num_examples):
    image_raw = images[index].tostring() # 将图像矩阵转化为一个字符串
    example = tf.train.Example(features=tf.train.Features(feature={
        'pixels': _int64_feature(pixels),
        'label': _int64_feature(np.argmax(labels[index])),
        'image_raw': _bytes_feature(image_raw)
    })) # 将一个样例转化为 Example Protocol Buffer
    writer.write(example.SerializeToString())
writer.close()
print("TFRecords文件已保存")
```

实例：读取 TFRecords 格式数据

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
# 创建一个reader来读取文件
reader = tf.TFRecordReader()
filename_queue = tf.train.string_input_producer(["Records/output.tfrecords"])
_, serialized_example = reader.read(filename_queue)

# 解析读取的样例
features = tf.parse_single_example(
    serialized_example,
    features={
        'image_raw':tf.FixedLenFeature([],tf.string),
        'pixels':tf.FixedLenFeature([],tf.int64),
        'label':tf.FixedLenFeature([],tf.int64)
    })

images = tf.decode_raw(features['image_raw'],tf.uint8) # 将字符串解析成图像对应的像素数组
labels = tf.cast(features['label'],tf.int32)
pixels = tf.cast(features['pixels'],tf.int32)

sess = tf.Session()

# 启动多线程处理输入数据
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)

for i in range(10):
    image, label, pixel = sess.run([images, labels, pixels])
```

队列

在TensorFlow中，队列不仅是一种数据结构，也是多线程输入数据处理框架的基础。队列和变量类似，都是计算图上有状态的节点，其他节点可以修改它的内容，对于变量，可以通过赋值操作改变变量的取值。对于队列，其他节点可以把新元素插入到队列后端(rear)，也可以把队列前端(front)的元素删除，修改队列状态的操作主要有：Enqueue（进队列）、EnqueueMany（一次进多个）、Dequeue（出队列）。在使用队列前需要明确调用初始化过程。

TensorFlow 提供了 `FIFOQueue`（先进先出）和 `RandomShuffleQueue`（随机）两种队列，其中 `RandomShuffleQueue` 每次出队操作得到的是队列内所有元素中随机选择的一个。

Client

```
q = tf.FIFOQueue(3, "float")
init = q.enqueue_many([[0.,0.,0.]])

x = q.dequeue()
y = x+1
q_inc = q.enqueue([y])

init.run()
q_inc.run()
q_inc.run()
q_inc.run()
q_inc.run()
```

```
import tensorflow as tf
q = tf.RandomShuffleQueue(capacity=10,min_after_dequeue=2,dtypes="float32")
#创建一个随机队列，最大长度为10，出队后的最小长度为2，数据类型为32位浮点型
with tf.Session() as sess:
    for i in range(0, 10):
        sess.run(q.enqueue(i))
    for i in range(0, 8): # 在输出8次后会被阻塞
        print(sess.run(q.dequeue()))
```

输出结果：

8.0
7.0
3.0
9.0
2.0
4.0
1.0
5.0

队列

TensorFlow 提供了 `tf.Coordinator` 和 `tf.QueueRunner` 两个类来完成多线程协同的功能，`tf.Coordinator` 主要用于协同多个线程一起停止，并提供了 `should_stop`、`request_stop` 和 `join` 三个函数，在启动线程前，首先需要声明一个 `tf.Coordinator` 类，并将这个类传入每一个创建的线程中。启动的线程需要一直查询 `should_stop` 函数，当返回值为 `True` 时，当前的线程也需要退出。每一个启动的线程都可以通过调用 `request_stop` 函数来通知其他线程退出，当某一个线程调用 `request_stop` 函数之后，`should_stop` 函数的返回值将被设置为 `True`，这样其他的线程就可以同时终止了。

```
import tensorflow as tf
import numpy as np
import threading
import time

def MyLoop(coord, worker_id):
    while not coord.should_stop():
        if np.random.rand() < 0.1: # 随机停止所有进程
            print("Stoping from id: %d\n" % worker_id,)
            coord.request_stop()
        else:
            print("Working on id: %d\n" % worker_id,)
            time.sleep(1)

coord = tf.train.Coordinator()
threads = [threading.Thread(target=MyLoop, args=(coord, i, )) for i in range(5)]
for t in threads: t.start()
# 此处为 列表解析，效果和下两句类似
# for i in range(5):
#     threads = threading.Thread(target=MyLoop, args=(coord, i, )).start()
coord.join(threads) # 其他所有线程关闭之后，这一函数才能返回
```

输出结果：

```
Working on id: 0
Working on id: 1

Working on id: 2
Working on id: 3
Working on id: 4

Working on id: 0

Working on id: 1

Stoping from id: 2
Working on id: 3
```

队列

`tf.QueueRunner` 主要用于启动多个线程来操作同一个队列，第一个参数给出被操作的队列，第二个给出了队列的操作，可以定义好一个 `op` 后，使用 `[op] * N` 作为参数，表示需要启动N个线程，每个线程中运行的是 `op` 操作。在`tf.train`中要创建队列和执行入队操作，就要添加 `tf.train.QueueRunner`到一个使用`tf.train.add_queue_runner`函数的数据流图中。一旦数据流图构造成功，`tf.train.start_queue_runners` 函数就会要求数据流图中每个`QueueRunner`去开始它的线程运行入队操作。

实例代码：

```
import tensorflow as tf
queue = tf.FIFOQueue(100, "float")
# 声明一个先入先出队列
enqueue_op = queue.enqueue([tf.random_normal([1])])
# 定义队列的入队操作
qr = tf.train.QueueRunner(queue, [enqueue_op] * 5)
# 启动五个线程
tf.train.add_queue_runner(qr)
out_tensor = queue.dequeue()
# 定义出队操作
with tf.Session() as sess:
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    for _ in range(3):
        print(sess.run(out_tensor)[0]) # 获取队列中的取值
    coord.request_stop()
    coord.join(threads)
```

启动五个线程来执行队列入队操作，每个线程都是将随机数写入队列，于是每次运行出队操作时，可以得到一个随机数

输出结果：

```
-0.245295
1.13642
-0.0321272
```


输入文件队列

TensorFlow 提供了 `tf.train.match_filenames_once` 函数来获取符合某一正则表达式的所有文件，得到的文件列表可以通过 `tf.train.string_input_producer` 函数来进行有效的管理。`tf.train.string_input_producer` 会使用初始化时提供的文件列表创建一个输入队列，输入队列中原始的元素为文件列表中的所有文件。每次调用文件读取函数时，该函数会先判断是否已有打开的文件可读，如果没有，则会出队一个文件中读取数据。

通过设置 `shuffle` 参数，支持随机打乱文件列表中文件出队的顺序。生成的输入队列可以同时被多个文件读取线程获取，当一个输入队列中的所有文件都被处理完后，他会将初始化时提供的文件列表中的文件全部重新加入队列，可以设置 `num_epochs` 参数来限制加载初始文件列表的最大轮数，在测试神经网络模型时往往设置为1。

生成文件存储样例数据

```
import tensorflow as tf
def _int64_feature(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
num_shards = 2
instances_per_shard = 2
for i in range(num_shards):
    filename = ('data.tfrecords-%.5d-of-%.5d' % (i, num_shards))
    # 将Example结构写入TFRecord文件。
    writer = tf.python_io.TFRecordWriter(filename+'.tfrecords')
    for j in range(instances_per_shard):
        # Example结构仅包含当前样例属于第几个文件以及是当前文件的第几个样本。
        example = tf.train.Example(features=tf.train.Features(feature={
            'i': _int64_feature(i),
            'j': _int64_feature(j)}))
        writer.write(example.SerializeToString())
    writer.close()
```



data.tfrecords-00000-of-00002.tfrecords
data.tfrecords-00001-of-00002.tfrecords

读取文件

```
import tensorflow as tf
directory = "data.tfrecords-*.tfrecords"
files = tf.train.match_filenames_once(directory)
# 获取文件列表
filename_queue = tf.train.string_input_producer(files, shuffle=True)
# 创建输入队列
reader = tf.TFRecordReader()
_, serialized_example = reader.read(filename_queue)
features = tf.parse_single_example(
    serialized_example,
    features={
        'i': tf.FixedLenFeature([], tf.int64),
        'j': tf.FixedLenFeature([], tf.int64),
    })

init = (tf.global_variables_initializer(), tf.local_variables_initializer())
# 初始化变量
with tf.Session() as sess:
    sess.run(init)
    print(sess.run(files))
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    # 启动线程
    for i in range(6):
        print(sess.run([features['i'], features['j']]))
    coord.request_stop()
    coord.join(threads)
```

输出结果：

运行三次得到不同顺序

```
>>> runfile('F:/Study/Python/Tensorflow/c7/untitled0.py',
wdir='F:/Study/Python/Tensorflow/c7')
[b'\\.\\data.tfrecords-00000-of-00002.tfrecords'
 b'\\.\\data.tfrecords-00001-of-00002.tfrecords']
[0, 0]
[0, 1]
[1, 0]
[1, 1]
[0, 0]
[0, 1]
>>> runfile('F:/Study/Python/Tensorflow/c7/untitled0.py',
wdir='F:/Study/Python/Tensorflow/c7')
[b'\\.\\data.tfrecords-00000-of-00002.tfrecords'
 b'\\.\\data.tfrecords-00001-of-00002.tfrecords']
[0, 0]
[0, 1]
[1, 0]
[1, 1]
[1, 0]
[1, 1]
>>> runfile('F:/Study/Python/Tensorflow/c7/untitled0.py',
wdir='F:/Study/Python/Tensorflow/c7')
[b'\\.\\data.tfrecords-00000-of-00002.tfrecords'
 b'\\.\\data.tfrecords-00001-of-00002.tfrecords']
[1, 0]
[1, 1]
[0, 0]
[0, 1]
[0, 0]
[0, 1]
```


组合训练数据

TensorFlow 提供了 `tf.train.batch` 和 `tf.train.shuffle_batch` 函数来将单个的样例组织成 batch 的形式输出，这两个函数都会生成一个队列，队列的入队操作时生成单个样例的方法，而每次出队得到的是一个 batch 的样例。通过设置（`tf.train.batch`）`tf.train.shuffle_batch` 的 `num_threads` 参数，可以指定多个线程同时读取一个文件中的不同样例，如果希望多个线程处理 **不同文件中样例**，可以使用 `tf.train.batch_join`（`tf.train.shuffle_batch_join`）函数。

```
import tensorflow as tf
directory = "data.tfrecords-*.tfrecords"
files = tf.train.match_filenames_once(directory)
filename_queue = tf.train.string_input_producer(files, shuffle=False)
reader = tf.TFRecordReader()
_, serialized_example = reader.read(filename_queue)
features = tf.parse_single_example(serialized_example,
    features={'i': tf.FixedLenFeature([], tf.int64), 'j': tf.FixedLenFeature([], tf.int64),})
example, label = features['i'], features['j']
batch_size = 3
capacity = 1000 + 3 * batch_size # 队列大小（过大会消耗过多内存，过小会应为没有数据而被阻碍（block））
#example_batch, label_batch = tf.train.batch([example, label], batch_size = batch_size, capacity = capacity) # 不随机
example_batch, label_batch = tf.train.shuffle_batch([example, label], batch_size=batch_size, capacity=capacity, min_after_dequeue=30)
init = (tf.global_variables_initializer(), tf.local_variables_initializer())
with tf.Session() as sess:
    sess.run(init)
    print(sess.run(files))
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    for i in range(2):
        print(sess.run([example_batch, label_batch]))
    coord.request_stop()
    coord.join(threads)
```

Batching 读取数据样例

```
>>> runfile('F:/Study/Python/Tensorflow/c7/untitled1.py', wdir=
[b'\\.\\data.tfrecords-00000-of-00002.tfrecords'
 b'\\.\\data.tfrecords-00001-of-00002.tfrecords']
[array([0, 0, 1], dtype=int64), array([1, 1, 0], dtype=int64)]
[array([0, 0, 0], dtype=int64), array([1, 0, 0], dtype=int64)]
>>> runfile('F:/Study/Python/Tensorflow/c7/untitled1.py', wdir=
[b'\\.\\data.tfrecords-00000-of-00002.tfrecords'
 b'\\.\\data.tfrecords-00001-of-00002.tfrecords']
[array([1, 0, 0], dtype=int64), array([1, 1, 0], dtype=int64)]
[array([1, 0, 1], dtype=int64), array([1, 1, 1], dtype=int64)]
```