# TensorFlow

TensorFlow的全部目的就是使用一个称之为计算图（computational graph）的东西，它会比直接在Python中进行相同计算量要高效得多。TensorFlow比Numpy更高效，因为TensorFlow了解整个需要运行的计算图，然而Numpy只知道某个时间点上唯一的数学运算。TensorFlow还能利用多核CPU和GPU。

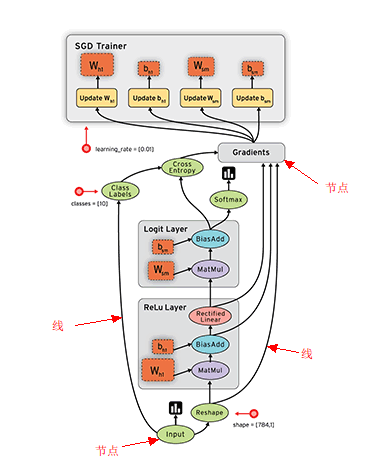
TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于Graph是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

# 基础构架

## 处理结构

1. 计算图纸

TensorFlow是采用数据流图（data、flow、graphs）来计算, 所以首先我们得创建一个数据流流图, 然后再将我们的数据（数据以张量(tensor)的形式存在）放在数据流图中计算. TensorFlow中所有计算都会被转换为计算图上的节点。节点（Nodes）在图中表示数学操作,图中的线（edges）则表示在节点间相互联系的多维数据数组, 即张量（tensor). 训练模型时tensor会不断的从数据流图中的一个节点flow到另一节点, 这就是TensorFlow名字的由来.



1. 张量

张量有多种，可以被简单理解为多维数组。 零阶张量为纯量或标量 (scalar) 也就是一个数值. 比如 [1]

一阶张量为 向量 (vector), 比如 一维的 [1, 2, 3]

二阶张量为 矩阵 (matrix), 比如 二维的 [[1, 2, 3],[4, 5, 6],[7, 8, 9]]

以此类推, 还有 三阶 三维的 …

## session会话

Session 是 Tensorflow 为了控制,和输出文件的执行的语句. 运行 session.run() 可以获得要得知的运算结果, 或者是所要运算的部分.

## Variable 变量

在 Tensorflow 中，定义了某字符串是变量，它才是变量，这一点是与 Python 所不同的。

state = tf.Variable()

定义了变量以后, 一定要定义 init = tf.initialize\_all\_variables() 初始化变量，使用sess.run(init)激活init.

## 特征标准化

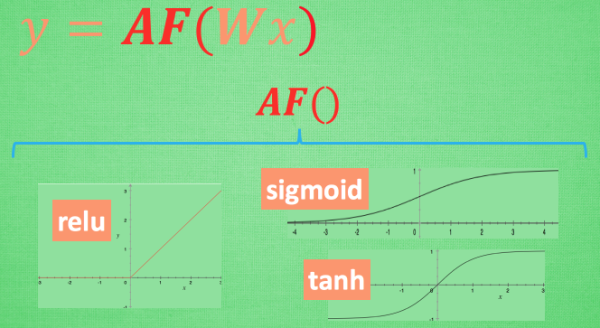
特征标准化(Feature Normalization)的途径有两种,一种叫做min max normalization, 将所有特征数据按比例缩放到0-1的这个取值区间，也可以是-1到1的区间. 还有一种叫做standard deviation normalization, 将所有特征数据缩放成平均值为0,方差为1. 使用这些标准化手段不仅可以快速推进机器学习的学习速度,还可以避免机器学习学得扭曲

## activation function（激励函数）

激励函数运行时激活神经网络中某一部分神经元，将激活信息向后传入下一层的神经系统。激励函数的实质是非线性方程。 Tensorflow 的神经网络 里面处理较为复杂的问题时都会需要运用激励函数 activation function

激励函数是为了解决不能用线性方程所概括的问题.比如说relu, sigmoid, tanh.嵌套在原有的结果之上, 强行把原有的线性结果给扭曲了. 使得输出结果 y 也有了非线性的特征。

甚至可以创造自己的激励函数来处理自己的问题, 不过要确保的是这些激励函数必须是可以微分的, 因为在 backpropagation 误差反向传递的时候, 只有这些可微分的激励函数才能把误差传递回去.



1. 恰当使用这些激励函数

比如当你的神经网络层只有两三层, 不是很多的时候, 对于隐藏层, 使用任意的激励函数,不会有特别大的影响. 不过, 当你使用特别多层的神经网络,需要谨慎选择，因为这会涉及到梯度爆炸, 梯度消失的问题.

1. 默认首选的激励函数

在少量层结构中可以尝试很多种不同的激励函数. 在卷积神经网络CNN的卷积层中, 推荐的激励函数是 relu. 在循环神经网络中 RNN, 推荐的是 tanh 或者是 relu

# TensorFlow计算图

组成：

* 占位符变量（Placeholder）改变图的输入
* 模型变量（Model）将会被优化，使得模型表现更好。
* 模型本质上就是一些数学函数，它根据Placeholder和模型的输入变量来计算一些输出。
* 一个cost度量用来指导变量的优化。
* 一个优化策略会更新模型的变量。

1. **占位符 （Placeholder）变量**

Placeholder作为图的输入，每次运行图的时候都可能会改变它们。这个过程称为feeding placeholder变量。

首先我们为输入图像定义placeholder变量。这让我们可以改变输入到TensorFlow图中的图像。这也是一个张量（tensor），代表一个多维向量或矩阵。

**数据类型**设置为float32，**形状**设为[None, img\_size\_flat]，None代表tensor可能保存着任意数量的图像，每张图象是一个长度为img\_size\_flat的向量，不定义即为任意长。

x = tf.placeholder(tf.float32, [*None*, img\_size\_flat])

1. **需要优化的变量**

除了给模型输入数据的变量之外，TensorFlow还需要改变一些模型变量，使得训练数据的表现更好。

* **权重weight**，TensorFlow变量需要被初始化为零。

权重形状是[img\_size\_flat, num\_classes]，因此它是一个img\_size\_flat行、num\_classes列的二维张量（或矩阵）。

weights = tf.Variable(tf.zeros([img\_size\_flat, num\_classes]))

* **偏差变量biases**，被定义成一个长度为num\_classes的1维张量（或向量）。

biases = tf.Variable(tf.zeros([num\_classes])

1. **模型**

模型将**placeholder变量x** [num\_images, img\_size\_flat]中的图像与**权重weight**[img\_size\_flat, num\_classes]相乘，结果是大小为**[num\_images, num\_classes]**的一个矩阵，然后加上**偏差biases**[num\_classes]添加到矩阵每一行中。

**logits = tf.matmul(x, weights) + biases**

现在logits是一个 num\_images 行num\_classes列的矩阵，第i行第j列的那个元素代表着第i张输入图像有多大可能性是第j个类别。

然而，这是很粗略的估计并且很难解释，因为数值可能很小或很大，因此我们想要对它们做**归一化**，使得logits矩阵的每一行相加为1，每个元素限制在0到1之间。这是用一个称为**softmax**的函数来计算的，结果保存在y\_pred中。

**y\_pred = tf.nn.softmax(logits)**

可以从y\_pred矩阵中取每行最大元素的索引值，来得到预测的类别。

**y\_pred\_cls = tf.argmax(y\_pred, dimension=1)**

1. **优化损失函数**

为了使模型更好地对输入图像进行分类，必须改变**weights**和**biases**变量。

首先需要比较模型的**预测输出y\_pred**和**期望输出y\_true**，来了解目前模型的性能如何。

交叉熵（cross-entropy）是在分类中使用的性能度量,是一个常为正值的连续函数，如果模型的预测值精准地符合期望的输出，它就等于零。因此，优化的目的就是**最小化交叉熵**，通过改变模型中weights和biases的值，使交叉熵越接近零越好。

TensorFlow有一个内置的计算交叉熵的函数。使用logits的值，内部计算了softmax。

**cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits,labels=y\_true)**

现在，我们已经为每个图像分类计算了交叉熵，所以有一个当前模型在每张图上的性能度量。但是为了用交叉熵来指导模型变量的优化，我们需要一个额外的标量值，因此我们简单地利用所有图像分类交叉熵的均值。

**cost = tf.reduce\_mean(cross\_entropy)**

1. **优化方法**

现在，我们有一个需要被最小化的损失度量，接着我们可以创建优化器。在这种情况中，用的是**梯度下降**的基本形式，步长设为0.5。

优化过程并不是在这里执行。实际上，还没计算任何东西，我们只是往TensorFlow图中添加了优化器，以便之后的操作。

**optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.5).minimize(cost)**

1. **性能度量**

需要另外一些性能度量来向用户展示这个过程。

这是一个布尔值向量，代表预测类型是否等于每张图片的真实类型。

**correct\_prediction = tf.equal(y\_pred\_cls, y\_true\_cls)**

上面先将布尔值向量类型转换成浮点型向量，这样子False就变成0，True变成1，然后计算这些值的平均数，以此来计算分类的准确度。

**accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))**

# 运行TensorFlow

1. 创建TensorFlow会话（session）

一旦创建了TensorFlow图，需要创建一个TensorFlow session，用来运行图。

session = tf.Session()

1. 初始化变量

我们需要在开始优化weights和biases变量之前对它们进行初始化。

session.run(tf.global\_variables\_initializer())

1. 运算

运行 session.run() 可以获得你要得知的运算结果, 或者是你所要运算的部分.

session.run(train)

1. 用来优化迭代的帮助函数

在训练集中有50,000张图。用这些图像计算模型的梯度会花很多时间。因此利用随机梯度下降的方法，它在优化器的每次迭代里只用到了一小部分的图像。

batch\_size = 100

函数执行了多次的优化迭代来逐步地提升模型的weights和biases。在每次迭代中，从训练集中选择一批新的数据，然后TensorFlow用这些训练样本来执行优化器。

# 构建神经网络

[源码](https://github.com/MaLei666/TensorFlow/blob/master/%E6%9E%84%E5%BB%BA%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C.py)

## add\_layer（）

定义一个添加层的函数可以很容易的添加神经层，常见的参数通常有weights、biases和激励函数

*def* add\_layer(*input*,*in\_size*,*out\_size*,*activation\_function*=*None*):  
 # 生成初始参数时，随机变量(normal distribution)会比全部为0要好很多  
 weights=tf.Variable(tf.random\_normal([*in\_size*,*out\_size*]))  
 # biases的推荐值不为0  
 biases=tf.Variable(tf.zeros([1,*out\_size*])+0.1)  
 # tf.matmul()是矩阵的乘法  
 out\_pre=tf.matmul(*input*,weights)+biases  
 # 当activation\_function——激励函数为None时，输出就是当前的预测值——Wx\_plus\_b  
 *if activation\_function is None*:  
 outputs=out\_pre  
 # 不为None时，就把Wx\_plus\_b传到activation\_function()函数中得到输出  
 *else*:  
 outputs=*activation\_function*(out\_pre)  
 *return* outputs

## 导入数据

# 构建所需的数据  
x\_data=np.linspace(-1,1,300,dtype=np.float32)[:,np.newaxis]  
noise=np.random.normal(0,0.05,x\_data.shape)   
y\_data=np.square(x\_data)-0.5+noise  
  
# 利用占位符定义我们所需的神经网络的输入

# None代表无论输入有多少都可以，因为输入只有一个特征，所以这里是1  
xs=tf.placeholder(tf.float32,[*None*,1])  
ys=tf.placeholder(tf.float32,[*None*,1])

## 搭建网络

通常神经层都包括输入层、隐藏层和输出层。

这里的输入层只有一个属性，所以我们就只有一个输入；隐藏层可以自己假设，这里假设隐藏层有10个神经元； 输出层和输入层的结构是一样的，所以输出层也是只有一层。 所以，我们构建的是——输入层1个、隐藏层10个、输出层1个的神经网络。

# 定义神经层  
# 定义隐藏层，使用relu激活函数  
l1=add\_layer(xs,1,10,activation\_function= tf.nn.relu)  
# 定义输出层  
prediction=add\_layer(l1,10,1,activation\_function=*None*)  
# 计算预测和真实值的误差，差的平方求和再取平均  
loss=tf.reduce\_mean(tf.reduce\_sum(tf.square(ys-prediction),reduction\_indices=[1]))  
# 梯度下降法调优  
train\_step=tf.train.GradientDescentOptimizer(0.1).minimize(loss)  
# 变量初始化  
init=tf.global\_variables\_initializer()  
sess=tf.Session()  
sess.run(init)

## 训练

机器学习的内容是train\_step, 用 Session 来 run 每一次 training 的数据，逐步提升神经网络的预测准确性。 (注意：当运算要用到placeholder时，就需要feed\_dict这个字典来指定输入。)

*for* i *in* range(1000):  
 sess.run(train\_step,feed\_dict={xs:x\_data,ys:y\_data})  
 *if* i%50 ==0:  
 print(sess.run(loss,feed\_dict={xs:x\_data,ys:y\_data}))

## 结果可视化

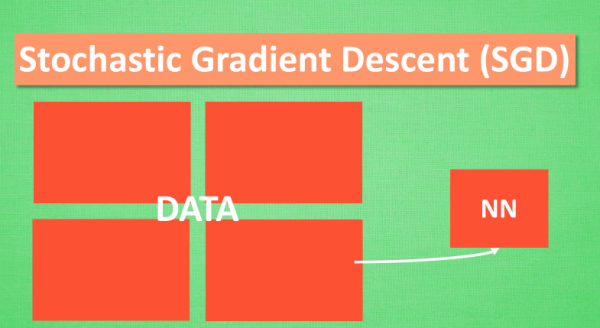
# 散点图描述真实数据  
fig = plt.figure()  
ax = fig.add\_subplot(1,1,1)  
ax.scatter(x\_data, y\_data)

plt.ion() #用于连续显示  
plt.show()

*for* i *in* range(1000):  
 sess.run(train\_step,feed\_dict={xs:x\_data,ys:y\_data})  
 *if* i%50 ==0:  
 print(sess.run(loss,feed\_dict={xs:x\_data,ys:y\_data}))  
 # 显示预测数据，每隔50次训练刷新一次图形  
 *try*:  
 ax.lines.remove(lines[0])  
 *except* Exception:  
 *pass* prediction\_value = sess.run(prediction, feed\_dict={xs: x\_data})  
 # 用红色，宽度为5的线显示预测和输入数据之间的关系  
 lines = ax.plot(x\_data, prediction\_value, 'r-', lw=5)  
 plt.pause(1) # 暂停0.1s  
# 绘图保持打开状态  
plt.ioff()  
plt.show()

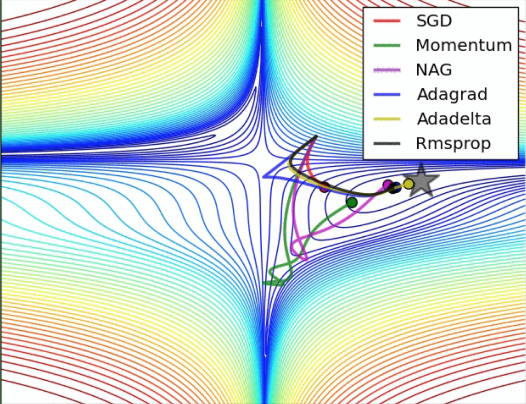
# 加速神经网络训练

## Stochastic Gradient Descent (SGD)



想像红色方块是我们要训练的 data, 如果用普通的训练方法, 就需要重复不断的把整套数据放入神经网络 NN训练, 这样消耗的计算资源会很大.

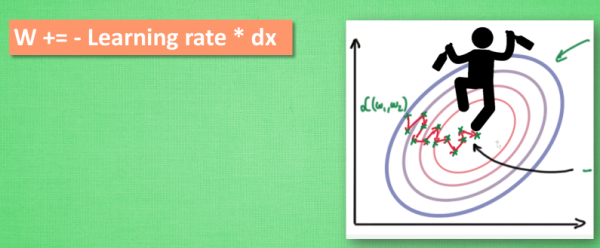
如果把这些数据拆分成小批小批的, 然后再分批不断放入 NN 中计算, 这就是SGD 的正确打开方式. 每次使用批数据, 虽然不能反映整体数据的情况, 不过却很大程度上加速了 NN 的训练过程, 而且也不会丢失太多准确率.



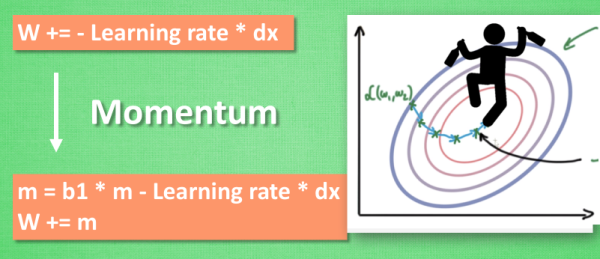
SGD 并不是最快速的训练方法, 红色的线是 SGD, 但它到达学习目标的时间是在这些方法中最长的一种

## Momentum

大多数其他途径是在更新神经网络参数那一步上动动手脚. 传统的参数 W 的更新是把原始的 W 累加上一个负的学习率(learning rate) 乘以校正值 (dx). 这种方法可能会让学习过程曲折无比, 看起来像 喝醉的人回家时, 摇摇晃晃走了很多弯路.

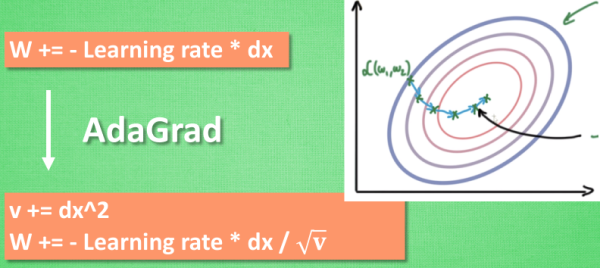


所以我们把这个人从平地上放到了一个斜坡上, 只要他往下坡的方向走一点点, 由于向下的惯性, 他不自觉地就一直往下走, 走的弯路也变少了. 这就是 Momentum 参数更新.



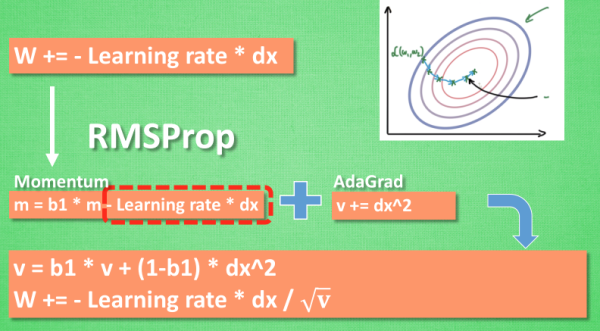
## AdaGrad

这种方法使每一个参数更新都会有自己与众不同的学习率, 他的作用和 momentum 类似, 不过不是给喝醉酒的人安排另一个下坡, 而是给他一双不好走路的鞋子, 使得他一摇晃着走路就脚疼, 鞋子成为了走弯路的阻力, 逼着他往前直着走. 把下坡和不好走路的鞋子合并起来就有了 RMSProp 更新方法.



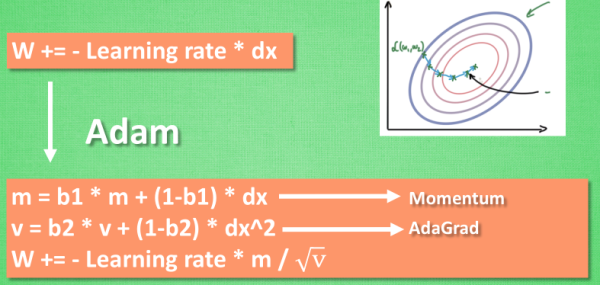
## RMSProp

有了 momentum 的惯性原则, 加上 adagrad 的对错误方向的阻力,让 RMSProp同时具备两种方法的优势. RMSProp 还缺少了 momentum 中的这一部分.在 Adam 方法中补上了这种想法.



## Adam

计算m 时有 momentum 下坡的属性, 计算 v 时有 adagrad 阻力的属性, 然后再更新参数时 把 m 和 V 都考虑进去. 实验证明, 大多数时候, 使用 adam 都能又快又好的达到目标, 迅速收敛.



# Optimizer优化器

# Tensorboard

使用这个工具可以很直观的看到整个神经网络的结构、框架和可视化训练过程。

1. 编辑layer框架

# 添加n\_layer标识层数  
*def* add\_layer(*input*,*in\_size*,*out\_size*,*n\_layer*,*activation\_function*=*None*):  
 # layer\_name 代表其每层的名名称  
 layer\_name = 'layer%s' % *n\_layer*

*with* tf.name\_scope('layer'):

# 定义部件weights、biases、out\_pre  
*with* tf.name\_scope('weights'):  
 weights=tf.Variable(tf.random\_normal([*in\_size*,*out\_size*]),name='w')  
 # tf.histogram\_summary()方法用来绘制图片  
 # 第一个参数是图表的名称, 第二个参数是图表要记录的变量  
 tf.summary.histogram(layer\_name + '/weights', weights)

*with* tf.name\_scope('biases'):  
 biases=tf.Variable(tf.zeros([1,*out\_size*])+0.1,name='b')

# 绘制biases图  
 tf.summary.histogram(layer\_name + '/biases', biases)  
 *with* tf.name\_scope('out\_pre'):  
 out\_pre=tf.matmul(*input*,weights)+biases

# activation\_function 的话，可以暂时忽略。因为当你自己选择用 tensorflow 中的激励函数的时候，tensorflow会默认添加名称  
 *if activation\_function is None*:  
 outputs=out\_pre  
 *else*:  
 outputs=*activation\_function*(out\_pre)  
 # 绘制outputs图  
 tf.summary.histogram(layer\_name + '/outputs', outputs)  
 *return* outputs

1. 编辑input框架

# with tf.name\_scope()将xs和ys包含进来，形成一个大的图层，图层名字就是方法里的参数。  
*with* tf.name\_scope('inputs'):  
 # 为xs指定名称为x\_input，ys为y\_input  
 xs=tf.placeholder(tf.float32,[*None*,1],name='x\_input')  
 ys=tf.placeholder(tf.float32,[*None*,1],name='y\_input')

1. 修改调用

# 修改调用addlayer()函数的地方,添加n\_layer参数  
l1=add\_layer(xs,1,10,n\_layer=1,activation\_function=tf.nn.relu)  
prediction=add\_layer(l1,10,1,n\_layer=2,activation\_function=*None*)

1. 绘制loss和train

# 绘制loss和变化图  
*with* tf.name\_scope('loss'):  
 loss=tf.reduce\_mean(tf.reduce\_sum(tf.square(ys-prediction),reduction\_indices=[1]))

#绘制变化图  
 tf.summary.scalar('loss', loss)  
# 绘制train  
*with* tf.name\_scope('train'):  
 train\_step=tf.train.GradientDescentOptimizer(0.1).minimize(loss)

1. 绘图保存、训练图合并

sess=tf.Session()  
# 使用tf.summary.FileWriter() 将绘画出的图保存到一个目录中  
# 第二个参数需要使用sess.graph，将前面定义的框架信息收集起来，然后放在logs/目录下面  
# 所有训练图合并  
# tf.merge\_all\_summaries() 方法会对所有的 summaries 合并到一起  
merged = tf.summary.merge\_all()  
writer = tf.summary.FileWriter("logs/", sess.graph)  
sess.run(init)

1. 训练数据

*for* i *in* range(1000):  
 sess.run(train\_step,feed\_dict={xs:x\_data,ys:y\_data})  
 *if* i%50 ==0:  
 # merged 也需要run 才能发挥作用  
 rs = sess.run(merged, feed\_dict={xs: x\_data, ys: y\_data})  
 writer.add\_summary(rs, i)  
 print(sess.run(loss,feed\_dict={xs:x\_data,ys:y\_data}))

and so on…

1. 到项目文件夹下，终端开启tensorboard

tensorboard --logdir=logs

浏览器根据终端输出的网址，Chrome打开

[源码](https://github.com/MaLei666/TensorFlow/blob/master/tensorboard_test.py)

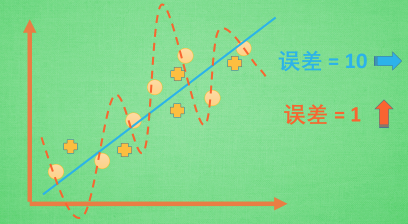
# Classification（分类）学习

分类和回归的区别在于输出变量的类型上。 通俗理解定量输出是回归，或者说是连续变量预测； 定性输出是分类，或者说是离散变量预测。如预测房价这是一个回归任务； 把东西分成几类, 比如猫狗猪牛，就是一个分类任务。

# Overfitting过拟合

## 概念

1. Classification分类情况

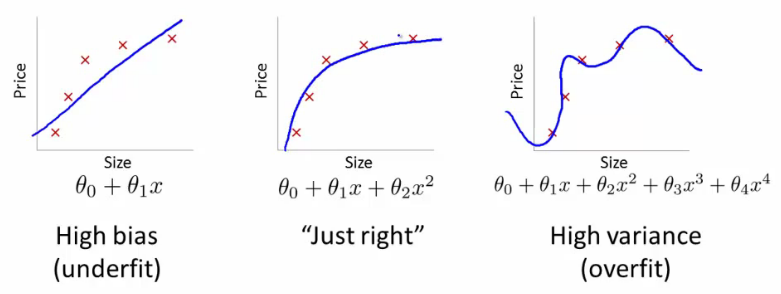


蓝线与数据的总误差是10.有时候机器过于纠结把误差减到更小,所以,学到的会几乎经过每一个数据点,误差值会更小. 尽管、很精确的区分了所有的训练数据，但是并没有描述数据的整体特征，对新测试数据的适应性较差

当这个模型运用在现实中,之前误差大的蓝线误差基本保持不变.误差小的红线误差值突然飙高,就不能成功的表达除了训练数据以外的其他数据.这就是过拟合（Overfitting）

1. Regression回归情况

尽管它经过了所有的训练点，但是不能很好的反应数据的趋势，预测能力严重不足。



解决方法:

1. 增加数据量

大部分过拟合产生的原因是因为数据量太少了. 如果我们有成千上万的数据, 红线也会慢慢被拉直, 变得没那么扭曲

1. 运用正规化

L1, L2 regularization等等, 这些方法适用于大多数的机器学习, 包括神经网络.我们简化机器学习的关键公式为 y=Wx。W为机器需要学习到的各种参数. 在过拟合中, W的值往往变化得特别大或特别小。为了不让W变化太大, 我们在计算误差上做些手脚. 原始的cost误差=预测值-真实值的平方. 如果 W 变得太大, 就让 cost 也跟着变大, 变成一种惩罚机制. 所以把 W 自己考虑进来. 这里 abs 是绝对值. 这一种形式的正规化, 叫做L1正规化. L2 正规化和 L1 类似, 只是绝对值换成了平方. 其他的L3, L4 也都是换成了立方和4次方等等. 形式类似. 用这些方法就能保证让学出来的线条不会过于扭曲.

还有一种专门用在神经网络的正规化的方法, 叫作**dropout**。在训练的时候,随机忽略掉一些神经元和神经联结,使这个神经网络变得”不完整.用一个不完整的神经网络训练一次，到第二次再随机忽略另一些,变成另一个不完整的神经网络。有了这些随机 drop 掉的规则,每次训练的时候都让每一次预测结果都不会依赖于其中某部分特定的神经元. 像l1, l2正规化一样, 过度依赖的 W，也就是训练参数的数值会很大, l1, l2会惩罚这些大的参数. Dropout的做法是从根本上让神经网络没机会过度依赖.

## Dropout解决过拟合

源码

1. 建立dropout层

*import* tensorflow *as* tf  
*from* sklearn.datasets *import* load\_digits  
*from* sklearn.cross\_validation *import* train\_test\_split  
*from* sklearn.preprocessing *import* LabelBinarizer

1. 准备数据

digits=load\_digits()  
x=digits.data  
y=digits.target  
y=LabelBinarizer().fit\_transform(y)  
# x\_train为训练数据，x\_test为测试数据  
x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y,test\_size=.3)

1. 定义添加层函数

*def* add\_layer(*input*,*in\_size*,*out\_size*,*layer\_name*,*activation\_function*=*None*):  
 weights = tf.Variable(tf.random\_normal([*in\_size*, *out\_size*]))  
 biases = tf.Variable(tf.zeros([1, *out\_size*]) + 0.1)  
 out\_pre = tf.matmul(*input*, weights) + biases  
 # 调用dropout函数  
 out\_pre=tf.nn.dropout(out\_pre,keep\_prob)  
*and so on…*

1. 构建所需数据

# keep\_prob定义保留概率，即要保留的结果所占比例，为一个placeholder，在run时传入  
# 当keep\_prob=1的时候，相当于100%保留，也就是dropout没有起作用。  
keep\_prob=tf.placeholder(tf.float32)  
xs=tf.placeholder(tf.float32,[*None*,64]) #8\*8=64个特征  
ys=tf.placeholder(tf.float32,[*None*,10])

1. 添加隐含层和输出层

l1 = add\_layer(xs, 64, 50, 'L1', activation\_function=tf.nn.tanh)  
prediction=add\_layer(l1,50,10,'L2',activation\_function=tf.nn.softmax)

1. 训练

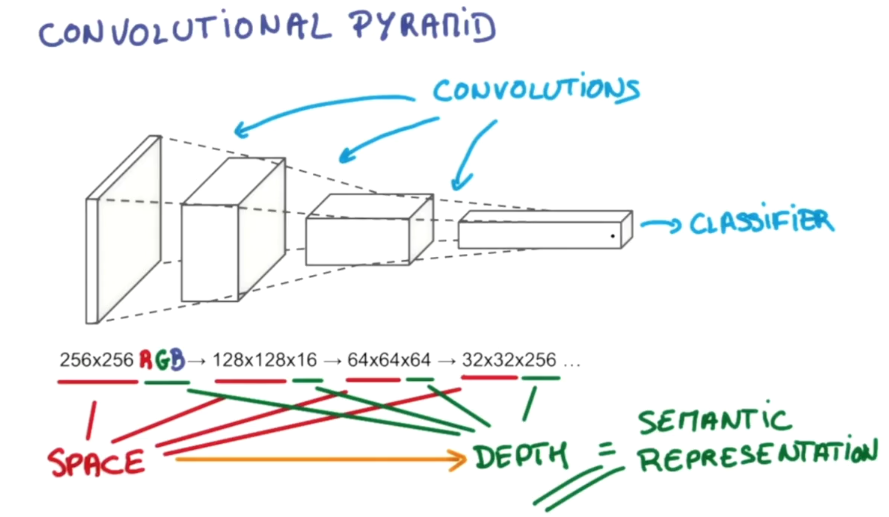
*for* i *in* range(500):  
sess.run(train\_step,feed\_dict={xs:x\_train,ys:y\_train,keep\_prob:1})

*and so on…*

# 卷积神经网络

卷积神经网络（Convolutional Neural Networks）不再是对每个像素的输入信息做处理了,而是图片上每一小块像素区域进行处理, 这种做法加强了图片信息的连续性. 使得神经网络能看到图形, 而非一个点. 这种做法同时也加深了神经网络对图片的理解.

比较流行的一种搭建结构, 首先是输入的图片(image), 经过一层卷积层 (convolution), 然后在用池化(pooling)方式处理卷积的信息, 然后在经过一次同样的处理, 把得到的第二次处理的信息传入两层全连接的神经层 (fully connected),这也是一般的两层神经网络层,最后在接上一个分类器(classifier)进行分类预测.



## 网络结构

卷积神经网络包含**输入层、隐藏层和输出层**，隐藏层又包含卷积层和pooling层

1. 图像输入到卷积神经网络后通过卷积来不断的提取特征，每提取一个特征就会增加一个feature map，所以会不断的增加厚度
2. pooling层也就是下采样，通常采用的是最大值pooling和平均值pooling，因为参数太多，所以通过pooling来稀疏参数，使网络不至于太复杂。

## 程序结构

1. 定义weight和biases变量

# 输入shape，返回变量的参数  
# 使用tf.truncated\_normal产生随机变量来初始化  
*def* weight\_variable(*shape*):  
 inital=tf.truncated\_normal(*shape*,stddev=0.1)  
 *return* tf.Variable(inital)  
# 使用tf.constant常量函数进行初始化  
*def* bias\_variable(*shape*):  
 inital=tf.constant(0.1,shape=*shape*)  
 *return* tf.Variable(inital)

1. 定义卷积和池化

# 定义卷积，tf.nn.conv2d函数是tensoflow里面的二维的卷积函数  
# x是图片的所有参数，W是此卷积层的权重，然后定义步长strides=[1,1,1,1]  
# strides[0]和strides[3]的两个1是默认值，中间两个1代表padding时在x方向运动一步，y方向运动一步，这样得到的图片尺寸没有变化  
# padding采用的方式是SAME。  
*def* conv2d(*x*,*w*):  
 *return* tf.nn.conv2d(*x*,*w*,strides=[1,1,1,1],padding='SAME')  
# 定义池化，pooling 有两种，一种是最大值池化，一种是平均值池化，  
# 采用最大值池化tf.max\_pool()。池化的核函数大小为2x2，因此ksize=[1,2,2,1]，步长为2，因此strides=[1,2,2,1]:  
*def* max\_pool(*x*):  
 *return* tf.nn.max\_pool(*x*,ksize=[1,2,2,1],strides=[1,2,2,1],padding='SAME')

1. 构建数据

xs=tf.placeholder(tf.float32,[*None*,784])/255 #28\*28=784个特征  
ys=tf.placeholder(tf.float32,[*None*,10]) # 输出数字为0-9，共10类  
# keep\_prob定义保留概率，即要保留的结果所占比例，为一个placeholder，在run时传入，当keep\_prob=1的时候，相当于100%保留，也就是dropout没有起作用。  
keep\_prob=tf.placeholder(tf.float32)  
# 重定义输入数据大小，-1代表先不考虑输入的图片例子多少这个维度，后面的1是channel的数量，图片是黑白的channel是1，RGB图像channel是3。  
x\_image=tf.reshape(xs,[-1,28,28,1])

1. 建立卷积层
2. 定义第一层卷积

# 定义weight，选择卷积核patch大小是5x5，黑白图片channel是1，输出是32个featuremap，即使用32个卷积核  
w\_conv1=weight\_variable([5,5,1,32])  
b\_conv1=bias\_variable([32]) # 定义bias，大小是32个长度  
# 定义第一个卷积层h\_conv1，同时对其进行非线性（激活函数tf.nn.relu修正线性单元）处理，因为采用了SAME的padding方式，输出图片的大小依然是28x28，只是厚度变厚，为28\*28\*32大小  
h\_conv1=tf.nn.relu(conv2d(x\_image,w\_conv1)+b\_conv1)  
h\_pool1=max\_pool(h\_conv1) # 经过pooling处理后输出变为14\*14\*32

1. 定义第二层卷积

# 本层输入即为第一层输出，卷积核设置为5\*5，输入为32，输出定为64  
w\_conv2=weight\_variable([5,5,32,64])  
b\_conv2=bias\_variable([64])  
# 定义第二个卷积层h\_conv2，输出大小为14\*14\*64  
h\_conv2=tf.nn.relu(conv2d(h\_pool1,w\_conv2)+b\_conv2)  
# pooling处理后输出变为7\*7\*64  
h\_pool2=max\_pool(h\_conv2)

1. 建立全连接层
2. 建立全连接层1

# 将h\_pool2通过reshape从三维变成一维数据  
# -1表示先不考虑输入图片例子维度, 将结果展平  
h\_pool2\_flat=tf.reshape(h\_pool2,[-1,7\*7\*64])  
# 此时weight\_variable的shape输入就是第二个卷积层展平了的输出大小: 7x7x64，  
# [n\_samples,7,7,64]->>[n\_samples,7\*7\*64]，后面的输出size定为1024  
w\_fc1=weight\_variable([7\*7\*64,1024])  
b\_fc1=bias\_variable([1024])  
# h\_pool2\_flat与本层的W\_fc1相乘（注意这个时候不是卷积了）  
h\_fc1=tf.nn.relu(tf.matmul(h\_pool2\_flat,w\_fc1)+b\_fc1)  
# 加入dropout处理过拟合问题  
h\_fc1\_drop=tf.nn.dropout(h\_fc1,keep\_prob)

1. 建立全连接层2

# 输入为1024，输出为10个类，prediction为预测值  
w\_fc2=weight\_variable([1024,10])  
b\_fc2=bias\_variable([10])  
# 使用softmax分类器对输出进行分类  
prediction=tf.nn.softmax(tf.matmul(h\_fc1\_drop,w\_fc2)+b\_fc2)

1. 优化

# 优化  
# loss函数选用交叉熵函数。交叉熵用来衡量预测值和真实值的相似程度，如果完全相同，交叉熵就等于零  
cross\_entropy=tf.reduce\_mean(-tf.reduce\_sum(ys\*tf.log(prediction),reduction\_indices=[1]))

# 用tf.train.AdamOptimizer()作为优化器进行优化

train\_step=tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)

1. 训练

# 变量初始化  
sess=tf.Session()  
sess.run(tf.global\_variables\_initializer())  
# 训练  
# 机器学习的内容是train\_step，用session来run每次training的数据  
# 当运算要用到placeholder时，就需要feed\_dict这个字典来指定输入。  
*for* i *in* range(1000):  
 batch\_xs,batch\_ys=mnist.train.next\_batch(100)  
sess.run(train\_step,feed\_dict={xs:batch\_xs,ys:batch\_ys,keep\_prob:0.5})  
 *if* i%0==0:  
 print(compute\_accuracy(mnist.test.images[:1000], mnist.test.labels[:1000]))

## 函数简析

**tf.nn.conv2d(input, filter, strides, padding, use\_cudnn\_on\_gpu=None, name=None)**

* input： 指需要做卷积的输入图像，它要求是一个Tensor，具有[batch, in\_height, in\_width, in\_channels]这样的shape，具体含义是[训练时一个batch的图片数量, 图片高度, 图片宽度, 图像通道数]，注意这是一个4维的Tensor，要求类型为float32和float64其中之一
* filter： 相当于CNN中的卷积核，它要求是一个Tensor，具有[filter\_height, filter\_width, in\_channels, out\_channels]这样的shape，具体含义是[卷积核的高度，卷积核的宽度，图像通道数，卷积核个数]，要求类型与参数input相同，有一个地方需要注意，第三维in\_channels，就是参数input的第四维
* strides：卷积时在图像每一维的步长，这是一个一维的向量，长度4
* padding： string类型的量，只能是”SAME”,”VALID”其中之一，这个值决定了不同的卷积方式
* use\_cudnn\_on\_gpu： bool类型，是否使用cudnn加速，默认为true

结果返回一个Tensor，这个输出，就是我们常说的feature map（特征图）

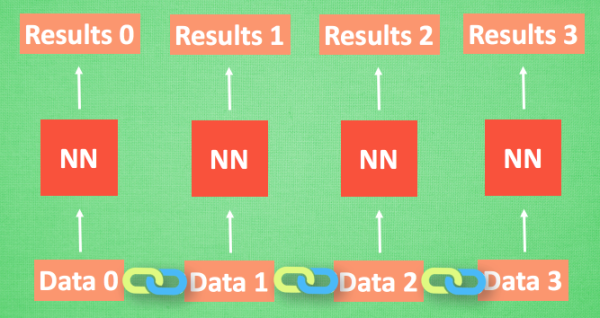
使用Python的captcha库来生成验证码

pip3 install captcha pillow

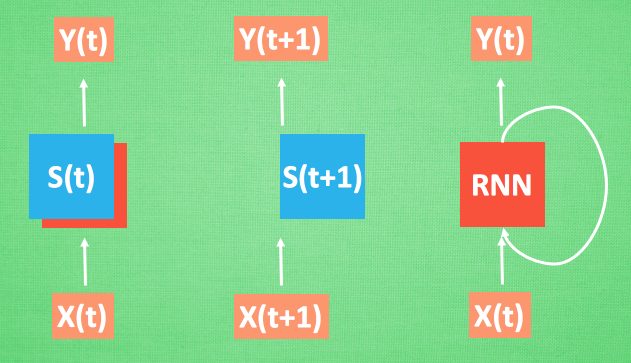
*from* captcha.image *import* ImageCaptcha  
*from* PIL *import* Image  
text = '1234'  
image = ImageCaptcha()  
captcha = image.generate(text)  
captcha\_image = Image.open(captcha)  
captcha\_image.save('test\_img.jpg')

# 循环神经网络

## 循环神经网络（Recurrent Neural Network）



现在有一组序列数据 data 0,1,2,3. 在当预测 result0 的时候基于的是 data0, 在预测其他数据的时候也都只单单基于单个的数据. 每次使用的神经网络都是同一个NN. 不过这些数据是有关联顺序的，就像在厨房做菜,酱料A要比酱料B早放。所以普通的神经网络结构并不能让 NN 了解这些数据之间的关联.

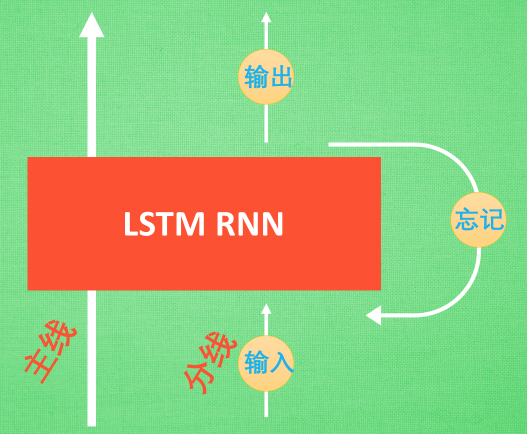


每次RNN运算完之后都会产生一个对于当前状态的描述，state.用简写S(t)代替,然后这个RNN开始分析x(t+1) , 他会根据 x(t+1)产生s(t+1), 不过此时y(t+1)是由s(t)和s(t+1)共同创造的.

## LSTM RNN

RNN是在有顺序的数据上进行学习的. 为了记住这些数据, RNN 会像人一样产生对先前发生事件的记忆. 但是普通 RNN 没有办法回忆起久远记忆。

信息原的记忆要进过长途跋涉才能抵达最后一个时间点. 然后得到误差, 而且在反向传递 得到的误差的时候,他在每一步都会乘以一个自己的参数 W. 如果这个 W 是一个小于1 的数,比如0.9.这个0.9 不断乘以误差, 误差传到初始时间点也会是一个接近于零的数, 所以对于初始时刻, 误差相当于就消失了. 我们把这个问题叫做梯度消失或者梯度弥散 Gradient vanishing. 反之如果 W 是一个大于1 的数, 比如1.1 不断累乘, 则到最后变成了无穷大的数, RNN被这无穷大的数撑死了, 这种情况我们叫做剃度爆炸, Gradient exploding.



LSTM(long-short term memory)就是为了解决这个问题而诞生的.LSTM和普通RNN相比, 多出了三个控制器. (输入控制, 输出控制, 忘记控制).

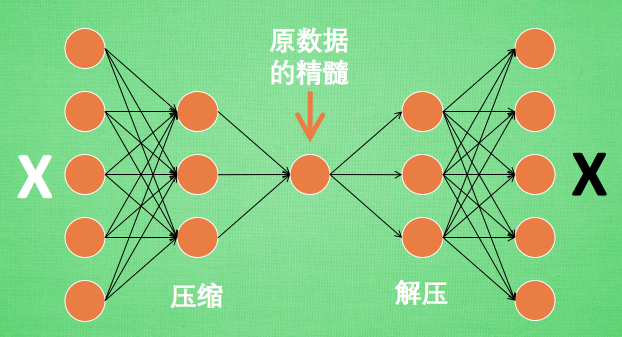
多了一个控制全局的记忆, 我们用粗线代替. 而原本的 RNN 体系就是分线.三个控制器都是在原始的 RNN 体系上。

* 输入方面：如果此时的分线对于结果十分重要, 输入控制就会将这个分线按重要程度写入主线进行分析.再看忘记方面,如果此时的分线更改了我们对之前的想法,那么忘记控制就会将之前的某些主线忘记, 按比例替换成现在的新记忆. 所以主线的更新就取决于输入 和忘记控制.
* 输出方面：输出控制会基于目前的主线和分线判断要输出的到底是什么.基于这些控制机制, LSTM 就像延缓记忆衰退的良药, 可以带来更好的结果.

## RNN分类和回归例子

[分类](https://github.com/MaLei666/TensorFlow/blob/master/Classification_RNN.py)、[回归](https://github.com/MaLei666/TensorFlow/blob/master/Regression_RNN.py)

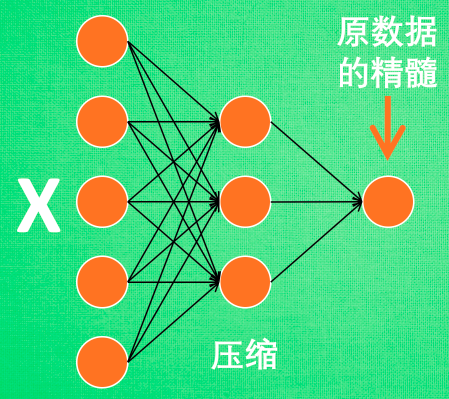
# 自编码（autoencoder）



有时神经网络要接受大量的输入信息, 比如输入信息是高清图片时, 输入信息量可能达到上千万, 让神经网络直接从上千万个信息源中学习是一件很吃力的工作. 所以,压缩一下, 提取出原图片中的最具代表性的信息,缩减输入信息量,再把缩减过后的信息放进神经网络学习.这样学习起来就简单轻松了.

自编码通过将原数据白色的X 压缩,解压成黑色的X,然后通过对比黑白X,求出预测误差,进行反向传递,逐步提升自编码的准确性.训练好的自编码中间这一部分就是能总结原数据的精髓. 可以看出, 从头到尾, 我们只用到了输入数据 X, 并没有用到 X 对应的数据标签, 所以也可以说自编码是一种非监督学习. 到了真正使用自编码的时候. 通常只会用到自编码前半部分.

## 编码器 Encoder



编码器能得到原数据的精髓,只需要再创建一个小的神经网络学习这个精髓的数据,不仅减少了神经网络的负担, 而且同样能达到很好的效果.

## 解码器 Decoder

解码器在训练的时候是要将精髓信息解压成原始信息,提供了一个解压器的作用, 我们可以认为是一个生成器.做这件事的一种特殊自编码为variational autoencoders.

相关代码

# scope 命名方法

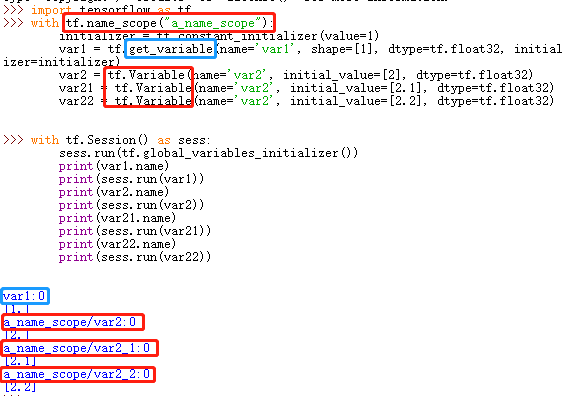
两种定义 scope 的方式——tf.name\_scope()、tf.variable\_scope()

## tf.name\_scope()

在 Tensorflow 当中有两种途径生成变量 variable, 一种是 tf.get\_variable(), 另一种是 tf.Variable().

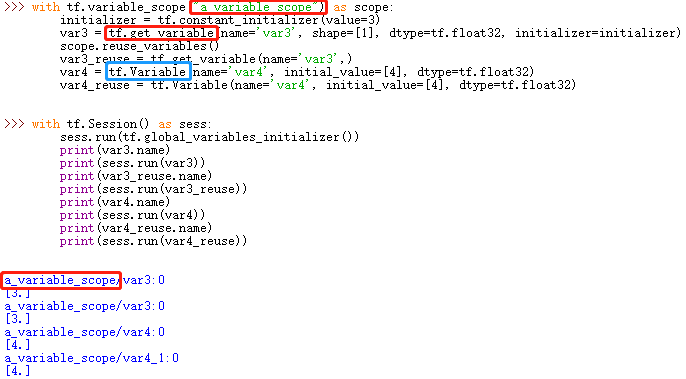
* tf.Variable()

使用 tf.Variable() 定义的时候, 虽然 name 都一样, 但是为了不重复变量名, Tensorflow 输出的变量名并不是一样的。所以, 本质上 var2, var21, var22 并不是一样的变量. 而另一方面, 使用tf.get\_variable()定义的变量不会被tf.name\_scope()当中的名字所影响.



## tf.variable\_scope()

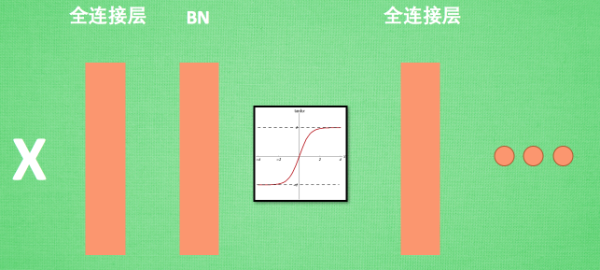
如果想要达到重复利用变量的效果就要使用tf.variable\_scope(),并搭配 tf.get\_variable() 这种方式产生和提取变量. 不像 tf.Variable() 每次都会产生新的变量, tf.get\_variable() 如果遇到了同样名字的变量时, 它会单纯的提取这个同样名字的变量(避免产生新变量). 而在重复使用的时候, 一定要在代码中强调 scope.reuse\_variables(), 否则系统将会报错, 以为你只是单纯的不小心重复使用到了一个变量.



# 批标准化

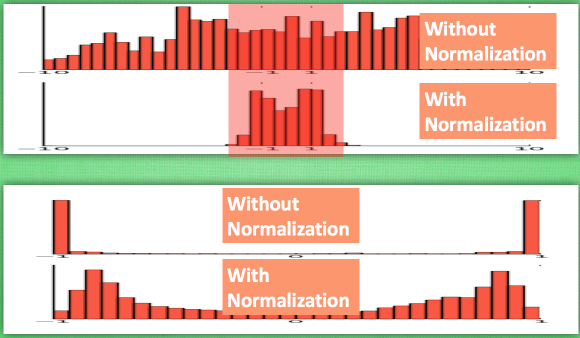
批标准化（Batch Normalization）, 和普通的数据标准化类似, 是将分散的数据统一的一种做法, 也是优化神经网络的一种方法.

在神经网络中, 数据分布对训练会产生影响. 比如某个神经元x的值为1, 某个 Weights 的初始值为0.1, 这样后一层神经元计算结果就是 Wx = 0.1; 又或者 x = 20, 这样 Wx 的结果就为2. 现在还不能看出什么问题, 但是, 当我们加上一层激励函数, 激活这个 Wx 值的时候,问题就来了.如果使用像tanh的激励函数, Wx 的激活值就变成了 ~0.1 和 ~1, 接近于1的部已经处在了激励函数的饱和阶段,也就是如果 x 无论再怎么扩大,tanh激励函数输出值也还是接近1.神经网络在初始阶段已经不对那些比较大的x特征范围敏感了.可以用之前提到的对数据做 normalization 预处理, 使得输入的 x 变化范围不会太大, 让输入值经过激励函数的敏感部分.但这个不敏感问题不仅仅发生在神经网络的输入层, 而且在隐藏层中也经常会发生，batch normalization, 正是处理这种情况.

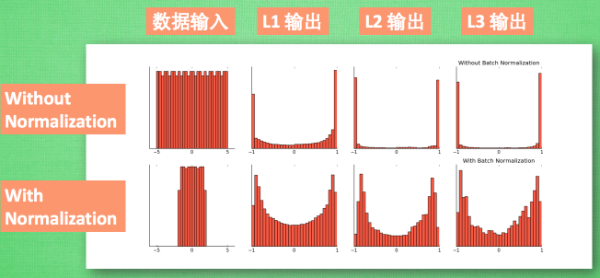


Batch normalization 的 batch 是批数据, 把数据分成小批小批进行 stochastic gradient descent. 而且在每批数据进行前向传递 forward propagation 的时候, 对每一层都进行 normalization 的处理。

Batch normalization 也可以被看做一个层面. 在一层层的添加神经网络的时候,先有数据 X, 再添加全连接层, 全连接层的计算结果会经过激励函数成为下一层的输入,接着重复之前的操作. Batch Normalization (BN) 就被添加在每一个全连接和激励函数之间.



没有 normalize 的数据 使用 tanh 激活以后, 激活值大部分都分布到了饱和阶段, 也就是大部分的激活值不是-1, 就是1, 而 normalize 以后, 大部分的激活值在每个分布区间都还有存在. 再将这个激活后的分布传递到下一层神经网络进行后续计算, 每个区间都有分布的这一种对于神经网络就会更加有价值.



Batch normalization让每一层的值在有效的范围内传递下去.

## 实现

为了实现 Batch Normalization, 要对每一层的代码进行修改, 给函数加上 norm 参数, 表示是否是 Batch Normalization 层

*def* add\_layer(inputs, in\_size, out\_size, activation\_function=*None*, norm=*False*):

然后每层的Wx\_plus\_b需要进行一次batch normalize的步骤,这样输出到 activation 的Wx\_plus\_b就已经被 normalize 过了

*if* norm: # 判断书否是 BN 层  
 fc\_mean, fc\_var = tf.nn.moments(  
 Wx\_plus\_b,  
 axes=[0], # 想要 normalize 的维度, [0] 代表 batch 维度  
 # 如果是图像数据, 可以传入 [0, 1, 2], 相当于求[batch, height, width] 的均值/方差, 注意不要加入 channel 维度  
 )  
 scale = tf.Variable(tf.ones([out\_size]))  
 shift = tf.Variable(tf.zeros([out\_size]))  
 epsilon = 0.001  
 Wx\_plus\_b = tf.nn.batch\_normalization(Wx\_plus\_b, fc\_mean, fc\_var, shift, scale, epsilon)  
 # 上面那一步, 在做如下事情:  
 # Wx\_plus\_b = (Wx\_plus\_b - fc\_mean) / tf.sqrt(fc\_var + 0.001)  
 # Wx\_plus\_b = Wx\_plus\_b \* scale + shift

# 迁移学习

(transfer learning)