# 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。

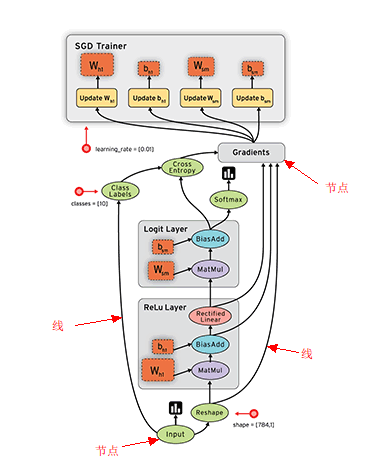
TensorFlow也能够自动地计算需要优化的变量的梯度，使得模型有更好的表现。这是由于Graph是简单数学表达式的结合，因此整个图的梯度可以用链式法则推导出来。

## 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()

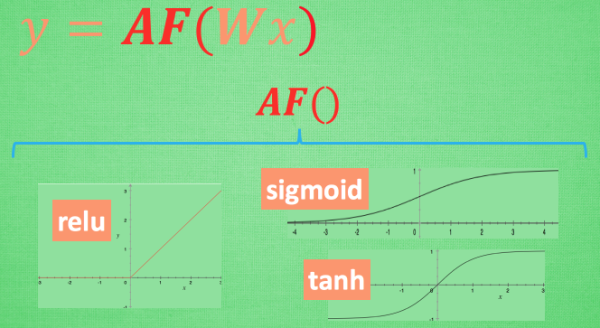
定义了变量以后, 一定要定义 init = tf.initialize\_all\_variables() 初始化变量，使用sess.run(init)激活init.

### 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

### 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

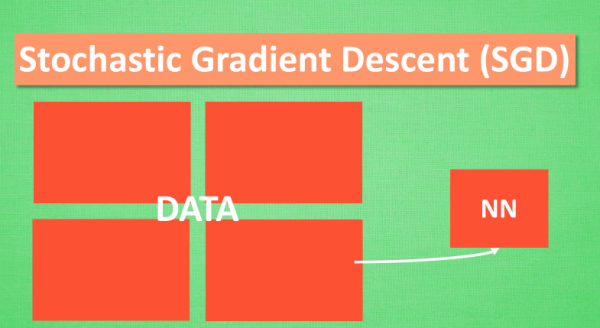
# 散点图描述真实数据  
fig = plt.figure()  
ax = fig.add\_subplot(1,1,1)  
ax.scatter(x\_data, y\_data)

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()

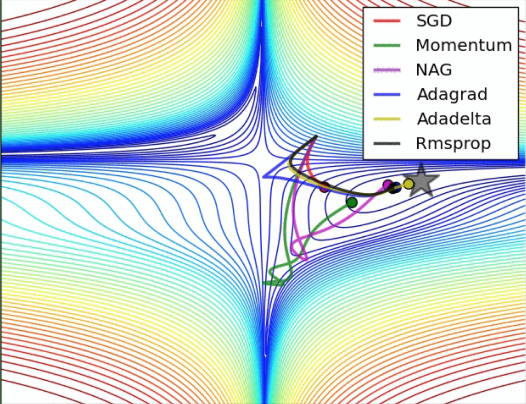
## 加速神经网络训练

### 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.

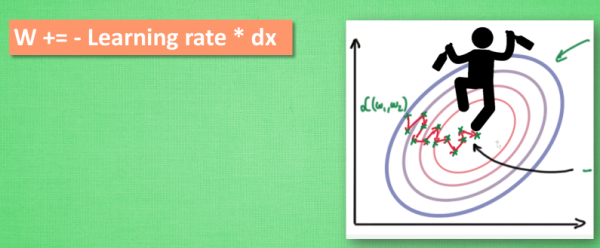
如果把这些数据拆分成小批小批的, 然后再分批不断放入 NN 中计算, 这就是SGD 的正确打开方式. 每次使用批数据, 虽然不能反映整体数据的情况, 不过却很大程度上加速了 NN 的训练过程, 而且也不会丢失太多准确率.



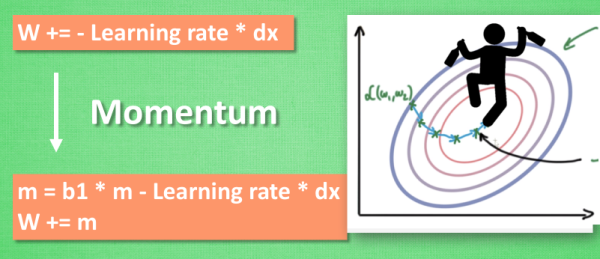
SGD 并不是最快速的训练方法, 红色的线是 SGD, 但它到达学习目标的时间是在这些方法中最长的一种

### 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.

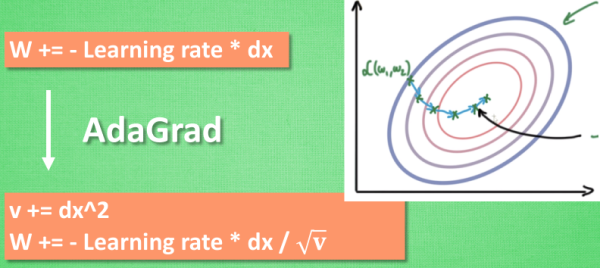


所以我们把这个人从平地上放到了一个斜坡上, 只要他往下坡的方向走一点点, 由于向下的惯性, 他不自觉地就一直往下走, 走的弯路也变少了. 这就是 Momentum 参数更新.



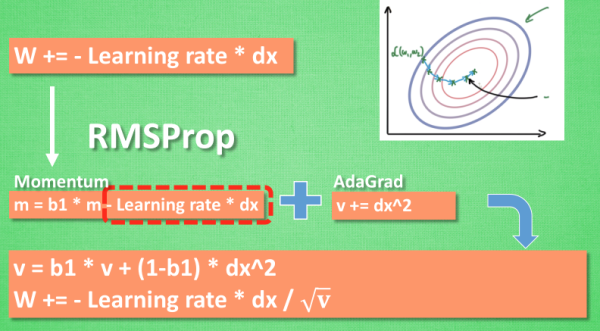
### AdaGrad

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.



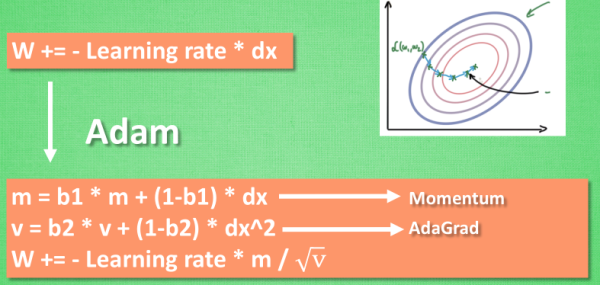
### RMSProp

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。

1. 编辑layer框架

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

#and so on…

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. 绘制loss和train

# 绘制loss  
*with* tf.name\_scope('loss'):  
 loss=tf.reduce\_mean(tf.reduce\_sum(tf.square(ys-prediction),reduction\_indices=[1]))  
# 绘制train  
*with* tf.name\_scope('train'):  
 train\_step=tf.train.GradientDescentOptimizer(0.1).minimize(loss)

1. 绘图保存

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  
writer = tf.summary.FileWriter("logs/", sess.graph)  
sess.run(init)

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

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')

## 卷积神经网络

**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）