Tensorflow 指南

2015-12-16

目录

第一章　起步

第二章　基础教程

2.1　综述

2.2 MNIST 之机器学习入门

这个教程的目标读者是对机器学习和 TensorFlow 都不太了解的新手。如果你已经了解 MNIST 和 softmax 回归 (softmax regression) 的相关知识，你可以阅读这个快速上手教程。

当我们开始学习编程的时候，第一件事往往是学习打印“Hello World”。就好比编程入门有 Hello World，机器学习入门有 MNIST。MNIST 是一个入门级的计算机视觉数据集，它包含各种手写数字图片：

它也包含每一张图片对应的标签，告诉我们这个是数字几。比如，上面这四张图片的标签分别是 5,0,4,1。

在此教程中，我们将训练一个机器学习模型用于预测图片里面的数字。我们的目的不是要设计一个世界一流的复杂模型 – 尽管我们会在之后给你源代码去实现一流的预测模型 – 而是要介绍下如何使用 TensorFlow。所以，我们这里会从一个很简单的数学模型开始，它叫做 Softmax Regression。

对应这个教程的实现代码很短，而且真正有意思的内容只包含在三行代码里面。但是，去理解包含在这些代码里面的设计思想是非常重要的：TensorFlow 工作流程和机器学习的基本概念。因此，这个教程会很详细地介绍这些代码的实现原理。

2.2.1 MNIST 数据集

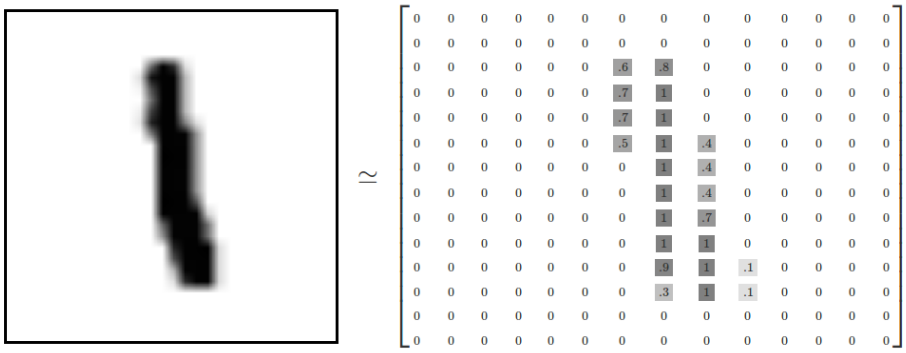
MNIST 数据集的官网是Yann LeCun's website。在这里，我们提供了一份 python 源代码用于自动下载和安装这个数据集。你可以下载这段代码，然后用下面的代码导入到你的项目里面，也可以直接复制粘贴到你的代码文件里面。

```
import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

下载下来的数据集被分成两部分：60000 行的训练数据集（‘mnist.train‘）和 10000 行的测试数据集（‘mnist.test‘）。这样的切分很重要，在机器学习模型设计时必须有一个单独的测试数据集不用于训练而是用来评估这个模型的性能，从而更加容易把设计的模型推广到其他数据集上（泛化）。

正如前面提到的一样，每一个 MNIST 数据单元有两部分组成：一张包含手写数字的图片和一个对应的标签。我们把这些图片设为“xs”，把这些标签设为“ys”。训练数据集和测试数据集都包含 xs 和 ys，比如训练数据集的图片是 ‘mnist.train.images‘ ，训练数据集的标签是 ‘mnist.train.labels‘。

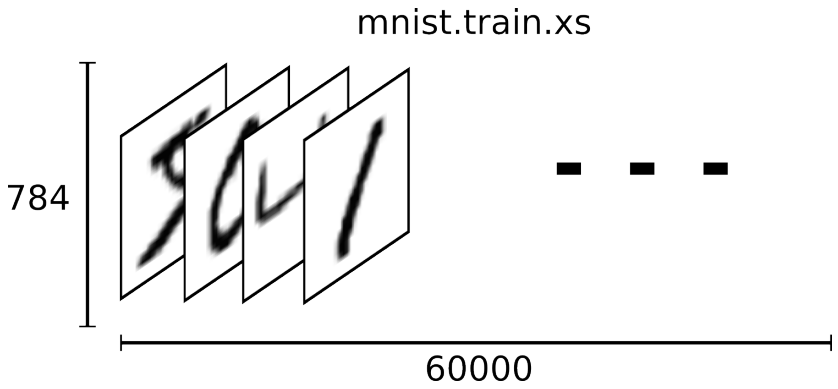
每一张图片包含 28×28 像素。我们可以用一个数字数组来表示这张图片：



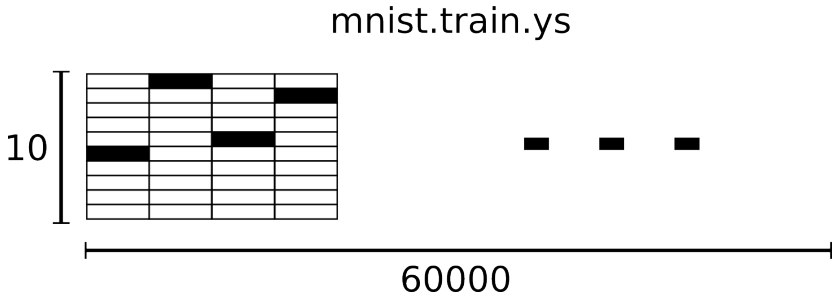
我们把这个数组展开成一个向量，长度是 $28 \times 28 = 784$。如何展开这个数组（数字间的顺序）不重要，只要保持各个图片采用相同的方式展开。从这个角度来看，MNIST 数据集的图片就是在 784 维向量空间里面的点, 并且拥有比较复杂的结构 (提醒: 此类数据的可视化是计算密集型的)。

展平图片的数字数组会丢失图片的二维结构信息。这显然是不理想的，最优秀的计算机视觉方法会挖掘并利用这些结构信息，我们会在后续教程中介绍。但是在这个教程中我们忽略这些结构，所介绍的简单数学模型，softmax 回归 (softmax regression)，不会利用这些结构信息。

因此，在 MNIST 训练数据集中，mnist.train.images 是一个形状为 [60000, 784] 的张量，第一个维度数字用来索引图片，第二个维度数字用来索引每张图片中的像素点。在此张量里的每一个元素，都表示某张图片里的某个像素的强度值，值介于 0 和 1 之间。



相对应的 MNIST 数据集的标签是介于 0 到 9 的数字，用来描述给定图片里表示的数字。为了用于这个教程，我们使标签数据是"one-hot vectors"。一个 one-hot 向量除了某一位的数字是 1 以外其余各维度数字都是 0。所以在此教程中，数字 n 将表示成一个只有在第 n 维度（从 0 开始）数字为 1 的 10 维向量。比如，标签 0 将表示成 ([1,0,0,0,0,0,0,0,0,0])。因此，mnist.train.labels 是一个 [60000, 10] 的数字矩阵。



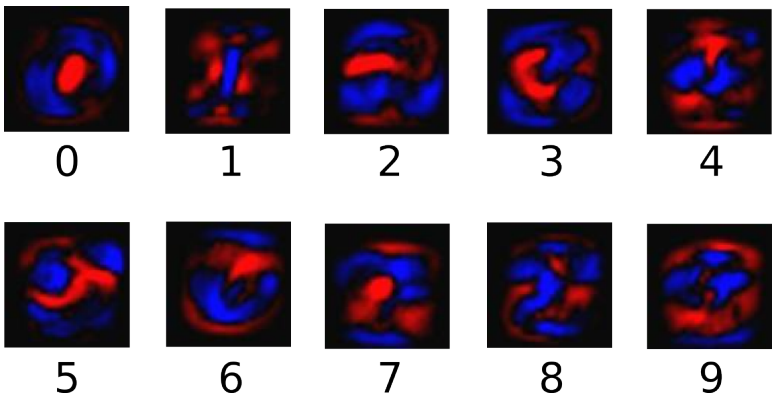
现在，我们准备开始真正的建模之旅啦！

2.2.2 Softmax 回归介绍

我们知道 MNIST 的每一张图片都表示一个数字，从 0 到 9。我们希望得到给定图片代表每个数字的概率。比如说，我们的模型可能推测一张包含 9 的图片代表数字 9 的概率是 80% 但是判断它是 8 的概率是 5%（因为 8 和 9 都有上半部分的小圆），然后给予它代表其他数字的概率更小的值。

这是一个使用 softmax 回归（softmax regression）模型的经典案例。softmax 模型可以用来给不同的对象分配概率。即使在之后，我们训练更加精细的模型时，最后一步也需要用 softmax 来分配概率。

softmax 回归（softmax regression）分两步：首先，为了得到一张给定图片属于某个特定数字类的证据（evidence），我们对图片像素值进行加权求和。如果这个像素具有很强的证据说明这张图片不属于该类，那么相应的权值为负数，相反如果这个像素拥有有利的证据支持这张图片属于这个类，那么权值是正数。下面的图片显示了一个模型学习到的图片上每个像素对于特定数字类的权值。红色代表负数权值，蓝色代表正数权值。



我们也需要加入一个额外的偏置量（bias），因为输入往往会带有一些无关的干扰量。因此对于给定的输入图片 x 它代表的是数字 x 的证据可以表示为

$$evidence_i = \sum_j W_{i,j} x_j + b_i \tag{2.1}$$

其中，W_i 代表权重，b_i 代表第 i 类的偏置量，j 代表给定图片 x 的像素索引用于像素求

和。然后用 softmax 函数可以把这些证据转换成概率 y:

$$y = softmax(evidence) \tag{2.2}$$

这里的 softmax 可以看成是一个激励（activation）函数或是链接（link）函数，把我们定义的线性函数的输出转换成我们想要的格式，也就是关于 10 个数字类的概率分布。因此，给定一张图片，它对于每一个数字的吻合度可以被 softmax 函数转换成为一个概率值。softmax 函数可以定义为：

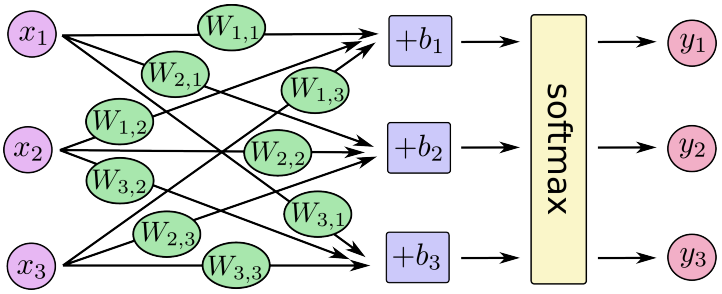
$$softmax(x) = normalize(exp(x)) \tag{2.3}$$

展开等式右边的子式，可以得到：

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{2.4}$$

但是更多的时候把 softmax 模型函数定义为前一种形式：把输入值当成幂指数求值，再正则化这些结果值。这个幂运算表示，更大的证据对应更大的假设模型（hypothesis）里面的乘数权重值。反之，拥有更少的证据意味着在假设模型里面拥有更小的乘数系数。假设模型里的权值不可以是 0 值或者负值。Softmax 然后会正则化这些权重值，使它们的总和等于 1，以此构造一个有效的概率分布。（更多的关于 Softmax 函数的信息，可以参考 Michael Nieslen 的书里面的这个部分，其中有关于 softmax 的可交互式的可视化解释。）

对于 softmax 回归模型可以用下面的图解释，对于输入的 xs 加权求和，再分别加上一个偏置量，最后再输入到 softmax 函数中：



如果把它写成一个方程，可以得到：

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left(\begin{matrix} W_{1,1}x_1 + W_{1,2}x_1 + W_{1,3}x_1 + b_1 \\ W_{2,1}x_2 + W_{2,2}x_2 + W_{2,3}x_2 + b_2 \\ W_{3,1}x_3 + W_{3,2}x_3 + W_{3,3}x_3 + b_3 \end{matrix} \right)$$

我们也可以用向量表示这个计算过程：用矩阵乘法和向量相加。这有助于提高计算效率（也是一种更有效的思考方式）。

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

更进一步，可以写成更加紧凑的方式：

$$y = softmax(W_x + b) \tag{2.5}$$

2.2.3 实现回归模型

为了用 python 实现高效的数值计算，我们通常会使用函数库，比如 NumPy，会把类似矩阵乘法这样的复杂运算使用其他外部语言实现。不幸的是，从外部计算切换回 Python 的每一个操作，仍然是一个很大的开销。如果你用 GPU 来进行外部计算，这样的开销会更大。用分布式的计算方式，也会花费更多的资源用来传输数据。

TensorFlow 也把复杂的计算放在 python 之外完成，但是为了避免前面说的那些开销，它做了进一步完善。TensorFlow 不单独地运行单一的复杂计算，而是让我们可以先用图描述一系列可交互的计算操作，然后全部一起在 Python 之外运行。（这样类似的运行方式，可以在不少的机器学习库中看到。）

使用 TensorFlow 之前，首先导入它：

```
import tensorflow as tf
```

我们通过操作符号变量来描述这些可交互的操作单元，可以用下面的方式创建一个：

```
x = tf.placeholder("float", [None, 784])
```

x 不是一个特定的值，而是一个占位符 placeholder，我们在 TensorFlow 运行计算时输入这个值。我们希望能够输入任意数量的 MNIST 图像，每一张图展平成 784 维的向量。我们用 2 维的浮点数张量来表示这些图，这个张量的形状是 [None，784]。（这里的 None 表示此张量的第一个维度可以是任何长度的。）

我们的模型也需要权重值和偏置量，当然我们可以把它们当做是另外的输入（使用占位符），但 TensorFlow 有一个更好的方法来表示它们：Variable 。一个 Variable 代表一个可修改的张量，存在在 TensorFlow 的用于描述交互性操作的图中。它们可以用于计算输入值，也可以在计算中被修改。对于各种机器学习应用，一般都会有模型参数，可以用 Variable 表示。

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

我们赋予 tf.Variable 不同的初值来创建不同的 Variable：在这里，我们都用全为零的张量来初始化 W 和 b 。因为我们要学习 W 和 b 的值，它们的初值可以随意设置。

注意，W 的维度是 [784，10]，因为我们想要用 784 维的图片向量乘以它以得到一个 10 维的证据值向量，每一位对应不同数字类。b 的形状是 [10]，所以我们可以直接把它加到输出上面。

现在，可以实现我们的模型了，只需以下一行代码：

```
y = tf.nn.softmax(tf.matmul(x,W) + b)
```

首先，我们用 tf.matmul(X，W) 表示 x 乘以 W，对应之前等式里面的 W_x，这里 x 是一个 2 维张量拥有多个输入。然后再加上 b，把和输入到 tf.nn.softmax 函数里面。

至此，我们先用了几行简短的代码来设置变量，然后只用了一行代码来定义我们的模型。TensorFlow 不仅仅可以使 softmax 回归模型计算变得特别简单，它也用这种非常灵活的方式来描述其他各种数值计算，从机器学习模型对物理学模拟仿真模型。一旦被定义好之后，我们的模型就可以在不同的设备上运行：计算机的 CPU，GPU，甚至是手机！

2.2.4 训练模型

为了训练我们的模型，我们首先需要定义一个指标来评估这个模型是好的。其实，在机器学习，我们通常定义指标来表示一个模型是坏的，这个指标称为成本（cost）或损失（loss），然后尽量最小化这个指标。但是，这两种方式是相同的。

一个非常常见的，非常漂亮的成本函数是“交叉熵”（cross-entropy）。交叉熵产生于信息论里面的信息压缩编码技术，但是它后来演变成为从博弈论到机器学习等其他领域里的重要技术手段。它的定义如下：

$$H_{y'}(u) = -\sum_i y_{i'} log(y_i) \tag{2.6}$$

y 是我们预测的概率分布,y' 是实际的分布（我们输入的 one-hot vector)。比较粗糙的理解是，交叉熵是用来衡量我们的预测用于描述真相的低效性。更详细的关于交叉熵的解释超出本教程的范畴，但是你很有必要好好理解它。

为了计算交叉熵，我们首先需要添加一个新的占位符用于输入正确值：

```
y = tf.placeholder("float", [None,10])
```

然后我们可以用

$$-\sum y' log(y) \tag{2.7}$$

计算交叉熵:

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y))
```

首先，用 tf.log 计算 y 的每个元素的对数。接下来，我们把 y_ 的每一个元素和 tf.log(y_) 的对应元素相乘。最后，用 tf.reduce_sum 计算张量的所有元素的总和。（注意，这里的交叉熵不仅仅用来衡量单一的一对预测和真实值，而是所有 100 幅图片的交叉熵的总和。对于 100 个数据点的预测表现比单一数据点的表现能更好地描述我们的模型的性能。

现在我们知道我们需要我们的模型做什么啦，用 TensorFlow 来训练它是非常容易的。因为 TensorFlow 拥有一张描述你各个计算单元的图，它可以自动地使用反向传播

算法 (backpropagation algorithm) 来有效地确定你的变量是如何影响你想要最小化的那个成本值的。然后，TensorFlow 会用你选择的优化算法来不断地修改变量以降低成本。

```
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

在这里，我们要求 TensorFlow 用梯度下降算法（gradient descent algorithm）以 0.01 的学习速率最小化交叉熵。梯度下降算法（gradient descent algorithm）是一个简单的学习过程，TensorFlow 只需将每个变量一点点地往使成本不断降低的方向移动。当然 TensorFlow 也提供了其他许多优化算法：只要简单地调整一行代码就可以使用其他的算法。

TensorFlow 在这里实际上所做的是，它会在后台给描述你的计算的那张图里面增加一系列新的计算操作单元用于实现反向传播算法和梯度下降算法。然后，它返回给你的只是一个单一的操作，当运行这个操作时，它用梯度下降算法训练你的模型，微调你的变量，不断减少成本。

现在，我们已经设置好了我们的模型。在运行计算之前，我们需要添加一个操作来初始化我们创建的变量：

```
init = tf.initialize_all_variables()
```

现在我们可以在一个 Session 里面启动我们的模型，并且初始化变量：

```
sess = tf.Session()
sess.run(init)
```

然后开始训练模型，这里我们让模型循环训练 1000 次！

```
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

该循环的每个步骤中，我们都会随机抓取训练数据中的 100 个批处理数据点，然后我们用这些数据点作为参数替换之前的占位符来运行 train_step。

使用一小部分的随机数据来进行训练被称为随机训练（stochastic training）- 在这里更确切的说是随机梯度下降训练。在理想情况下，我们希望用我们所有的数据来进行每一步的训练，因为这能给我们更好的训练结果，但显然这需要很大的计算开销。所以，每一次训练我们可以使用不同的数据子集，这样做既可以减少计算开销，又可以最大化地学习到数据集的总体特性。

2.2.5 评估我们的模型

那么我们的模型性能如何呢？

首先让我们找出那些预测正确的标签。tf.argmax 是一个非常有用的函数，它能给你在一个张量里沿着某条轴的最高条目的索引值。比如，tf.argmax(y,1) 是模型认为每个输入最有可能对应的那些标签，而 tf.argmax(y_,1) 代表正确的标签。我们可以用 tf.equal 来检测我们的预测是否真实标签匹配。

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

这行代码会给我们一组布尔值。为了确定正确预测项的比例，我们可以把布尔值转换成浮点数，然后取平均值。例如，[True, False, True, True] 会变成 [1,0,1,1] ，取平均值后得到 0.75 .

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

最后，我们计算所学习到的模型在测试数据集上面的正确率。

```
print sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels})
```

最终结果值应该大约是 91%。

这个结果好吗？嗯，并不太好。事实上，这个结果是很差的。这是因为我们仅仅使用了一个非常简单的模型。不过，做一些小小的改进，我们就可以得到 97% 的正确率。最好的模型甚至可以获得超过 99.7% 的准确率！（想了解更多信息，可以看看这个关于各种模型的性能对比列表。）

比结果更重要的是，我们从这个模型中学习到的设计思想。不过，如果你仍然对这里的结果有点失望，可以查看下一个教程，在那里你将学到如何用 FensorFlow 构建更加复杂的模型以获得更好的性能！

原文地址：MNIST For ML Beginners 翻译：linbojin 校对：

2.3 深入 MNIST

TensorFlow 是一个做大规模数值计算的强大库。其中一个特点就是它能够实现和训练深度神经网络。在这一小节里，我们将会学习在 MNIST 上构建深度卷积分类器的基本步骤。

这个教程假设你已经熟悉神经网络和 *MNIST* 数据集。如果你尚未了解，请查看新手指南.

2.3.1 安装

在创建模型之前，我们会先加载 MNIST 数据集，然后启动一个 TensorFlow 的 session。

加载 MINIST 数据

为了方便起见，我们已经准备了一个脚本来自动下载和导入 MNIST 数据集。它会自动创建一个'MNIST_data' 的目录来存储数据。

```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

开始 TensorFlow 的交互会话

Tensorflow 基于一个高效的 C++ 模块进行运算。与这个模块的连接叫做 session。一般而言，使用 TensorFlow 程序的流程是先创建一个图，然后在 session 中加载它。

这里，我们使用更加方便的 InteractiveSession 类。通过它，你可以更加灵活地构建你的代码。它能让你在运行图的时候，插入一些构建计算图的操作。这能给使用交互式文本 shell 如 iPython 带来便利。如果你没有使用 InteractiveSession 的话，你需要在开始 session 和加载图之前，构建整个计算图。

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

计算图

传统的计算行为中，为了更高效地在 Python 里进行数值计算，我们一般会使用像 NumPy 这样用其他语言编写的 lib，在 Python 外完成这些费时的操作（例如矩阵运算）。可是，每一步操作依然会经常在 Python 和第三方 lib 之间切换。这些操作很蛋疼，特别是当你想在 GPU 上进行计算，又或者想使用分布式的做法的时候。这些情况下数据传输代价高昂。

在 TensorFlow 中，也有 Python 与外界的频繁操作。但是它在这一方面，做了进一步的改良。TensorFlow 构建一个交互操作的图，作为一个整体在 Python 外运行，而不

是以代价高昂的单个交互操为单位在 Python 外运行。这与 Theano、Torch 的做法很相似。

所以，这部分 Python 代码，目的是构建这个在外部运行的计算图，并安排这个计算图的哪一部分应该被运行。详细请阅读计算图部分的基本用法。

2.3.2 构建 Softmax Regression 模型

在这小节里，我们将会构建一个一层线性的 softmax regression 模型。下一节里，我们会扩展到多层卷积网络。

占位符

我们先来创建计算图的输入（图片）和输出（类别）。

```
x = tf.placeholder("float", shape=[None, 784])
y_ = tf.placeholder("float", shape=[None, 10])
```

这里的 x 和 y 并不是具体值，他们是一个 placeholder，是一个变量，在 TensorFlow 运行计算的时候使用。

输入图片 x 是浮点数 2 维张量。这里，定义它的 shape 为 [None, 784]，其中 784 是单张展开的 MNIST 图片的维度数。shape 的第一维输入指代一个 batch 的大小，None，可为任意值。输出值 y_ 也是一个 2 维张量，其中每一行为一个 10 维向量代表对应 MNIST 图片的分类。

虽然 placeholder 的 shape 参数是可选的，但有了它，TensorFlow 能够自动捕捉因数据维度不一致导致的错误。

Variables

我们现在为模型定义权重 W 和偏置 b。它们可以被视作是额外的输入量，但是 TensorFlow 有一个更好的方式来处理：Variable。一个 Variable 代表着在 TensorFlow 计算图中的一个值，它是能在计算过程中被读取和修改的。在机器学习的应用过程中，模型参数一般用 Variable 来表示。

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

我们在调用 tf.Variable 的时候传入初始值。在这个例子里，我们把 W 和 b 都初始化为零向量。W 是一个 784x10 的矩阵（因为我们有 784 个特征和 10 个输出值）。b 是一个 10 维的向量（因为我们有 10 个分类）。

Variable 需要在 session 之前初始化，才能在 session 中使用。初始化需要初始值（本例当中是全为零）传入并赋值给每一个 Variable。这个操作可以一次性完成。

```
sess.run(tf.initialize_all_variables())
```

预测分类与损失函数

现在我们可以实现我们的 regression 模型了。这只需要一行！我们把图片 x 和权重矩阵 W 相乘，加上偏置 b，然后计算每个分类的 softmax 概率值。

```
y = tf.nn.softmax(tf.matmul(x,W) + b)
```

在训练中最小化损失函数同样很简单。我们这里的损失函数用目标分类和模型预测分类之间的交叉熵。

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y))
```

注意，tf.reduce_sum 把 minibatch 里的每张图片的交叉熵值都加起来了。我们计算的交叉熵是指整个 minibatch 的。

2.3.3 训练模型

我们已经定义好了模型和训练的时候用的损失函数，接下来使用 TensorFlow 来训练。因为 TensorFlow 知道整个计算图，它会用自动微分法来找到损失函数对于各个变量的梯度。TensorFlow 有大量内置的优化算法这个例子中，我们用最速下降法让交叉熵下降，步长为 0.01。

```
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

这一行代码实际上是用来往计算图上添加一个新操作，其中包括计算梯度，计算每个参数的步长变化，并且计算出新的参数值。

train_step 这个操作，用梯度下降来更新权值。因此，整个模型的训练可以通过反复地运行 train_step 来完成。

```
for i in range(1000):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

每一步迭代，我们都会加载 50 个训练样本，然后执行一次 train_step，使用 feed_dict，用训练数据替换 placeholder 向量 x 和 y_。

注意，在计算图中，你可以用 feed_dict 来替代任何张量，并不仅限于替换 placeholder。

评估模型

我们的模型效果怎样？

首先，要先知道我们哪些 label 是预测正确了。tf.argmax 是一个非常有用的函数。它会返回一个张量某个维度中的最大值的索引。例如，tf.argmax(y,1) 表示我们模型对每个输入的最大概率分类的分类值。而 tf.argmax(y_,1) 表示真实分类值。我们可以用 tf.equal 来判断我们的预测是否与真实分类一致。

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

这里返回一个布尔数组。为了计算我们分类的准确率，我们将布尔值转换为浮点数来代表对、错，然后取平均值。例如：[True, False, True, True] 变为 [1,0,1,1]，计算出平均值为 0.75。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

最后，我们可以计算出在测试数据上的准确率，大概是 91%。

```
print accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels})
```

2.3.4 构建多层卷积网络模型

在 MNIST 上只有 91% 正确率，实在太糟糕。在这个小节里，我们用一个稍微复杂的模型：卷积神经网络来改善效果。这会达到大概 99.2% 的准确率。虽然不是最高，但是还是比较让人满意。

权重初始化

在创建模型之前，我们先来创建权重和偏置。一般来说，初始化时应加入轻微噪声，来打破对称性，防止零梯度的问题。因为我们用的是 ReLU，所以用稍大于 0 的值来初始化偏置能够避免节点输出恒为 0 的问题（dead neurons）。为了不在建立模型的时候反复做初始化操作，我们定义两个函数用于初始化。

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

卷积和池化

TensorFlow 在卷积和池化上有很强的灵活性。我们怎么处理边界？步长应该设多大？在这个实例里，我们会一直使用 vanilla 版本。我们的卷积使用 1 步长（stride size），0 边距（padding size）的模板，保证输出和输入是同一个大小。我们的池化用简单传统的 2x2 大小的模板做 max pooling。为了代码更简洁，我们把这部分抽象成一个函数。

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME'
    )
```

第一层卷积

现在我们可以开始实现第一层了。它由一个卷积接一个 max pooling 完成。卷积在每个 5x5 的 patch 中算出 32 个特征。权重是一个 [5, 5, 1, 32] 的张量，前两个维度是 patch 的大小，接着是输入的通道数目，最后是输出的通道数目。输出对应一个同样大小的偏置向量。

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

为了用这一层，我们把 x 变成一个 4d 向量，第 2、3 维对应图片的宽高，最后一维代表颜色通道。

```
x_image = tf.reshape(x, [-1,28,28,1])
```

我们把 x_image 和权值向量进行卷积相乘，加上偏置，使用 ReLU 激活函数，最后 max pooling。

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

第二层卷积

为了构建一个更深的网络，我们会把几个类似的层堆叠起来。第二层中，每个 5x5 的 patch 会得到 64 个特征。

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

密集连接层

现在，图片降维到 7x7，我们加入一个有 1024 个神经元的全连接层，用于处理整个图片。我们把池化层输出的张量 reshape 成一些向量，乘上权重矩阵，加上偏置，使用 ReLU 激活。

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Dropout

为了减少过拟合，我们在输出层之前加入 dropout。我们用一个 placeholder 来代表一个神经元在 dropout 中被保留的概率。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。

```
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

输出层

最后，我们添加一个 softmax 层，就像前面的单层 softmax regression 一样。

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

训练和评估模型

这次效果又有多好呢？我们用前面几乎一样的代码来测测看。只是我们会用更加复杂的 ADAM 优化器来做梯度最速下降，在 feed_dict 中加入额外的参数 keep_prob 来控制 dropout 比例。然后每 100 次迭代输出一次日志。

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print "step %d, training accuracy %g"%(i, train_accuracy)
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print "test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0})
```

以上代码，在最终测试集上的准确率大概是 99.2

目前为止，我们已经学会了用 TensorFlow 来快速和简易地搭建、训练和评估一个复杂一点儿的深度学习模型。

原文地址：Deep MNIST for Experts 翻译：chenweican 校对：HongyangWang

2.4 TensorFlow 运作方式

2.5 卷积神经网络

2.5.1 概述

2.5.2 代码组织

2.5.3 CIFAR-10 模型

2.5.4 开始执行并训练模型

2.5.5 模型评估

2.6 Vector Representations of Words

2.7 循环神经网络

2.8 曼德博 (Mandelbrot) 集合

2.9 偏微分方程

2.10 MNIST 数据集下载

第三章　运作方式

第四章　资源

第五章　其他