每周学习进展阶段汇报

----CS231n Stanford University深度学习基础课程

汇报人 : 胡小婉

时间段 : 2018年3月5日(周一)至2018年3月11日(周日)

[1. Neural Nets\_1 3](#_Toc1981059282)

[1.1 简介 3](#_Toc1360266428)

[1.2 单个神经元建模 3](#_Toc1011036526)

[1.3 激活函数 4](#_Toc318539184)

[1.3.1 sigmoid函数 4](#_Toc924559686)

[1.3.2 tanh函数 4](#_Toc1947351617)

[1.3.3 ReLU函数 4](#_Toc1522463463)

[1.3.4 其他 4](#_Toc1346630889)

[1.4 层组织 5](#_Toc697237812)

[1.4.1 简介 5](#_Toc1334708837)

[1.4.2 确定网络尺寸 5](#_Toc1652220965)

[1.4.3. 设置层的数量和尺寸 5](#_Toc943417300)

[2. Neural Nets\_2 6](#_Toc1896441074)

[2.1 数据预处理 6](#_Toc1104811532)

[2.1.1 均值减法（Mean subtraction） 6](#_Toc304453991)

[2.1.2 归一化（Normalization） 6](#_Toc307080979)

[2.1.3 PCA和白化 6](#_Toc1665854936)

[2.2 权重初始化 7](#_Toc1543427820)

[2.3 偏置（biases）的初始化 7](#_Toc1405325386)

[2.4 批量归一化 7](#_Toc1149912269)

[2.5 正则化 Regularization 7](#_Toc379259448)

[2.5.1 L2正则化 8](#_Toc616587281)

[2.5.2 L1正则化 8](#_Toc1632771059)

[2.5.3 最大范式约束（Max norm constraints） 8](#_Toc1136030131)

[2.5.4 随机失活（Dropout） 8](#_Toc643826878)

[Tips：随机失活dropout的意义 8](#_Toc778670707)

[2.6 损失函数 10](#_Toc827630747)

[2.6.1 SVM 10](#_Toc1893787662)

[2.6.2 Softmax 10](#_Toc619005426)

[2.6.3 回归问题 10](#_Toc1889083443)

[3. Neural Nets\_3 11](#_Toc1502176019)

[3.1 梯度检查 11](#_Toc452581060)

[3.1.1 使用中心化公式 11](#_Toc1101866223)

[3.1.2 使用相对误差来比较 11](#_Toc365728897)

[3.1.3 目标函数的不可导点（kinks） 11](#_Toc771120245)

[3.1.4 不要让正则化吞没数据 11](#_Toc2026425909)

[3.2 合理性检查 12](#_Toc165596866)

[3.3 检查整个学习过程 12](#_Toc146100060)

[3.4 训练集和验证集准确率 12](#_Toc1225573150)

[3.5 权重更新比例 13](#_Toc862834678)

[4. ConvNet notes（卷积神经网络） 13](#_Toc1480808897)

[4.1 结构概述 13](#_Toc730310467)

[4.2 各种层 14](#_Toc1806251979)

[4.2.1 卷积层 14](#_Toc1229766324)

[Tips： 参数共享 15](#_Toc1835121999)

[4.2.2 汇聚层（Pooling） 16](#_Toc2110705970)

[4.2.3 归一化层 17](#_Toc1536847303)

[4.2.4 全连接层 17](#_Toc1353493288)

[4.3 卷积神经网络的结构 17](#_Toc1506650142)

[4.3.1 层的排列规律 17](#_Toc794689041)

[4.3.2 层的尺寸设置规律 17](#_Toc355921909)

[5. TensorFlow中文社区入门学习 18](#_Toc1885909591)

[5.1 基本使用 18](#_Toc1411276322)

[5.2 综述 18](#_Toc1988692968)

[5.3 计算图 18](#_Toc874456074)

[5.4 构建图 18](#_Toc2055103201)

[5.5 在一个会话中启动图 19](#_Toc619880028)

[5.6 交互式使用 20](#_Toc1702086821)

[5.7 Tensor 20](#_Toc1801407215)

[5.7.1 变量 20](#_Toc1238885454)

[5.7.2 Fetch 21](#_Toc1443686616)

[5.7.3 Feed 21](#_Toc1156099586)

[5.8 训练模型 22](#_Toc1691466515)

[5.9 评估模型 22](#_Toc398069191)

[6. Tensorflow构建一个多层卷积网络 23](#_Toc1521828483)

[6.1 权重初始化&&卷积和池化 23](#_Toc315103112)

[6.2 第一层卷积 23](#_Toc277011453)

[6.3 第二层卷积&&密集连接层 24](#_Toc1687425350)

[6.4 Dropout&&输出 24](#_Toc461203172)

[6.5 训练和评估模型 25](#_Toc1502584603)

[7. Tensorflow构建一个卷积神经网络 25](#_Toc402776380)

[Assignment #2 26](#_Toc1942012070)

[Q1: Fully-connected Neural Network 26](#_Toc85411423)

[Q2: Batch Normalization 33](#_Toc61544711)

[Q3: Dropout 37](#_Toc1024294746)

[Q4: Convolutional Networks 38](#_Toc1920533422)

[8. 其他概念与函数 48](#_Toc24767034)

[8.1 CPU vs GPU 48](#_Toc413658401)

[8.2 numpy.std() 50](#_Toc1126543062)

[8.3 np.random.normal()正态分布 50](#_Toc1531417176)

[8.4 Python str() 函数 50](#_Toc1208347442)

[8.5 numpy.logspace用法 51](#_Toc1482464972)

# 1. Neural Nets\_1

## 1.1 简介

在使用数据库CIFAR-10的案例中，x是一个[3072x1]的列向量，W是一个[10x3072]的矩阵，所以输出的评分是一个包含10个分类评分的向量。

神经网络算法则不同，它的计算公式是s=W\_2max(0,W\_1x)。其中W\_1的含义是这样的：举个例子来说，它可以是一个[100x3072]的矩阵，其作用是将图像转化为一个100维的过渡向量。函数max(0,-)是非线性的，它会作用到每个元素。最终，矩阵W\_2的尺寸是[10x100]，因此将得到10个数字，这10个数字可以解释为是分类的评分。

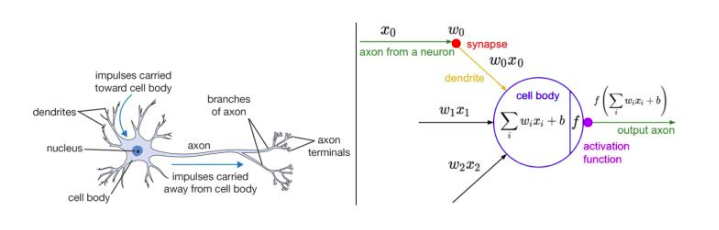
参数W\_1,W\_2将通过随机梯度下降来学习到，他们的梯度在反向传播过程中，通过链式法则来求导计算得出。

一个三层的神经网络可以类比地看做s=W\_3max(0,W\_2max(0,W\_1x))，其中W\_1,W\_2,W\_3是需要进行学习的参数。中间隐层的尺寸是网络的超参数

## 1.2 单个神经元建模

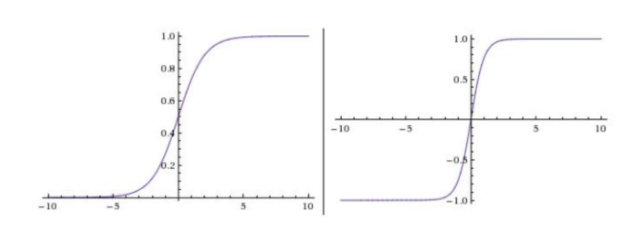
每个神经元都从它的树突获得输入信号，然后沿着它唯一的轴突（axon）产生输出信号。轴突在末端会逐渐分枝，通过突触和其他神经元的树突相连。

在基本模型中，树突将信号传递到细胞体，信号在细胞体中相加。如果最终之和高于某个阈值，那么神经元将会激活，向其轴突输出一个峰值信号。在计算模型中，我们假设峰值信号的准确时间点不重要，是激活信号的频率在交流信息。基于这个速率编码的观点，将神经元的激活率建模为激活函数（activation function）



## 1.3 激活函数

### 1.3.1 sigmoid函数



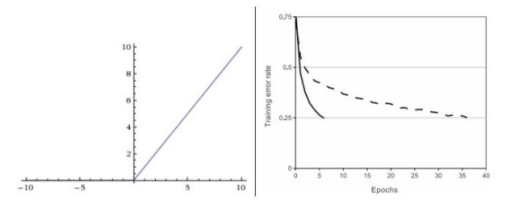
缺点：1. Sigmoid函数饱和使梯度消失

2. Sigmoid函数的输出不是零中心的

### 1.3.2 tanh函数

将实数值压缩到[-1,1]之间。和sigmoid神经元一样，它也存在饱和问题，但是和sigmoid神经元不同的是，它的输出是零中心的。

### 1.3.3 ReLU函数



使用ReLU有以下一些优缺点：

优点：

1.相较于sigmoid和tanh函数，ReLU对于随机梯度下降的收敛有巨大的加速作用,这是由它的线性，非饱和的公式导致的。

2.sigmoid和tanh神经元含有指数运算等耗费计算资源的操作，而ReLU可以简单地通过对一个矩阵进行阈值计算得到。

缺点：

在训练的时候，ReLU单元比较脆弱并且可能“死掉”。举例来说，当一个很大的梯度流过ReLU的神经元的时候，可能会导致梯度更新到一种特别的状态，在这种状态下神经元将无法被其他任何数据点再次激活。如果这种情况发生，那么从此所以流过这个神经元的梯度将都变成0。也就是说，这个ReLU单元在训练中将不可逆转的死亡，因为这导致了数据多样化的丢失。

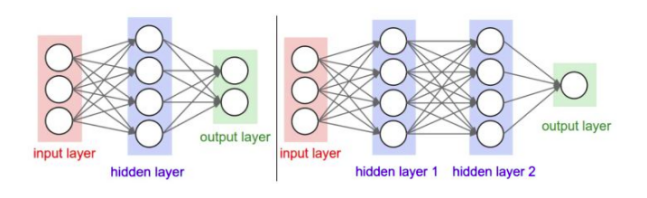
### 1.3.4 其他

Leaky ReLU：

Maxout：

## 1.4 层组织

### 1.4.1 简介



左边是一个2层神经网络，隐层由4个神经元（也可称为单元（unit））组成，输出层由2个神经元组成，输入层是3个神经元。右边是一个3层神经网络，两个含4个神经元的隐层。注意：层与层之间的神经元是全连接的，但是层内的神经元不连接

当我们说N层神经网络的时候，我们没有把输入层算入

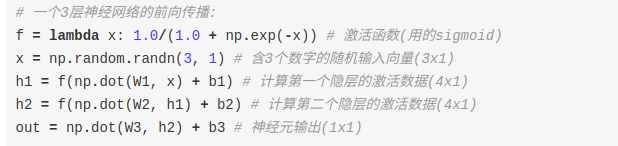
### 1.4.2 确定网络尺寸

用来度量神经网络的尺寸的标准主要有两个：一个是神经元的个数，另一个是参数的个数，用上面图示的两个网络举例：

第一个网络有4+2=6个神经元（输入层不算），[3x4]+[4x2]=20个权重，还有4+2=6个偏置，共26个可学习的参数。

第二个网络有4+4+1=9个神经元，[3x4]+[4x4]+[4x1]=32个权重，4+4+1=9个偏置，共41个可学习的参数。

**不断重复的矩阵乘法与激活函数交织。**将神经网络组织成层状的一个主要原因，就是这个结构让神经网络算法使用矩阵向量操作变得简单和高效



**全连接层的前向传播一般就是先进行一个矩阵乘法，然后加上偏置并运用激活函数。**

### 1.4.3. 设置层的数量和尺寸

有更多神经元的神经网络可以表达更复杂的函数。然而这既是优势也是不足，优势是可以分类更复杂的数据，不足是可能造成对训练数据的过拟合。**过拟合（Overfitting）**是网络对数据中的噪声有很强的拟合能力，而没有重视数据间（假设）的潜在基本关系。

防止神经网络的过拟合有很多方法（L2正则化，dropout和输入噪音等），在实践中，使用这些方法来控制过拟合比减少网络神经元数目要好得多。

# 2. Neural Nets\_2

神经网络就是进行了一系列的线性映射与非线性激活函数交织的运算,神经网络将这些神经元组织成各个层。这些做法共同定义了**评分函数（score function）**的新形式

## 2.1 数据预处理

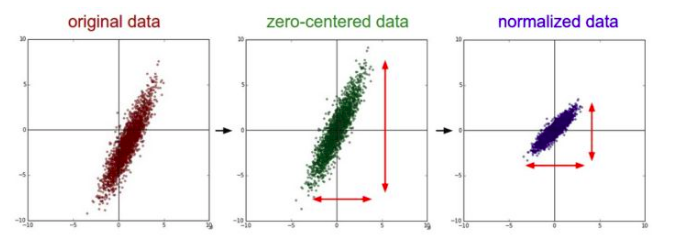
关于数据预处理我们有3个常用的符号，数据矩阵X，假设其尺寸是[N x D]（N是数据样本的数量，D是数据的维度）。

### 2.1.1 均值减法（Mean subtraction）

预处理最常用的形式。它对数据中每个独立特征减去平均值，从几何上可以理解为在每个维度上都将数据云的中心都迁移到原点。

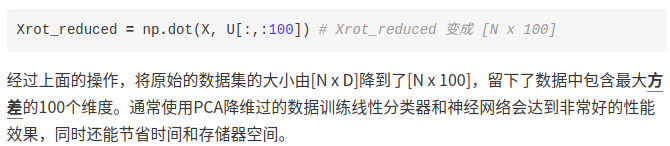
### 2.1.2 归一化（Normalization）

指将数据的所有维度都归一化，使其数值范围都近似相等。有两种常用方法可以实现归一化。第一种是先对数据做零中心化（zero-centered）处理，然后每个维度都除以其标准差，实现代码为X /= np.std(X, axis=0)。第二种方法是对每个维度都做归一化，使得每个维度的最大和最小值是1和-1

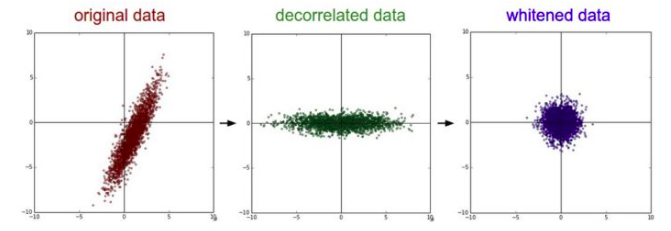


### 2.1.3 PCA和白化

在这种处理中，先对数据进行零中心化处理，然后计算协方差矩阵，它展示了数据中的相关性结构。



白化操作的输入是特征基准上的数据，然后对每个维度除以其特征值来对数值范围进行归一化。该变换的几何解释是：如果数据服从多变量的高斯分布，那么经过白化后，数据的分布将会是一个均值为零，且协方差相等的矩阵



## 2.2 权重初始化

**错误：全零初始化**

如果网络中的每个神经元都计算出同样的输出，然后它们就会在反向传播中计算出同样的梯度，从而进行同样的参数更新。换句话说，如果权重被初始化为同样的值，神经元之间就失去了不对称性的源头

**小随机数初始化。**因此，权重初始值要非常接近0又不能等于0。解决方法就是将权重初始化为很小的数值，以此来打破对称性。其思路是：如果神经元刚开始的时候是随机且不相等的，那么它们将计算出不同的更新，并将自身变成整个网络的不同部分。小随机数权重初始化的实现方法是：W = 0.01 \* np.random.randn(D,H)。

每个神经元的权重向量都被初始化为一个随机向量，而这些随机向量又服从一个多变量高斯分布，这样在输入空间中，所有的神经元的指向是随机的

**使用1/sqrt(n)校准方差**。上面做法存在一个问题，随着输入数据量的增长，随机初始化的神经元的输出数据的分布中的方差也在增大。我们可以除以输入数据量的平方根来调整其数值范围，这样神经元输出的方差就归一化到1了。也就是说，建议将神经元的权重向量初始化为：w = np.random.randn(n) / sqrt(n)。其中n是输入数据的数量。这样就保证了网络中所有神经元起始时有近似同样的输出分布。

## 2.3 偏置（biases）的初始化

通常将偏置初始化为0，这是因为随机小数值权重矩阵已经打破了对称性。

## 2.4 批量归一化

做法是让激活数据在训练开始前通过一个网络，网络处理数据使其服从标准高斯分布。

**批量归一化可以理解为在网络的每一层之前都做预处理，只是这种操作以另一种方式与网络集成在了一起**

## 2.5 正则化 Regularization

是通过控制神经网络的容量来防止其过拟合

### 2.5.1 L2正则化

是最常用的正则化方法了。可以通过惩罚目标函数中所有参数的平方将其实现

即对于网络中的每个权重w，向目标函数中增加一个，其中λ是正则化强度，它对于大数值的权重向量进行严厉惩罚，倾向于更加分散的权重向量，使网络更倾向于使用所有输入特征，而不是严重依赖输入特征中某些小部分特征。

### 2.5.2 L1正则化

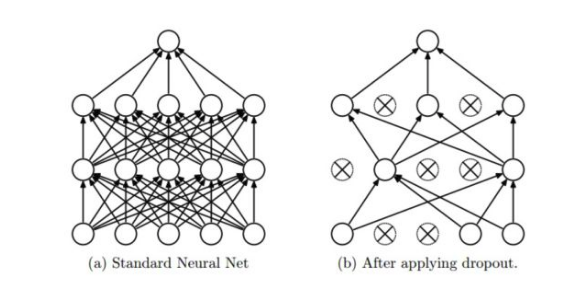
相对常用的正则化方法。对于每个w我们都向目标函数增加一个λ|w|。会让权重向量在最优化的过程中变得稀疏（即非常接近0）。

### 2.5.3 最大范式约束（Max norm constraints）

另一种形式的正则化是给每个神经元中权重向量的量级设定上限，并使用投影梯度下降来确保这一约束

### 2.5.4 随机失活（Dropout）

一个简单又极其有效的正则化方法。与L1正则化，L2正则化和最大范式约束等方法互为补充。在训练的时候，随机失活的实现方法是让神经元以超参数p的概率被激活或者被设置为0。



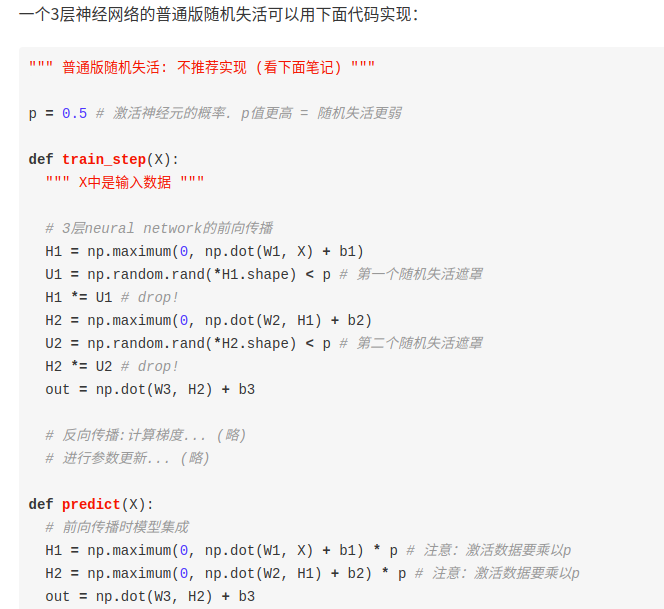
在训练过程中，随机失活可以被认为是对完整的神经网络抽样出一些子集，每次基于输入数据只更新子网络的参数（然而，数量巨大的子网络们并不是相互独立的，因为它们都共享参数）。在测试过程中不使用随机失活，可以理解为是对数量巨大的子网络们做了模型集成（model ensemble），以此来计算出一个平均的预测

### Tips：随机失活dropout的意义

Dropout的意义在于，减小了不同神经元的依赖度。有些中间输出，在给定的训练集上，可能发生只依赖某些神经元的情况，这就会造成对训练集的过拟合。而随机关掉一些神经元，可以让更多神经元参与到最终的输出当中。我觉得dropout方法也可以看成，联合很多规模比较小的网络的预测结果，去获取最终的预测，总结的更好一点：

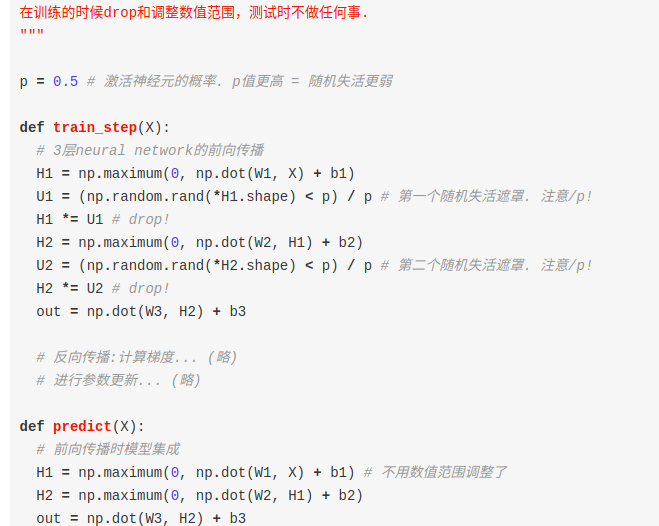
1.取平均的作用： 先回到正常的模型（没有dropout），我们用相同的训练数据去训练5个不同的神经网络，一般会得到5个不同的结果，此时我们可以采用 “5个结果取均值”或者“多数取胜的投票策略”去决定最终结果。（例如 3个网络判断结果为数字9,那么很有可能真正的结果就是数字9，其它两个网络给出了错误结果）。这种“综合起来取平均”的策略通常可以有效防止过拟合问题。因为不同的网络可能产生不同的过拟合，取平均则有可能让一些“相反的”拟合互相抵消。dropout掉不同的隐藏神经元就类似在训练不同的网络（随机删掉一半隐藏神经元导致网络结构已经不同)，整个dropout过程就相当于 对很多个不同的神经网络取平均。而不同的网络产生不同的过拟合，一些互为“反向”的拟合相互抵消就可以达到整体上减少过拟合。

2.减少神经元之间复杂的共适应关系： 因为dropout程序导致两个神经元不一定每次都在一个dropout网络中出现。（这样权值的更新不再依赖于有固定关系的隐含节点的共同作用，阻止了某些特征仅仅在其它特定特征下才有效果的情况）。 迫使网络去学习更加鲁棒的特征 （这些特征在其它的神经元的随机子集中也存在）。换句话说假如我们的神经网络是在做出某种预测，它不应该对一些特定的线索片段太过敏感，即使丢失特定的线索，它也应该可以从众多其它线索中学习一些共同的模式（鲁棒性）。（这个角度看 dropout就有点像L1，L2正则，减少权重使得网络对丢失特定神经元连接的鲁棒性提高）



**在predict函数中不进行随机失活，但是对于两个隐层的输出都要乘以p，调整其数值范围。**

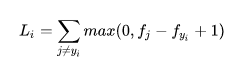
反向随机失活（inverted dropout）：



## 2.6 损失函数

我们已经讨论过损失函数的正则化损失部分，它可以看做是对模型复杂程度的某种惩罚。损失函数的第二个部分是数据损失，它是一个有监督学习问题，用于衡量分类算法的预测结果（即分类评分）和真实标签结果之间的一致性。数据损失是对所有样本的数据损失求平均

### 2.6.1 SVM



### 2.6.2 Softmax

使用交叉熵损失：



### 2.6.3 回归问题

预测实数的值的问题，比如预测房价，预测图片中某个东西的长度等。对于这种问题，通常是计算预测值和真实值之间的损失。然后用L2平方范式或L1范式度量差异。

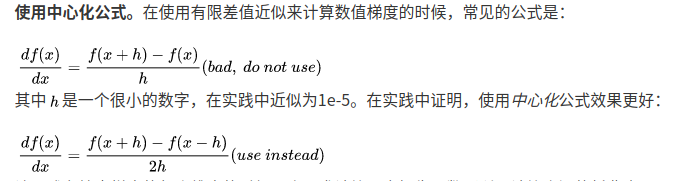
当面对一个回归任务，首先考虑是不是必须这样。一般而言，尽量把你的输出变成二分类，然后对它们进行分类，从而变成一个分类问题。

# 3. Neural Nets\_3

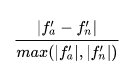
## 3.1 梯度检查

**把解析梯度和数值计算梯度进行比较**

### 3.1.1 使用中心化公式



### 3.1.2 使用相对误差来比较



考虑了**差值**占两个梯度绝对值的比例

网络的深度越深，相对误差就越高

### 3.1.3 目标函数的不可导点（kinks）

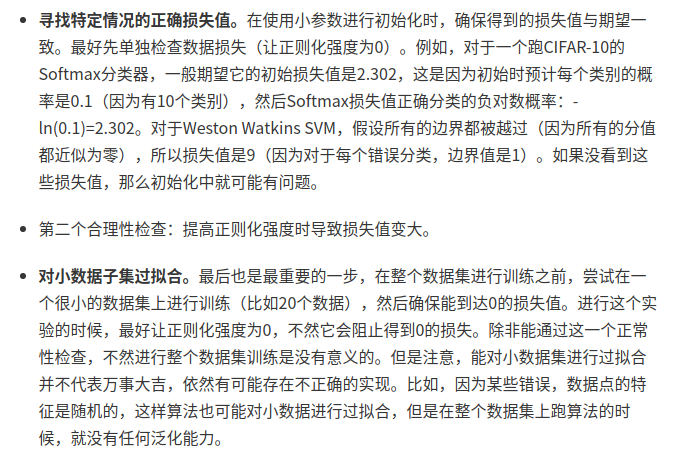
在进行梯度检查时，一个导致不准确的原因是不可导点问题。不可导点是指目标函数不可导的部分，由ReLU（max(0,x)）等函数，或SVM损失，Maxout神经元等引入。在计算损失的过程中是可以知道不可导点有没有被越过的

### 3.1.4 不要让正则化吞没数据

通常损失函数是数据损失和正则化损失的和（例如L2对权重的惩罚）。需要注意的危险是正则化损失可能吞没掉数据损失。

因此，推荐先关掉正则化对数据损失做单独检查，然后对正则化做单独检查。

## 3.2 合理性检查

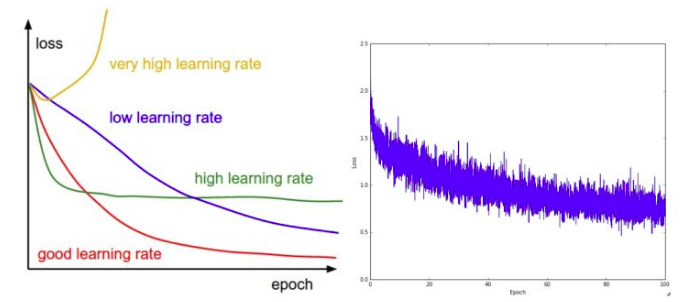


## 3.3 检查整个学习过程

在训练神经网络的时候，应该跟踪多个重要数值。这些数值输出的图表是观察训练进程的一扇窗口，是直观理解不同的超参数设置效果的工具，从而知道如何修改超参数以获得更高效的学习过程。

相较于迭代次数（iterations），一般更倾向跟踪周期，这是因为迭代次数与数据的批尺寸（batchsize）有关，而批尺寸的设置又可以是任意的。

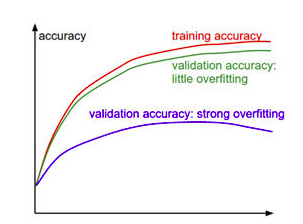
训练期间第一个要跟踪的数值就是损失值，它在前向传播时对每个独立的批数据进行计算。下图展示的是随着损失值随时间的变化，尤其是曲线形状会给出关于学习率设置的情况：



损失值的震荡程度和**批尺寸（batch size）**有关，当批尺寸为1，震荡会相对较大。当批尺寸就是整个数据集时震荡就会最小，因为每个梯度更新都是单调地优化损失函数（除非学习率设置得过高）。

## 3.4 训练集和验证集准确率

在训练分类器的时候，需要跟踪的第二重要的数值是验证集和训练集的准确率。这个图表能够展现知道模型过拟合的程度：



在训练集准确率和验证集准确率中间的空隙指明了模型过拟合的程度。在图中，蓝色的验证集曲线显示相较于训练集，验证集的准确率低了很多，这就说明模型有很强的过拟合。遇到这种情况，就应该增大正则化强度（更强的L2权重惩罚，更多的随机失活等）或收集更多的数据。另一种可能就是验证集曲线和训练集曲线如影随形，这种情况说明你的模型容量还不够大：应该通过增加参数数量让模型容量更大些。

## 3.5 权重更新比例

最后一个应该跟踪的量是权重中更新值的数量和全部值的数量之间的比例。注意：是更新的，而不是原始梯度（比如，在普通sgd中就是梯度乘以学习率）。需要对每个参数集的更新比例进行单独的计算和跟踪。一个经验性的结论是这个比例应该在1e-3左右。如果更低，说明学习率可能太小，如果更高，说明学习率可能太高。



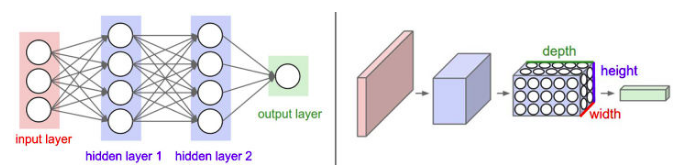
# 4. ConvNet notes（卷积神经网络）

卷积神经网络的结构基于一个假设，即输入数据是图像，基于该假设，我们就向结构中添加了一些特有的性质。这些特有属性使得前向传播函数实现起来更高效，并且大幅度降低了网络中参数的数量。

## 4.1 结构概述

常规神经网络对于大尺寸图像效果不尽人意，参数多，效率低，而且容易过拟合。

神经元的三维排列。卷积神经网络针对输入全部是图像的情况，将结构调整得更加合理，获得了不小的优势。与常规神经网络不同，卷积神经网络的各层中的神经元是3维排列的：宽度、高度和深度（这里的深度指的是激活数据体的第三个维度，而不是整个网络的深度，整个网络的深度指的是网络的层数）



左边是一个3层的神经网络。右边是一个卷积神经网络，图例中网络将它的神经元都排列成3个维度（宽、高和深度）。卷积神经网络的每一层都将3D的输入数据变化为神经元3D的激活数据并输出

**卷积神经网络是由层组成的。每一层都有一个简单的API：用一些含或者不含参数的可导的函数，将输入的3D数据变换为3D的输出数据**

## 4.2 各种层

卷积神经网络主要由三种类型的层构成：卷积层，汇聚（Pooling）层和全连接层（全连接层和常规神经网络中的一样）。

### 4.2.1 卷积层

卷积层是构建卷积神经网络的核心层，它产生了网络中大部分的计算量

卷积层的参数是有一些可学习的滤波器集合构成的。每个滤波器在空间上（宽度和高度）都比较小，但是深度和输入数据一致。

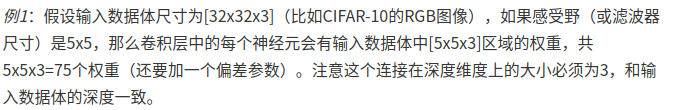
当滤波器沿着输入数据的宽度和高度滑过后，会生成一个2维的激活图（activation map），激活图给出了在每个空间位置处滤波器的反应。

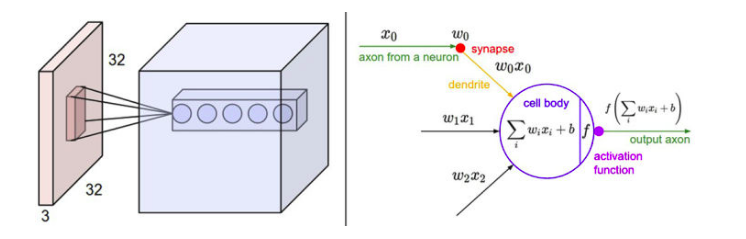
直观地来说，网络会让滤波器学习到当它看到某些类型的视觉特征时就激活。

在每个卷积层上，我们会有一整个集合的滤波器（比如12个），每个都会生成一个不同的二维激活图。将这些激活映射在深度方向上层叠起来就生成了输出数据。

用**大脑和生物神经元**来做比喻，那么输出的3D数据中的每个数据项可以被看做是神经元的一个输出，而该神经元只观察输入数据中的一小部分，并且和空间上左右两边的所有神经元共享参数

让每个神经元只与输入数据的一个局部区域连接。该连接的空间大小叫做神经元的**感受野（receptive field）**，它的尺寸是一个超参数（其实就是滤波器的空间尺寸。





3个超参数控制着输出数据体的尺寸：**深度（depth）**，**步长（stride）**和**零填充（zero-padding）**。下面是对它们的讨论：

1.首先，输出数据体的深度是一个超参数：它和使用的滤波器的数量一致，而每个滤波器在输入数据中寻找一些不同的东西。举例来说，如果第一个卷积层的输入是原始图像，那么在深度维度上的不同神经元将可能被不同方向的边界，或者是颜色斑点激活。我们将这些沿着深度方向排列、感受野相同的神经元集合称为**深度列（depth column）**，也有人使用纤维（fibre）来称呼它们。

2.其次，在滑动滤波器的时候，必须指定步长。当步长为1，滤波器每次移动1个像素。当步长为2，滤波器滑动时每次移动2个像素。这个操作会让输出数据体在空间上变小。

3.有时候将输入数据体用0在边缘处进行填充是很方便的。这个零填充（zero-padding）的尺寸是一个超参数。零填充有一个良好性质，即可以控制输出数据体的空间尺寸

输出数据体在空间上的尺寸可以通过输入数据体尺寸（W），卷积层中神经元的感受野尺寸（F），步长（S）和零填充的数量（P）的函数来计算。（译者注：这里假设输入数组的空间形状是正方形，即高度和宽度相等）输出数据体的空间尺寸为**(W-F +2P)/S+1**。

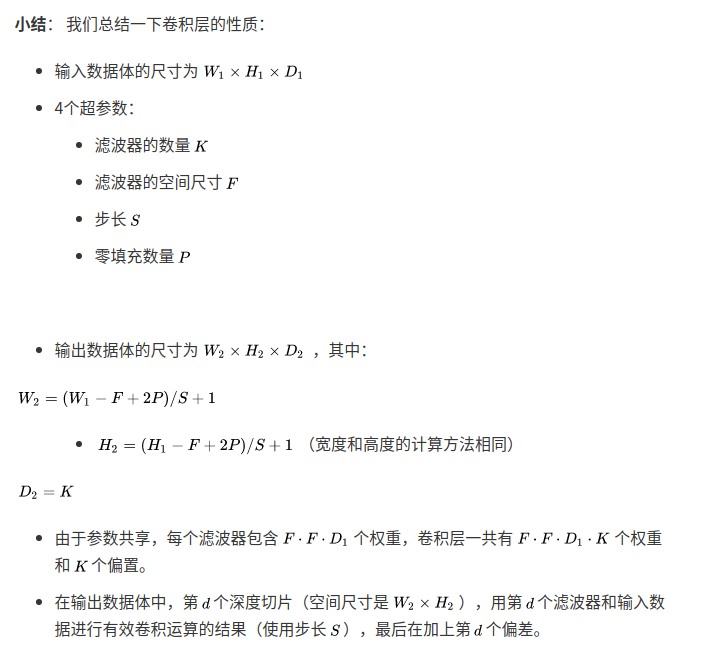
**这些空间排列的超参数之间是相互限制的**

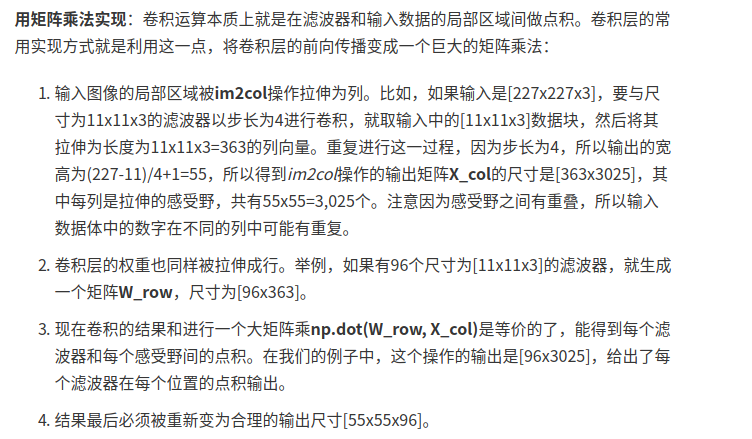
### Tips： 参数共享

在卷积层中使用参数共享是用来控制参数的数量。

在每个深度切片中的55x55个权重使用的都是同样的参数。在反向传播的时候，都要计算每个神经元对它的权重的梯度，但是需要把同一个深度切片上的所有神经元对权重的梯度累加，这样就得到了对共享权重的梯度。这样，每个切片只更新一个权重集

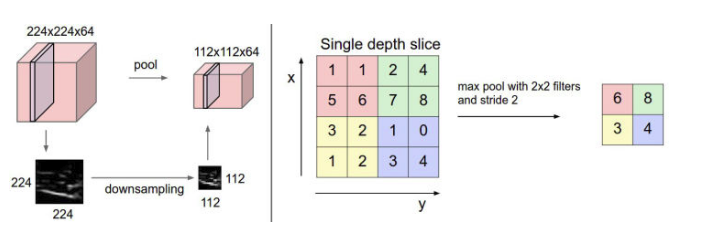
有时候参数共享假设可能没有意义，特别是当卷积神经网络的输入图像是一些明确的中心结构时候。这时候我们就应该期望在图片的不同位置学习到完全不同的特征。



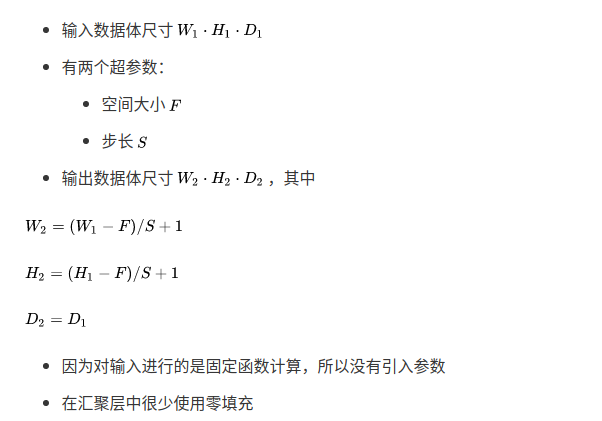


### 4.2.2 汇聚层（Pooling）

在连续的卷积层之间会周期性地插入一个汇聚层。它的作用是逐渐降低数据体的空间尺寸，这样的话就能减少网络中参数的数量，使得计算资源耗费变少，也能有效控制过拟合。汇聚层使用MAX操作，对输入数据体的每一个深度切片独立进行操作，改变它的空间尺寸。最常见的形式是汇聚层使用尺寸2x2的滤波器，以步长为2来对每个深度切片进行降采样，将其中75%的激活信息都丢掉。每个MAX操作是从4个数字中取最大值（也就是在深度切片中某个2x2的区域）。深度保持不变。汇聚层的一些公式：



**无超参数！**



普通汇聚（General Pooling）：除了最大汇聚，汇聚单元还可以使用其他的函数，比如平均汇聚（average pooling）或L-2范式汇聚（L2-norm pooling）。

**反向传播：**回顾一下反向传播的内容，其中max(x,y)函数的反向传播可以简单理解为将梯度只沿最大的数回传。因此，在向前传播经过汇聚层的时候，通常会把池中最大元素的索引记录下来（有时这个也叫作**道岔（switches）**），这样在反向传播的时候梯度的路由就很高效。

### 4.2.3 归一化层

### 4.2.4 全连接层

在全连接层中，神经元对于前一层中的所有激活数据是全部连接的，这个常规神经网络中一样。它们的激活可以先用矩阵乘法，再加上偏差。

把全连接层转化成卷积层：

全连接层和卷积层之间唯一的不同就是卷积层中的神经元只与输入数据中的一个局部区域连接，并且在卷积列中的神经元共享参数。然而在两类层中，神经元都是计算点积，所以它们的函数形式是一样的。

面对384x384的图像，让（含全连接层）的初始卷积神经网络以32像素的步长独立对图像中的224x224块进行多次评价，其效果和使用把全连接层变换为卷积层后的卷积神经网络进行一次前向传播是一样的。

## 4.3 卷积神经网络的结构

卷积神经网络通常是由三种层构成：卷积层，汇聚层（除非特别说明，一般就是最大值汇聚）和全连接层（简称FC）。ReLU激活函数也应该算是是一层，它逐元素地进行激活函数

操作。

### 4.3.1 层的排列规律

卷积神经网络最常见的形式就是将一些卷积层和ReLU层放在一起，其后紧跟汇聚层，然后重复如此直到图像在空间上被缩小到一个足够小的尺寸，在某个地方过渡成成全连接层也较为常见。最后的全连接层得到输出，比如分类评分等。

### 4.3.2 层的尺寸设置规律

到现在为止，我们都没有提及卷积神经网络中每层的超参数的使用。

输入层（包含图像的）应该能被2整除很多次。常用数字包括32（比如CIFAR-10），64，96（比如STL-10）或224（比如ImageNet卷积神经网络），384和512。

卷积层应该使用小尺寸滤波器（比如3x3或最多5x5），使用步长S=1。还有一点非常重要，就是对输入数据进行零填充，这样卷积层就不会改变输入数据在空间维度上的尺寸。

汇聚层负责对输入数据的空间维度进行降采样。最常用的设置是用用2x2感受野（即F=2）的最大值汇聚，步长为2（S=2）。

# 5. TensorFlow中文社区入门学习

## 5.1 基本使用

使用 TensorFlow, 你必须明白 TensorFlow:

1. 使用图 (graph) 来表示计算任务.

2. 在被称之为 会话 (Session) 的上下文 (context) 中执行图.

3. 使用 tensor 表示数据.

4. 通过 变量 (Variable) 维护状态.

5. 使用 feed 和 fetch 可以为任意的操作(arbitrary operation) 赋值或者从其中获取数据.

## 5.2 综述

TensorFlow 是一个编程系统, 使用图来表示计算任务. 图中的节点被称之为 op (operation 的缩写). 一个 op 获得 0 个或多个 Tensor, 执行计算, 产生 0 个或多个 Tensor. 每个 Tensor 是一个类型化的多维数组. 例如, 你可以将一小组图像集表示为一个四维浮点数数组, 这四个维度分别是 [batch, height, width, channels].

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 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 和 C++ 语言中, 返回的 tensor 是 tensorflow::Tensor 实例.

## 5.3 计算图

TensorFlow 程序通常被组织成一个构建阶段和一个执行阶段. 在构建阶段, op 的执行步骤 被描述成一个图. 在执行阶段, 使用会话执行执行图中的 op.

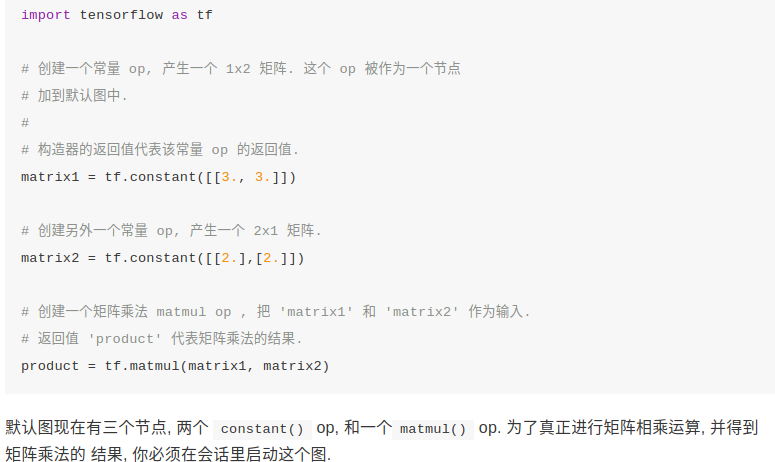
例如, 通常在构建阶段创建一个图来表示和训练神经网络, 然后在执行阶段反复执行图中的训练 op.

## 5.4 构建图

构建图的第一步, 是创建源 op (source op). 源 op 不需要任何输入, 例如 常量 (Constant). 源 op 的输出被传递给其它 op 做运算.

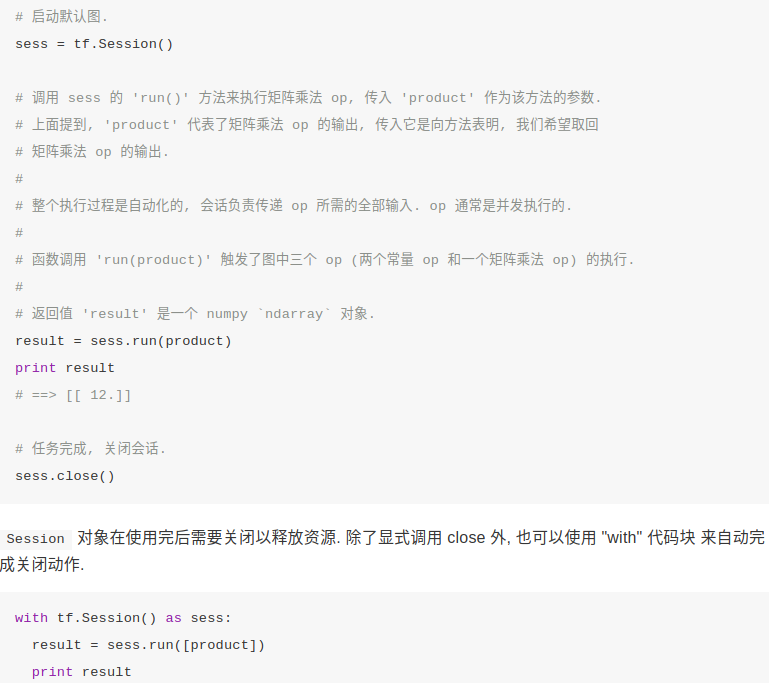
Python 库中, op 构造器的返回值代表被构造出的 op 的输出, 这些返回值可以传递给其它 op 构造器作为输入.

TensorFlow Python 库有一个默认图 (default graph), op 构造器可以为其增加节点. 这个默认图对 许多程序来说已经足够用了



## 5.5 在一个会话中启动图

构造阶段完成后, 才能启动图. 启动图的第一步是创建一个 Session 对象, 如果无任何创建参数, 会话构造器将启动默认图.



在实现上, TensorFlow 将图形定义转换成分布式执行的操作, 以充分利用可用的计算资源(如 CPU 或 GPU). 一般你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

## 5.6 交互式使用

文档中的 Python 示例使用一个会话 Session 来 启动图, 并调用 Session.run() 方法执行操作.为了便于使用诸如 IPython 之类的 Python 交互环境, 可以使用 InteractiveSession 代替 Session 类, 使用 Tensor.eval() 和 Operation.run() 方法代替 Session.run(). 这样可以避免使用一个变量来持有会话.

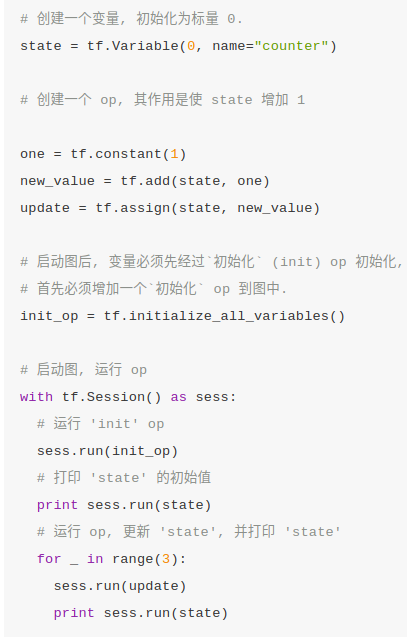


## 5.7 Tensor

TensorFlow 程序使用 tensor 数据结构来代表所有的数据, 计算图中, 操作间传递的数据都是 tensor. 你可以把 TensorFlow tensor 看作是一个 n 维的数组或列表. 一个 tensor 包含一个静态类型 rank, 和 一个 shape.

### 5.7.1 变量

Variables for more details. 变量维护图执行过程中的状态信息. 下面的例子演示了如何使用变量实现一个简单的计数器.



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.

### 5.7.2 Fetch

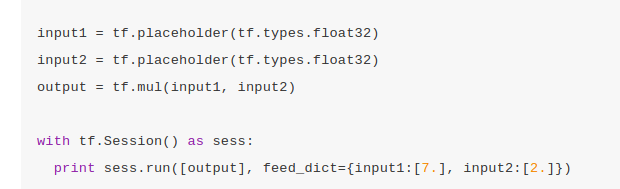
为了取回操作的输出内容, 可以在使用 Session 对象的 run() 调用 执行图时, 传入一些 tensor, 这些 tensor 会帮助你取回结果. 在之前的例子里, 我们只取回了单个节点 state, 但是你也可以取回多个 tensor:



### 5.7.3 Feed

上述示例在计算图中引入了 tensor, 以常量或变量的形式存储. TensorFlow 还提供了 feed 机制, 该机制 可以临时替代图中的任意操作中的 tensor 可以对图中任何操作提交补丁, 直接插入一个 tensor.

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.



## 5.8 训练模型



## 5.9 评估模型



# Tensorflow构建一个多层卷积网络

## 6.1 权重初始化&&卷积和池化



## 6.2 第一层卷积



## 6.3 第二层卷积&&密集连接层



## 6.4 Dropout&&输出



## 6.5 训练和评估模型

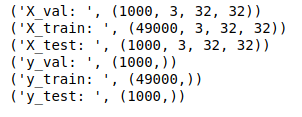
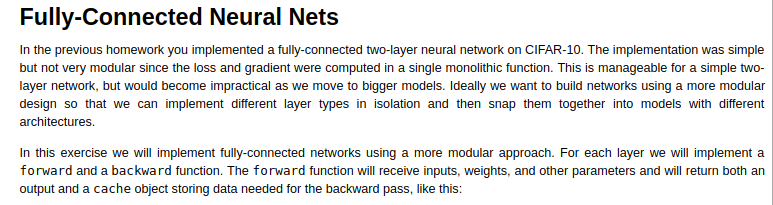


# 7. Tensorflow构建一个卷积神经网络

参考网址：<http://www.tensorfly.cn/tfdoc/tutorials/deep_cnn.html>

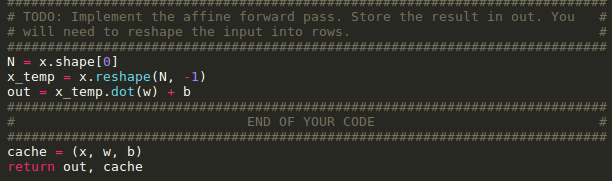
# Assignment #2

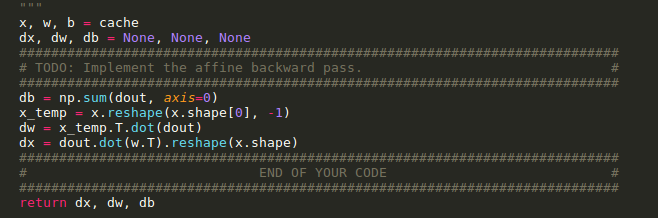
## Q1: Fully-connected Neural Network



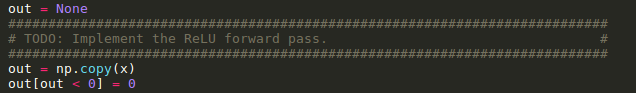
前向和后向传播算法实现;

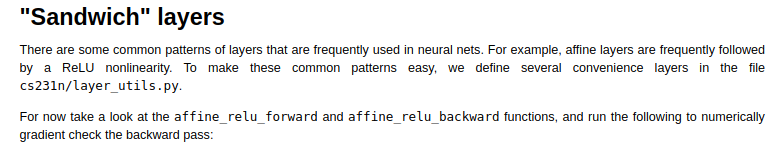
F(x) = wx + b

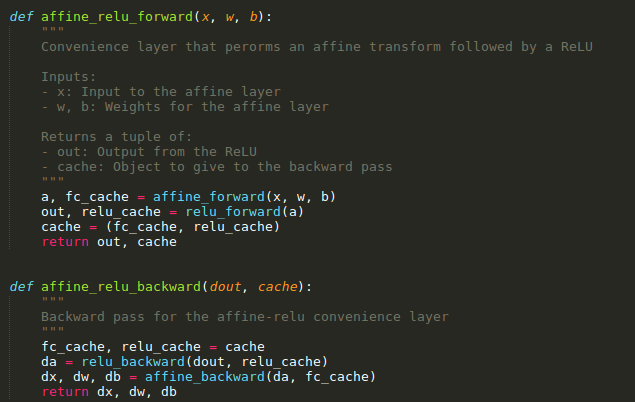


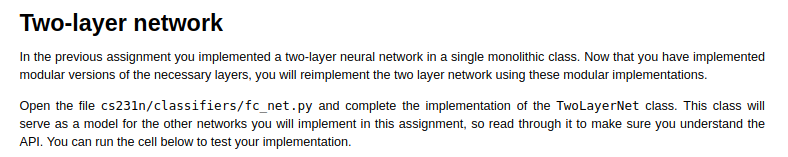


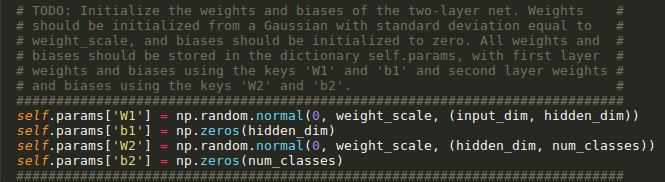
F(x) = Relu＊ｆ(x)

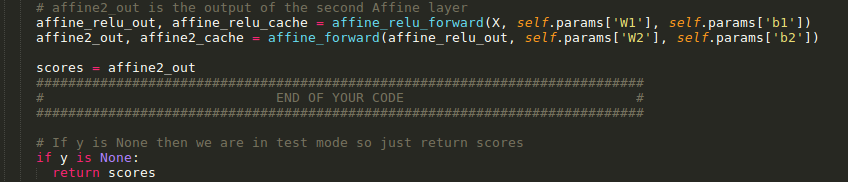


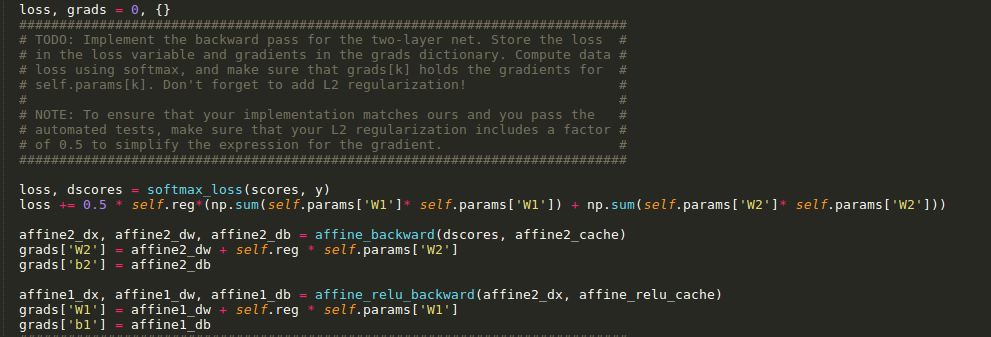


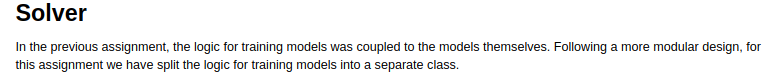






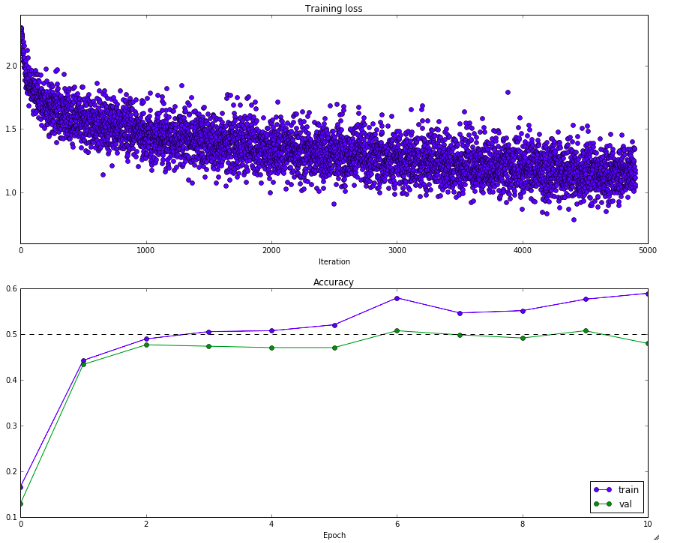






Slover实例：

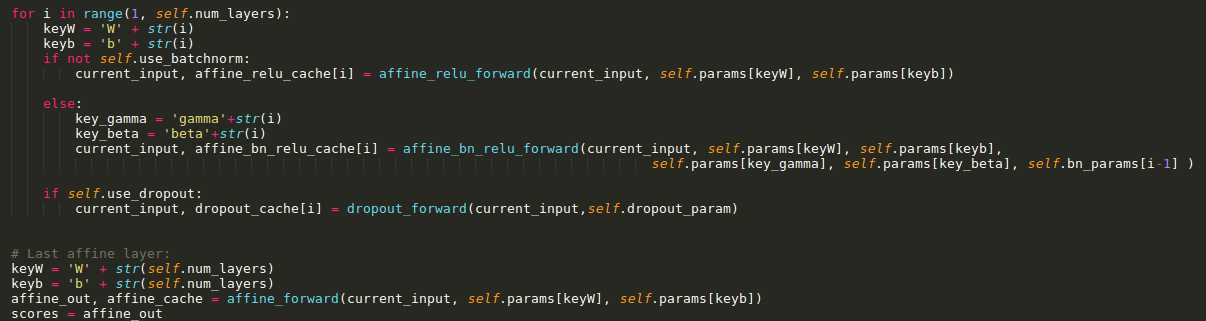




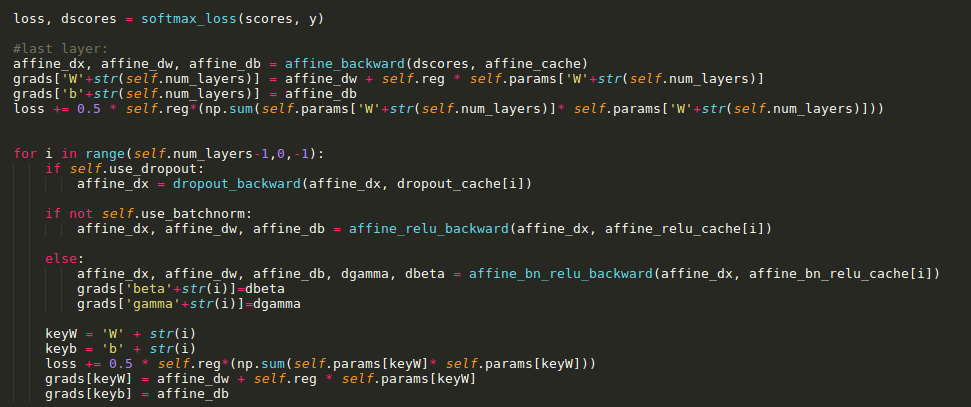


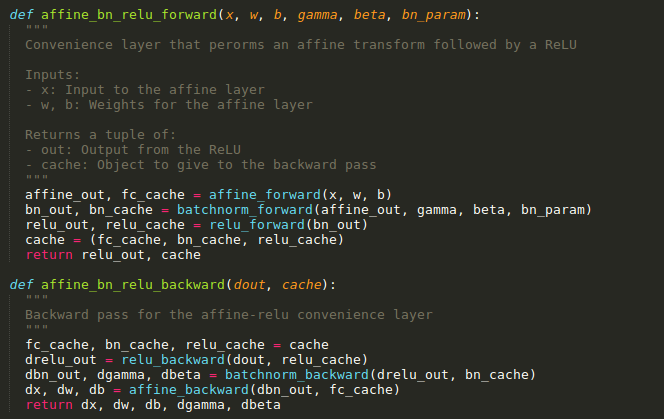


**Train:**

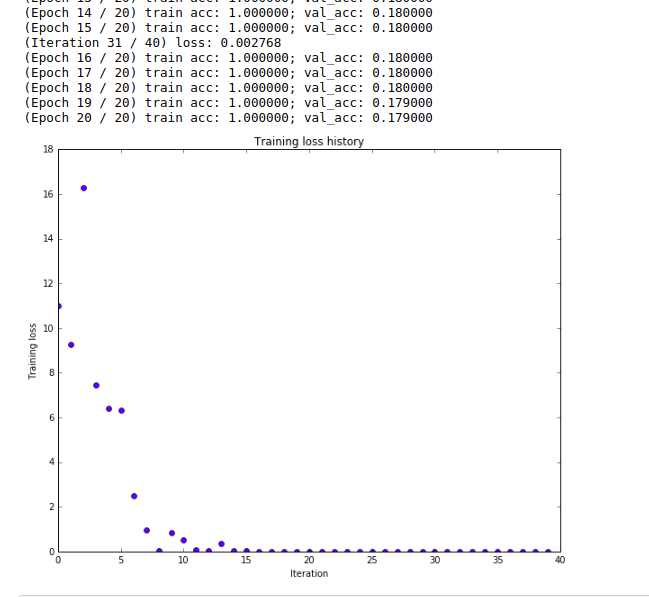
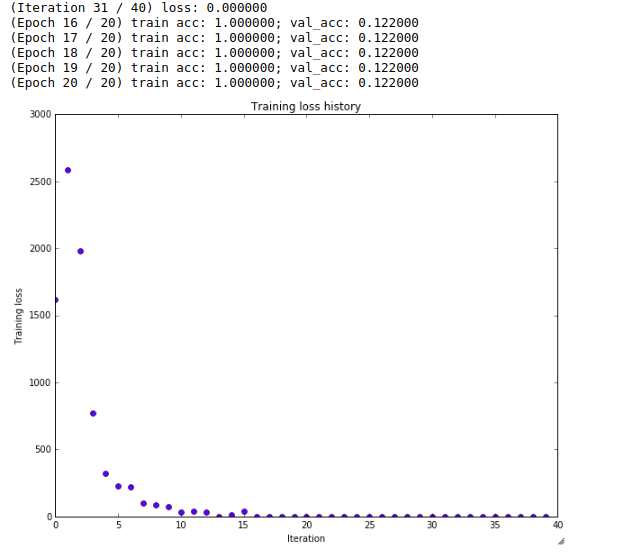


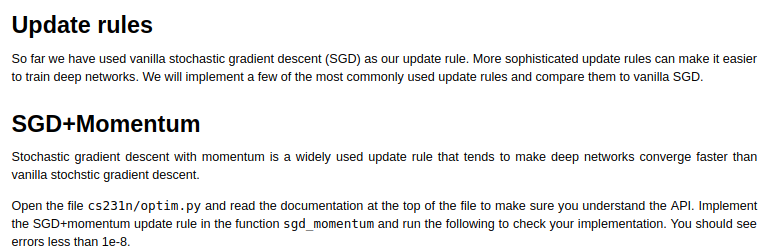
**Test:**

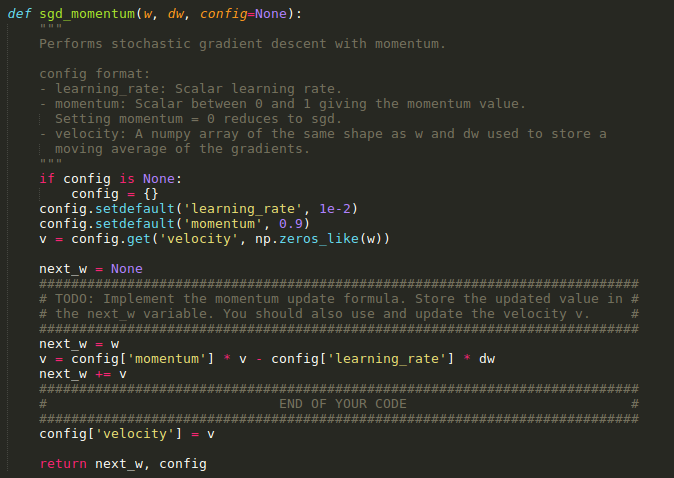




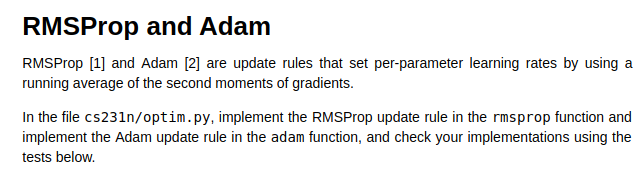
小批量过拟合（三层＆＆五层）

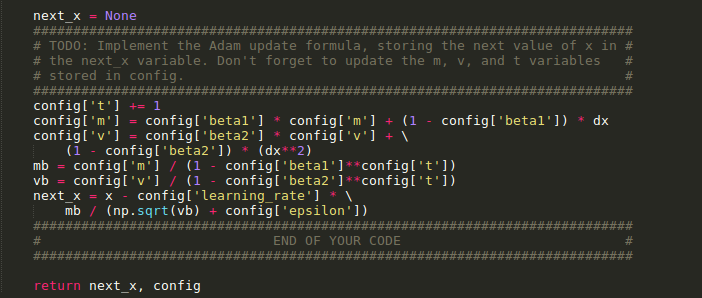




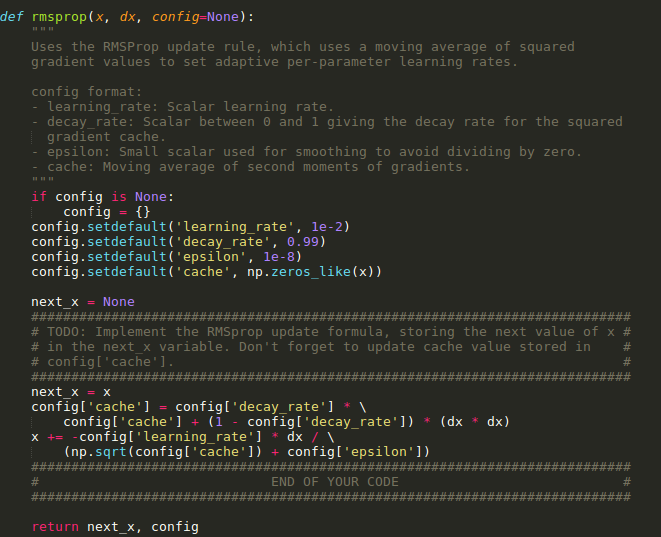


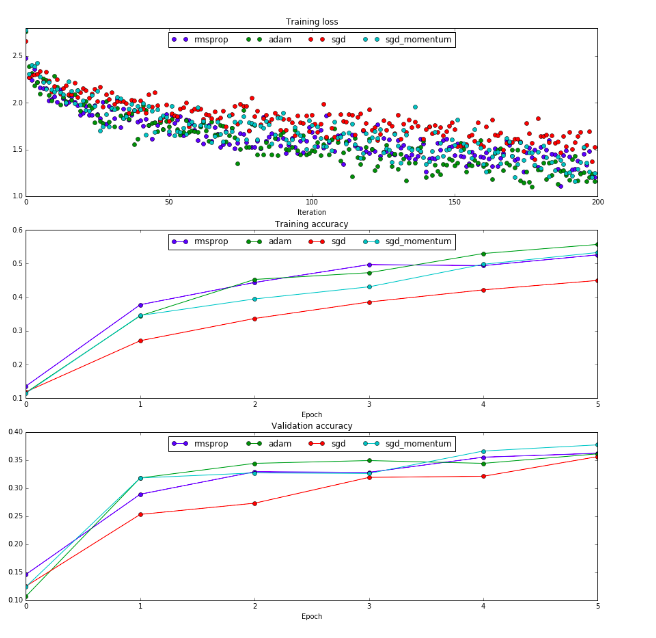


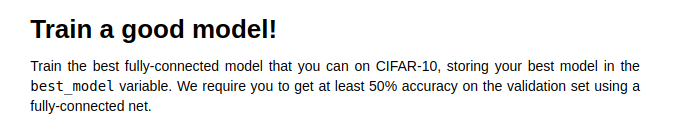
Adam

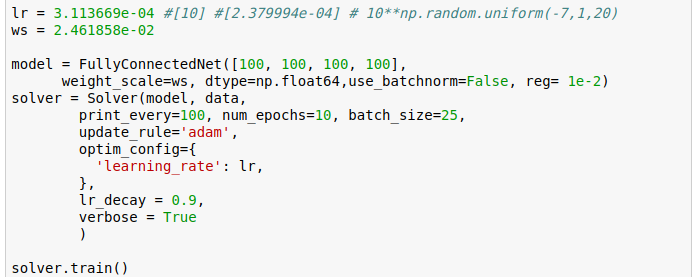


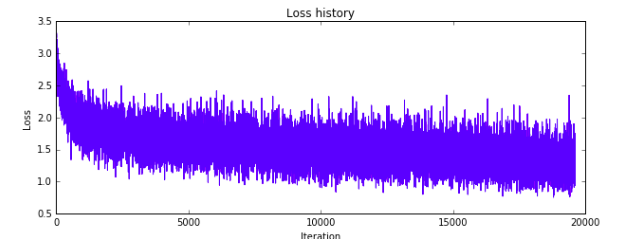
RMSProp

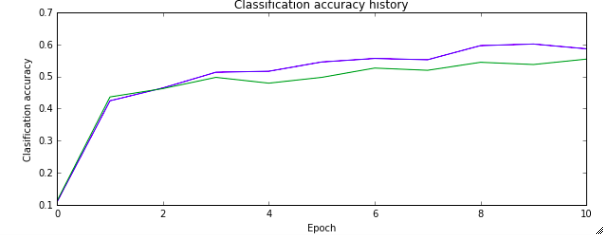


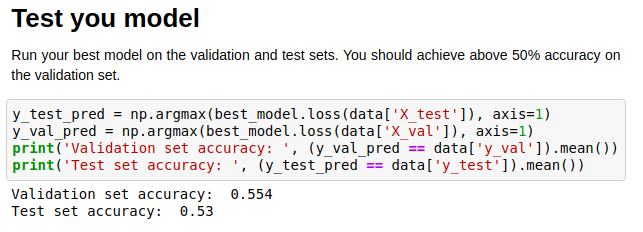




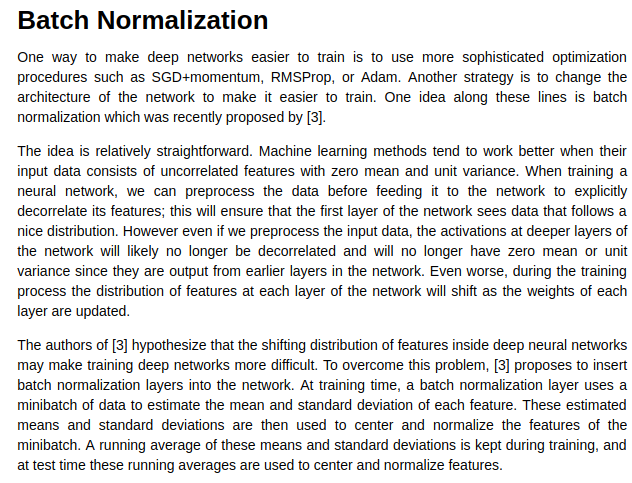


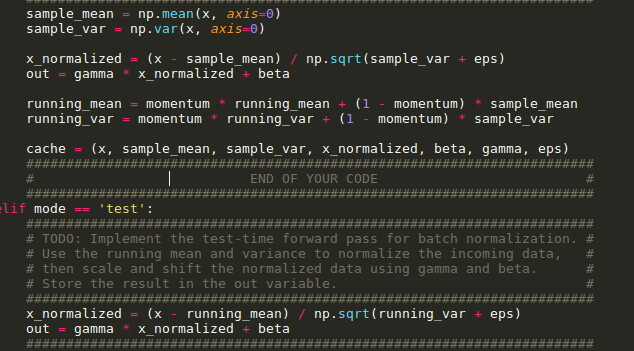


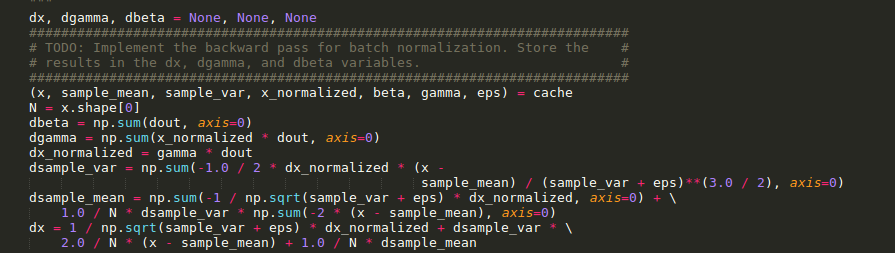


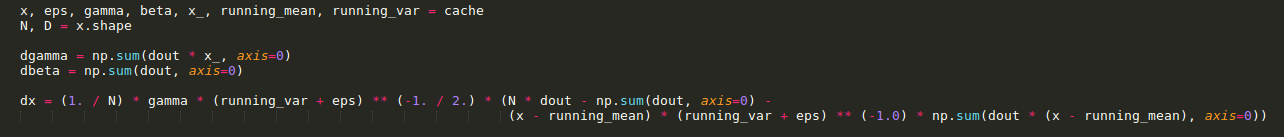


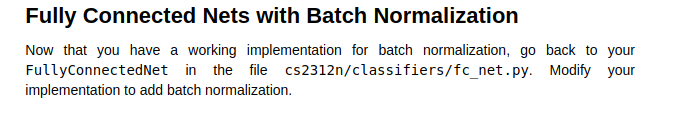
## Q2: Batch Normalization

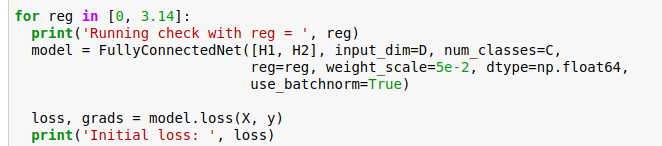






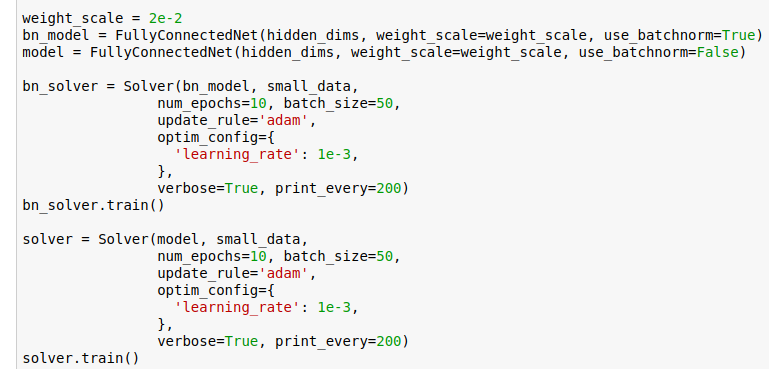




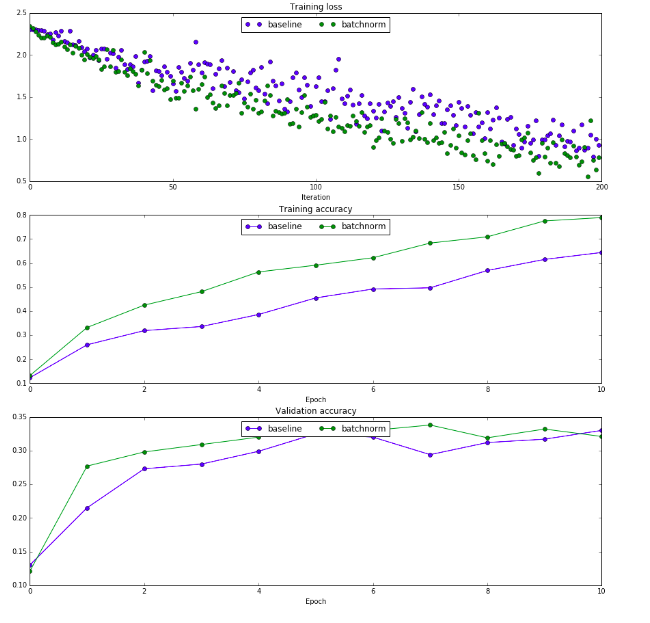


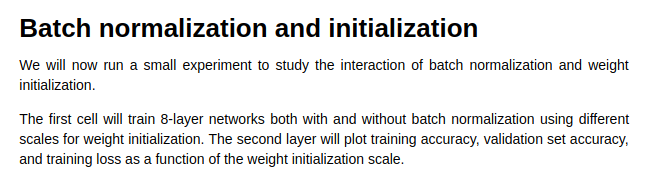
**Batchnorm for deep networks**

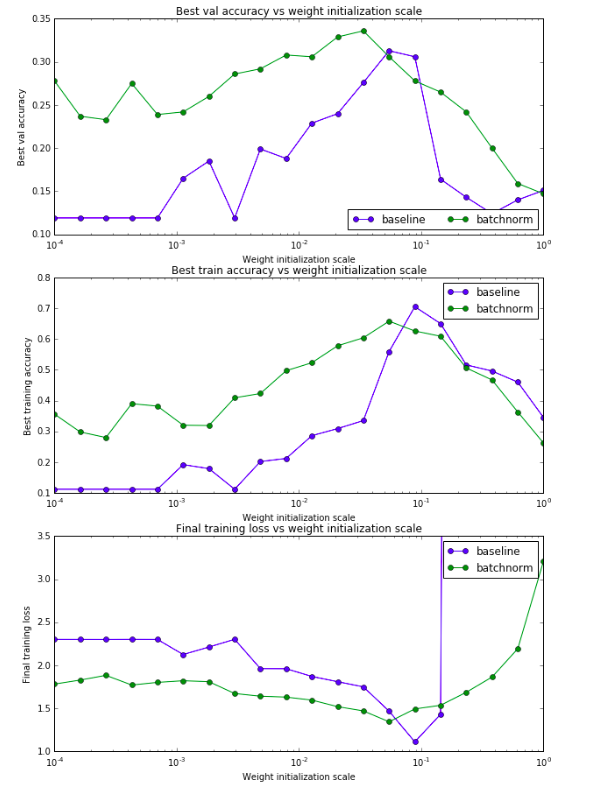
Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.



结果对比：







**Question:**

**Describe the results of this experiment, and try to give a reason why the experiment gave the results that it did.**

Answer:

In the last plot, when the weight initialization scale increases, the loss

for the training without batch normalization blows up, while the loss with

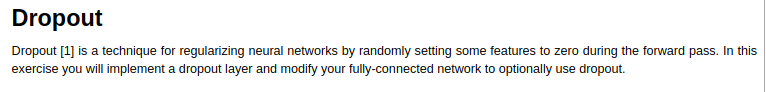
batch normalization stays finite. It seems that batch normalization makes

the training more robust with respect the change of weight initizaliation

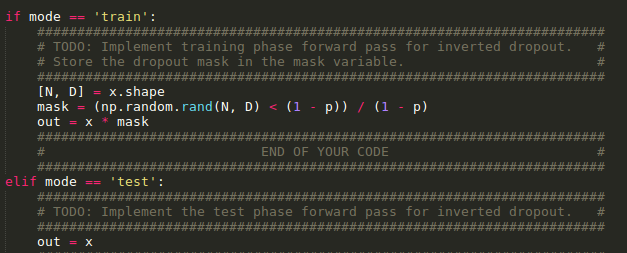
scale. The reason is that, batch normalization normalizes the maginitude

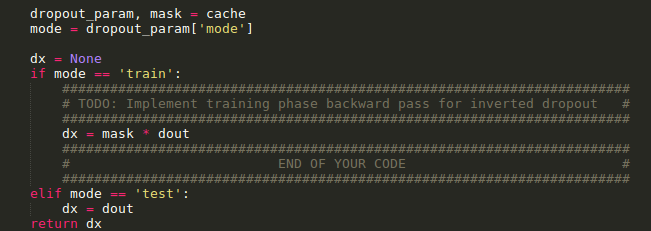
of input values, preventing them from aggregating to too large or too small.

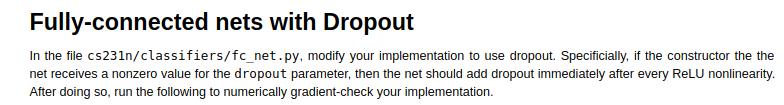
## Q3: Dropout

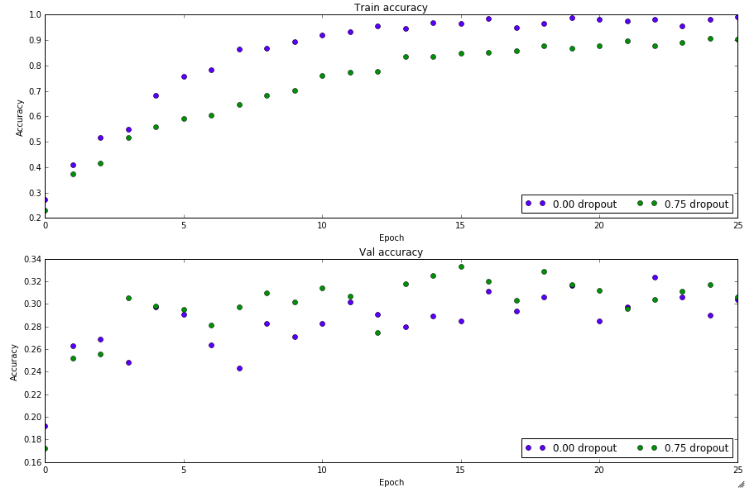
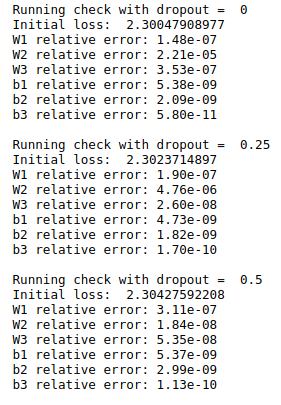


前向和后向传播：









**Question**

**Explain what you see in this experiment. What does it suggest about dropout?**

Answer:

Without using dropout, the final difference between training accuracy (99%)

and validation accuracy (29%) is around 70%. After using dropout with

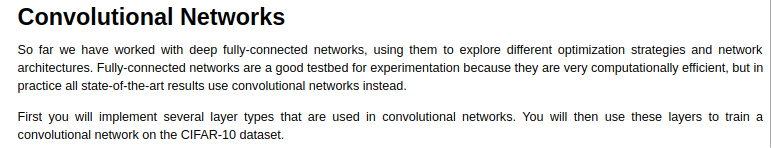
p=0.75, the final difference between training accuracy (92%) and validation

accuracy (31%) is around 61%. We see that the difference between traning

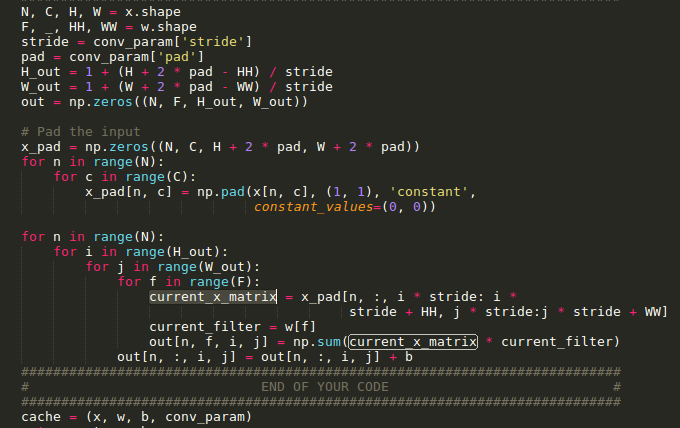
and validation accruacy has been reduced. So it suggests that dropout can

relieve the problem of overfitting.

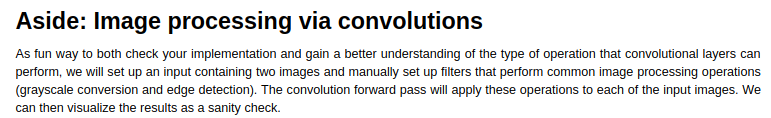
## Q4: Convolutional Networks

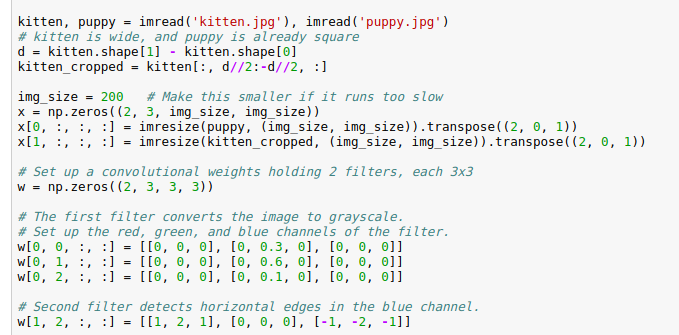


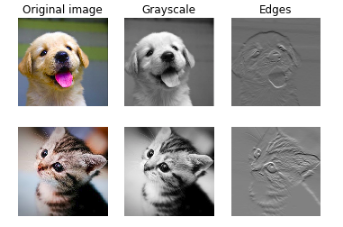
前向和后向传播算法

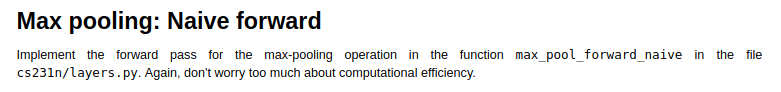




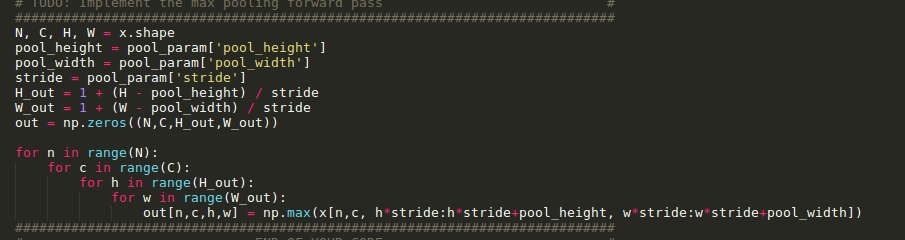


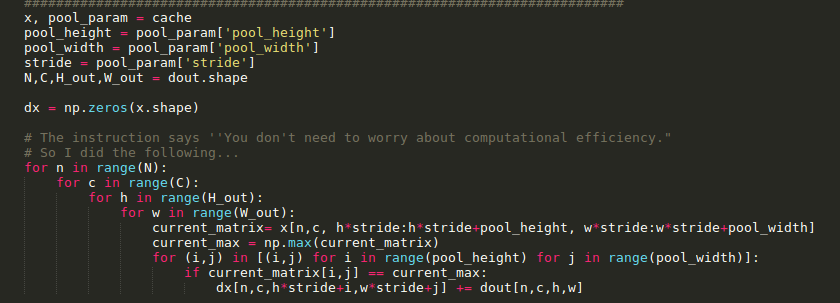


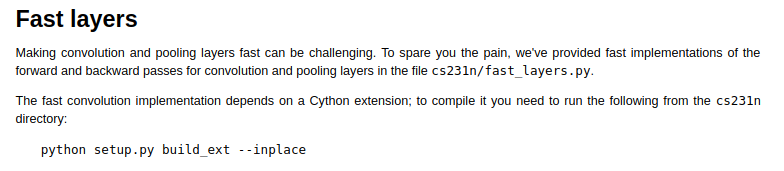


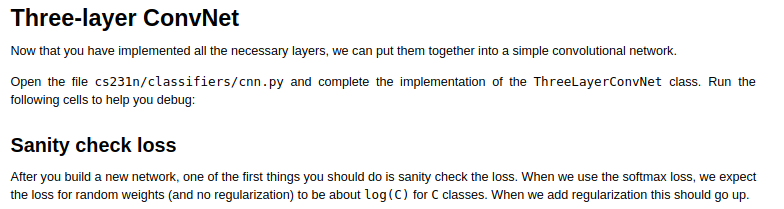
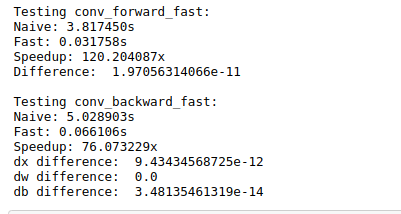


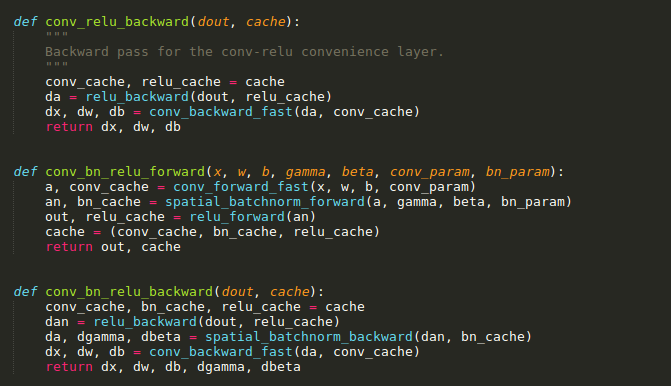
池化汇聚层

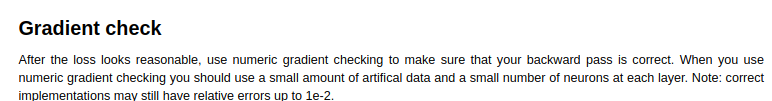


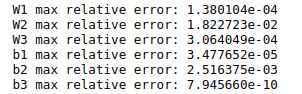


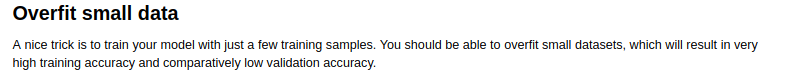


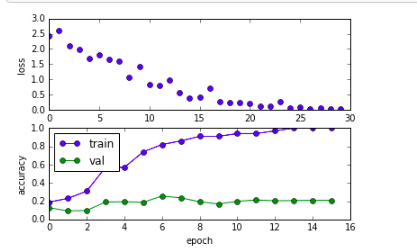


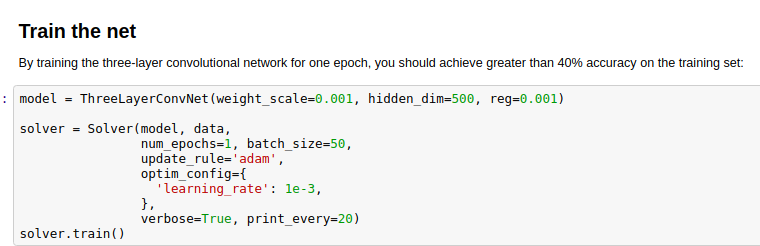


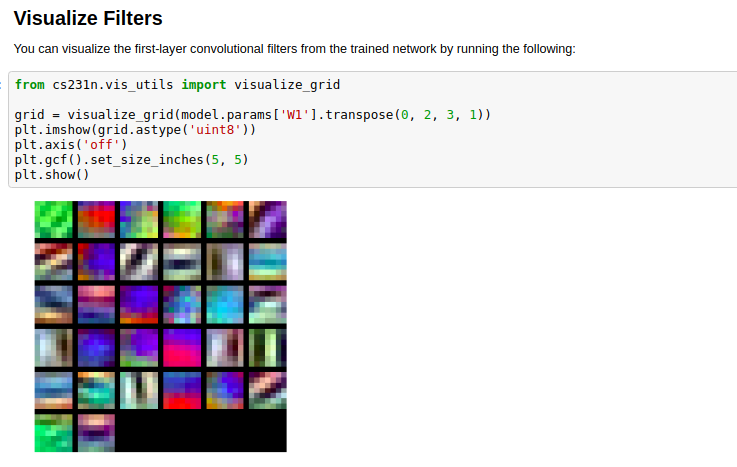


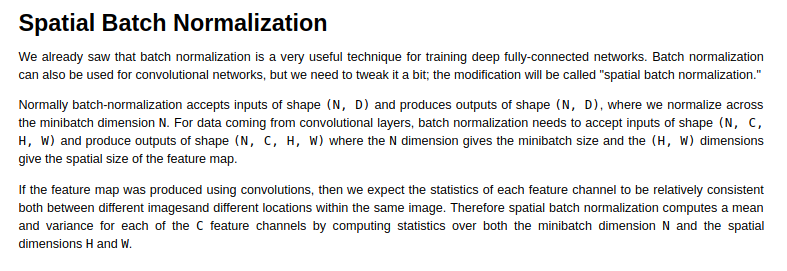




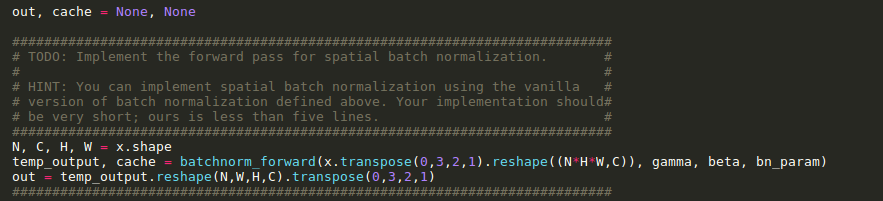


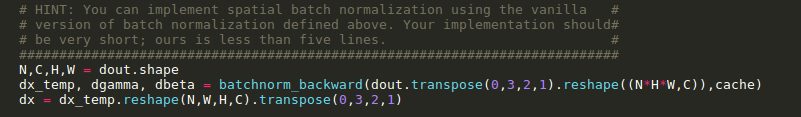


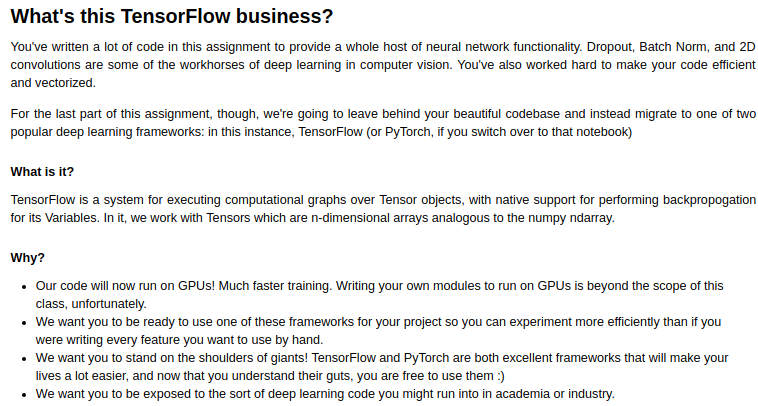


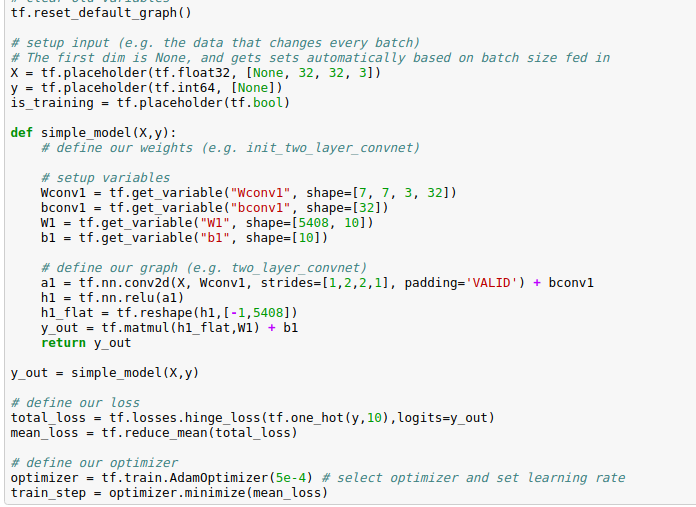


空间批量归一化

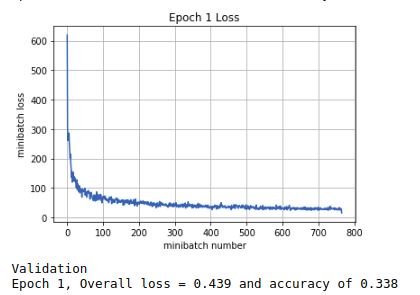


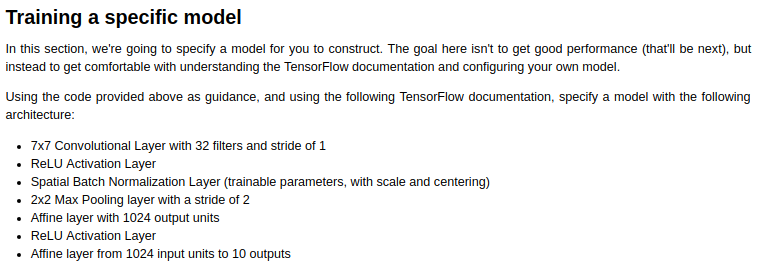
Q5: PyTorch / TensorFlow on CIFAR-10

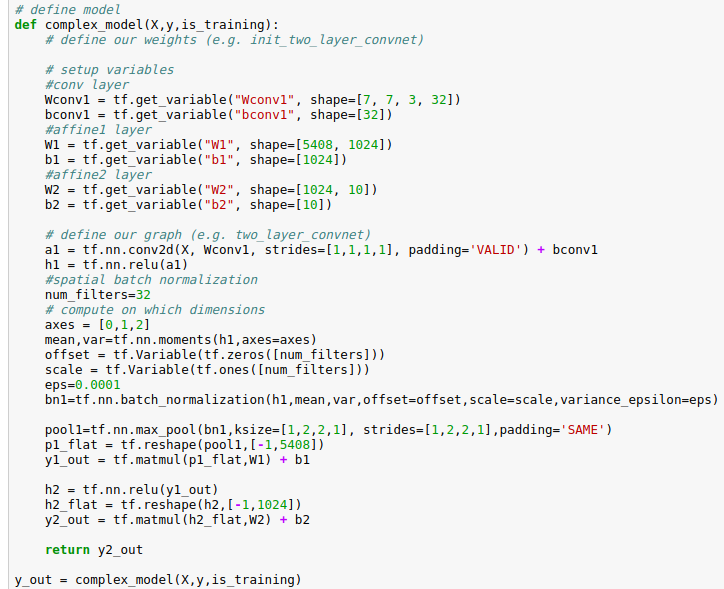


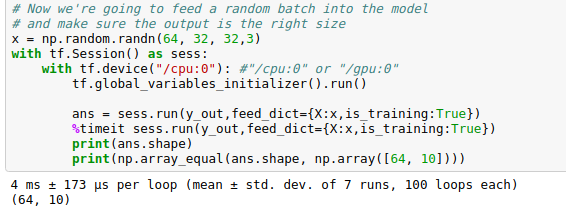


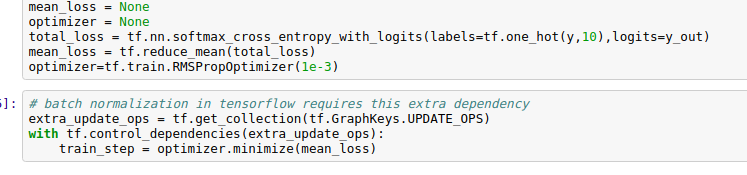


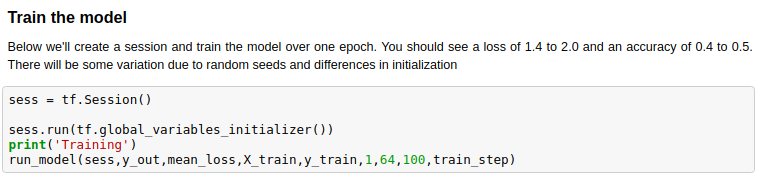


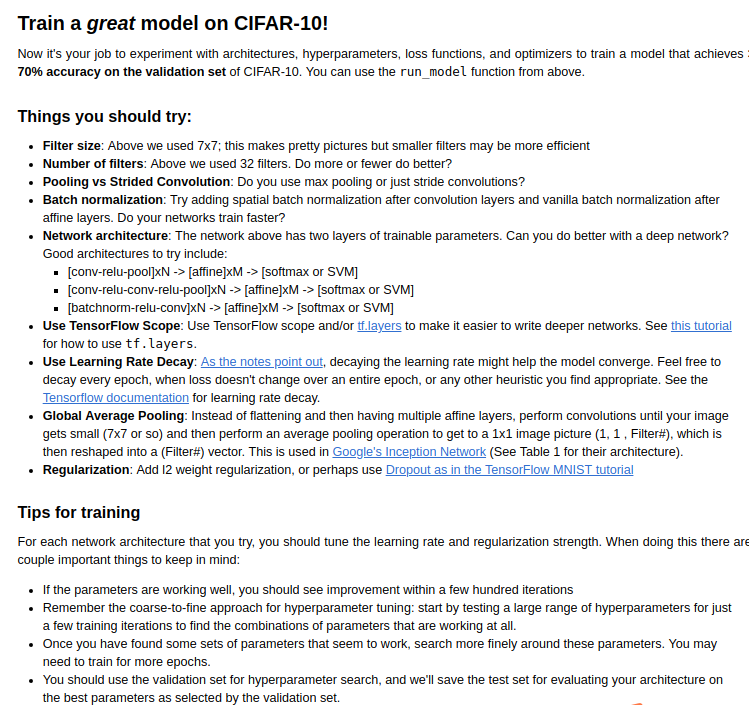


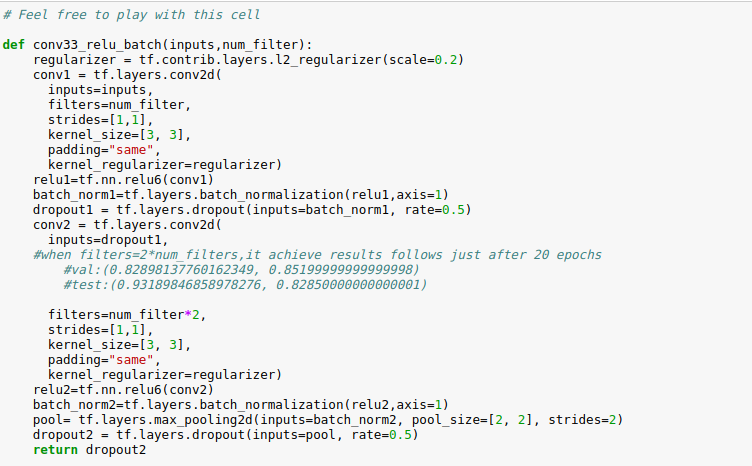


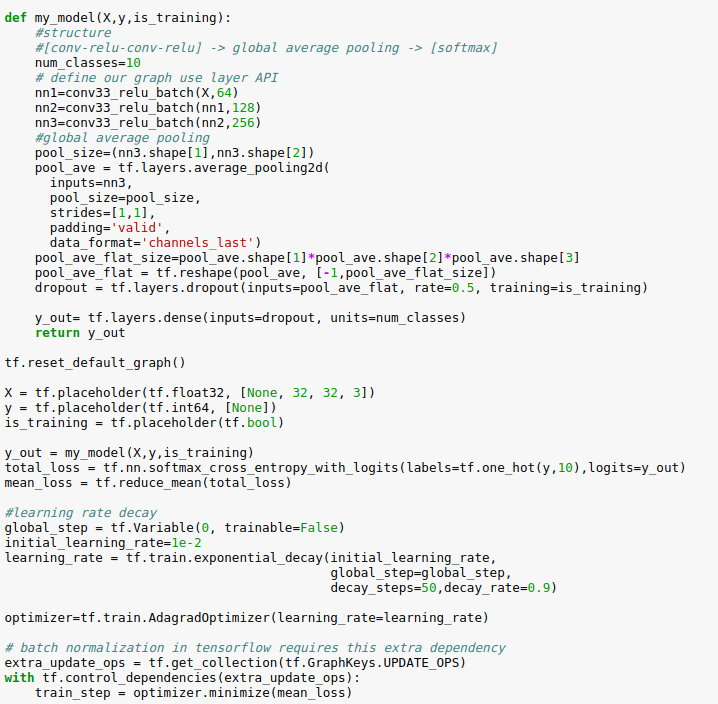


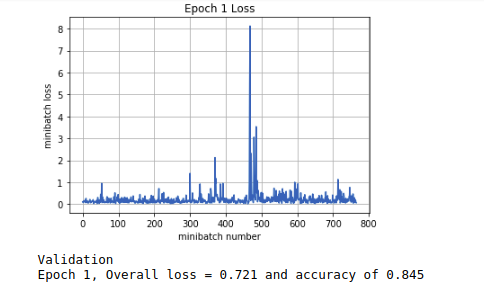


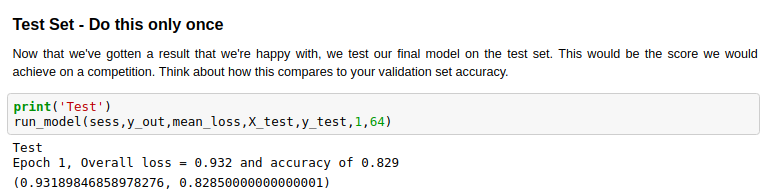












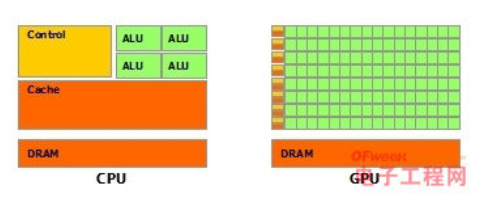
# 8. 其他概念与函数

## 8.1 CPU vs GPU

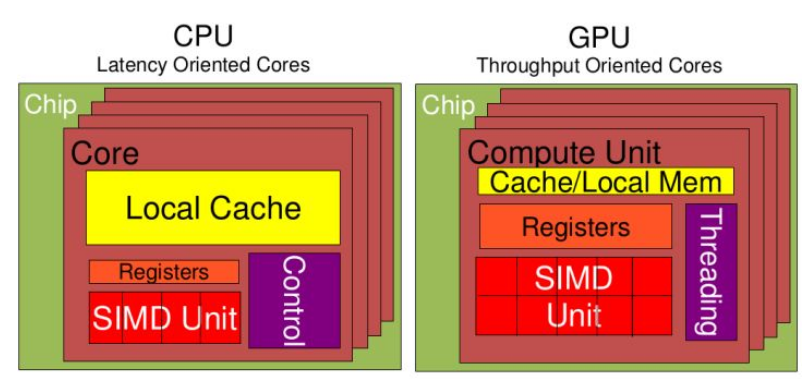
CPU即中央处理器，GPU即图形处理器.

CPU和GPU之所以大不相同，是由于其设计目标的不同，它们分别针对了两种不同的应用场景。CPU需要很强的通用性来处理各种不同的数据类型，同时又要逻辑判断又会引入大量的分支跳转和中断的处理。这些都使得CPU的内部结构异常复杂。而GPU面对的则是类型高度统一的、相互无依赖的大规模数据和不需要被打断的纯净的计算环境。

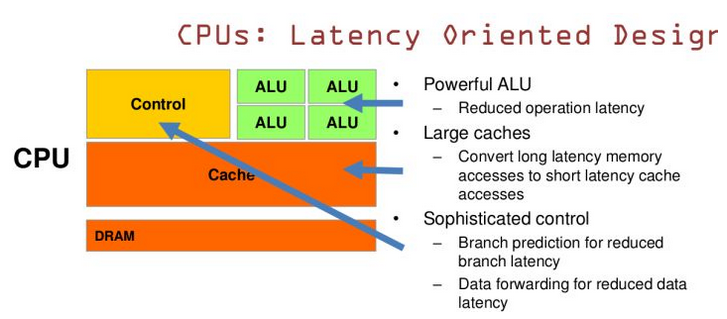
　于是CPU和GPU就呈现出非常不同的架构（示意图）：



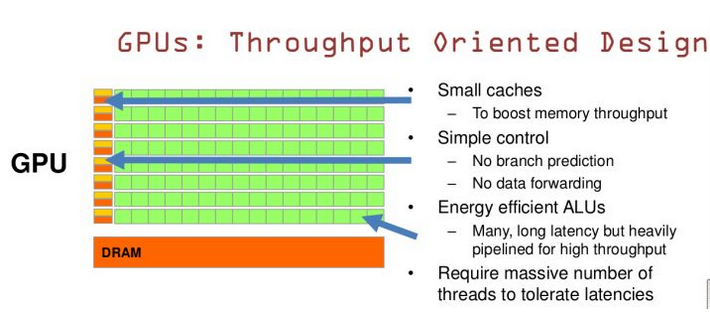
其中绿色的是计算单元，橙红色的是存储单元，橙黄色的是控制单元。GPU采用了数量众多的计算单元和超长的流水线，但只有非常简单的控制逻辑并省去了Cache。而CPU不仅被Cache占据了大量空间，而且还有有复杂的控制逻辑和诸多优化电路，相比之下计算能力只是CPU很小的一部分



从上图可以看出：Cache, local memory： CPU > GPU Threads(线程数): GPU > CPURegisters: GPU > CPU 多寄存器可以支持非常多的Thread,thread需要用到register,thread数目大，register也必须得跟着很大才行。SIMD Unit(单指令多数据流,以同步方式，在同一时间内执行同一条指令): GPU > CPU。 CPU 基于低延时的设计：



CPU有强大的ALU（算术运算单元）,它可以在很少的时钟周期内完成算术计算。当今的CPU可以达到64bit 双精度。执行双精度浮点源算的加法和乘法只需要1～3个时钟周期。CPU的时钟周期的频率是非常高的，达到1.532～3gigahertz(千兆HZ, 10的9次方).大的缓存也可以降低延时。保存很多的数据放在缓存里面，当需要访问的这些数据，只要在之前访问过的，如今直接在缓存里面取即可。复杂的逻辑控制单元。当程序含有多个分支的时候，它通过提供分支预测的能力来降低延时。数据转发。 当一些指令依赖前面的指令结果时，数据转发的逻辑控制单元决定这些指令在pipeline中的位置并且尽可能快的转发一个指令的结果给后续的指令。这些动作需要很多的对比电路单元和转发电路单元。



GPU的特点是有很多的ALU和很少的cache. 缓存的目的不是保存后面需要访问的数据的，这点和CPU不同，而是为thread提高服务的。如果有很多线程需要访问同一个相同的数据，缓存会合并这些访问，然后再去访问dram（因为需要访问的数据保存在dram中而不是cache里面），获取数据后cache会转发这个数据给对应的线程，这个时候是数据转发的角色。但是由于需要访问dram，自然会带来延时的问题。

所以与CPU擅长逻辑控制，串行的运算。和通用类型数据运算不同，GPU擅长的是大规模并发计算，这也正是密码破解等所需要的。所以GPU除了图像处理，也越来越多的参与到计算当中来。

## 8.2 numpy.std()

计算矩阵标准差



## 8.3 np.random.normal()正态分布



参数的意义为：

loc:float

概率分布的均值，对应着整个分布的中心center

scale:float

概率分布的标准差，对应于分布的宽度，scale越大越矮胖，scale越小，越瘦高

size:int or tuple of ints

输出的shape，默认为None，只输出一个值

## 8.4 Python str() 函数



## 8.5 numpy.logspace用法

logspac用于创建等比数列

