[**深度学习与计算机视觉系列(7)\_神经网络数据预处理，正则化与损失函数**](https://blog.csdn.net/yaoqiang2011/article/details/50451460)

作者：[寒小阳](http://blog.csdn.net/han_xiaoyang?viewmode=contents)   
时间：2016年1月。   
出处：<http://blog.csdn.net/han_xiaoyang/article/details/50451460>   
声明：版权所有，转载请联系作者并注明出处

## 1. 引言

上一节我们讲完了各种激励函数的优缺点和选择，以及网络的大小以及正则化对神经网络的影响。这一节我们讲一讲输入数据预处理、正则化以及损失函数设定的一些事情。

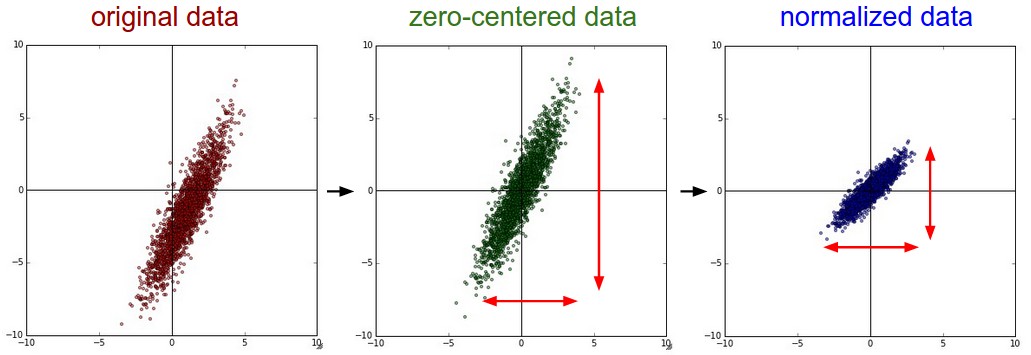
## 2. 数据与网络的设定

前一节提到前向计算涉及到的组件(主要是神经元)设定。神经网络结构和参数设定完毕之后，我们就得到得分函数/score function(忘记的同学们可以翻看一下之前的[博文](http://blog.csdn.net/han_xiaoyang/article/details/49999583))，总体说来，一个完整的神经网络就是在不断地进行线性映射(权重和input的内积)和非线性映射(部分激励函数作用)的过程。这一节我们会展开来讲讲**数据预处理**，**权重初始化**和**损失函数**的事情。

## 2.1 数据预处理

在卷积神经网处理图像问题的时候，图像数据有3种常见的预处理可能会用到，如下。我们假定数据表示成矩阵为X，其中我们假定X是[N\*D]维矩阵(N是样本数据量，D为单张图片的数据向量长度)。

* **去均值**，这是最常见的图片数据预处理，简单说来，它做的事情就是，对待训练的每一张图片的特征，都减去全部训练集图片的特征均值，这么做的直观意义就是，我们把输入数据各个维度的数据都中心化到0了。使用python的numpy工具包，这一步可以用X -= np.mean(X, axis = 0)轻松实现。当然，其实这里也有不同的做法：简单一点，我们可以直接求出所有像素的均值，然后每个像素点都减掉这个相同的值；稍微优化一下，我们可以在RGB三个颜色通道分别做这件事。
* **归一化**，归一化的直观理解含义是，我们做一些工作去保证所有的维度上数据都在一个变化幅度上。通常我们有两种方法来实现归一化。一个是在数据都去均值之后，每个维度上的数据都除以这个维度上数据的标准差(X /= np.std(X, axis = 0))。另外一种方式是我们除以数据绝对值最大值，以保证所有的数据归一化后都在-1到1之间。多说一句，其实在任何你觉得各维度幅度变化非常大的数据集上，你都可以考虑归一化处理。不过对于图像而言，其实这一步反倒可做可不做，因为大家都知道，像素的值变化区间都在[0,255]之间，所以其实图像输入数据天生幅度就是一致的。

上述两个操作对于数据的作用，画成示意图，如下：   


* **PCA和白化/whitening**，这是另外一种形式的数据预处理。在经过去均值操作之后，我们可以计算数据的协方差矩阵，从而可以知道数据各个维度之间的相关性，简单示例代码如下：

# 假定输入数据矩阵X是[N\*D]维的

X -= np.mean(X, axis = 0) # 去均值

cov = np.dot(X.T, X) / X.shape[0] # 计算协方差

* 1
* 2
* 3

得到的结果矩阵中元素(i,j)表示原始数据中，第i维和第j维之间的相关性。有意思的是，其实协方差矩阵的对角线包含了每个维度的变化幅度。另外，我们都知道协方差矩阵是对称的，我们可以在其上做矩阵奇异值分解(SVD factorization)：

U,S,V = np.linalg.svd(cov)

* 1

其中U为特征向量，我们如果相对原始数据(去均值之后)做去相关操作，只需要进行如下运算：

Xrot = np.dot(X, U)

* 1

这么理解一下可能更好，U是一组正交基向量。所以我们可以看做把原始数据X投射到这组维度保持不变的正交基底上，从而也就完成了对原始数据的去相关。如果去相关之后你再求一下Xrot的协方差矩阵，你会发现这时候的协方差矩阵是一个对角矩阵了。而numpy中的np.linalg.svd更好的一个特性是，它返回的U是对特征值排序过的，这也就意味着，我们可以用它进行降维操作。我们可以只取top的一些特征向量，然后做和原始数据做矩阵乘法，这个时候既降维减少了计算量，同时又保存下了绝大多数的原始数据信息，这就是所谓的**主成分分析/PCA**：

Xrot\_reduced = np.dot(X, U[:,:100])

* 1

这个操作之后，我们把原始数据集矩阵从[N\*D]降维到[N\*100]，保存了前100个能包含绝大多数数据信息的维度。实际应用中，你在PCA降维之后的数据集上，做各种机器学习的训练，在节省空间和时间的前提下，依旧能有很好的训练准确度。

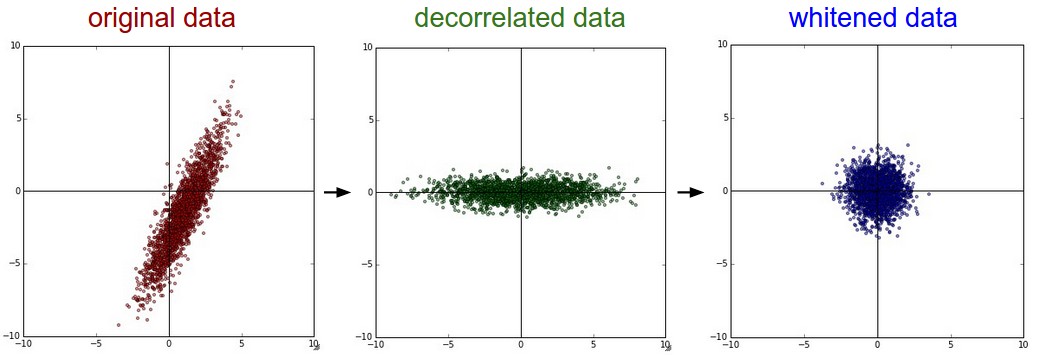
最后我们再提一下**whitening**操作。所谓whitening，就是把各个特征轴上的数据除以对应特征值，从而达到在每个特征轴上都归一化幅度的结果。whitening变换的几何意义和理解是，如果输入的数据是多变量高斯，那whitening之后的 数据是一个均值为0而不同方差的高斯矩阵。这一步简单代码实现如下：

#白化数据

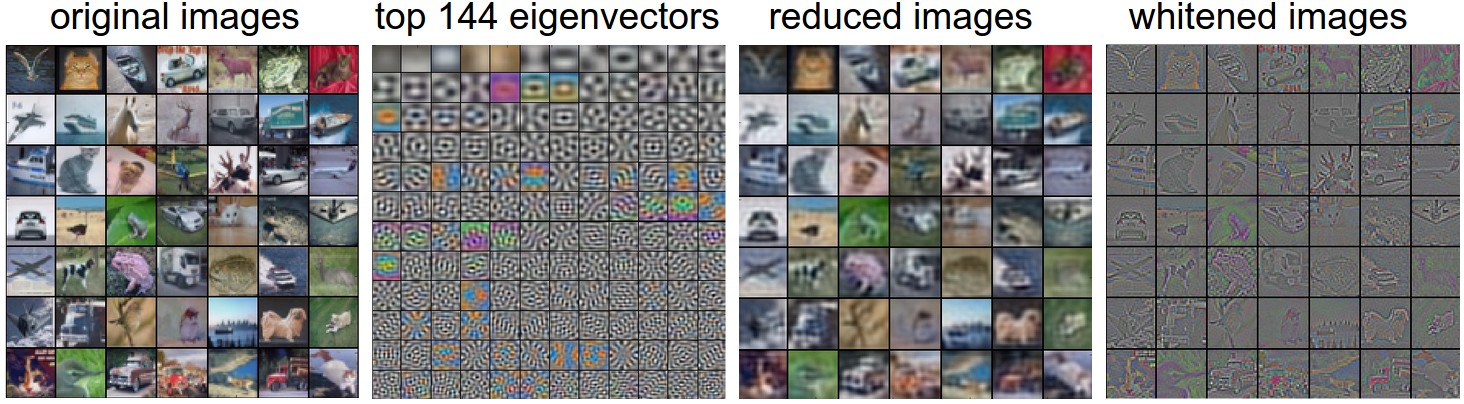
Xwhite = Xrot / np.sqrt(S + 1e-5)

* 1
* 2

提个醒：whitening操作会有严重化噪声的可能。注意到我们在上述代码中，分母的部分加入了一个很小的数1e-5，以防止出现除以0的情况。但是数据中的噪声部分可能会因whitening操作而变大，因为这个操作的本质是把输入的每个维度都拉到差不多的幅度，那么本不相关的有微弱幅度变化的噪声维度，也被拉到了和其他维度同样的幅度。当然，我们适当提高分母中的安全因子(1e-5)可以在一定程度上缓解这个问题。

下图为原始数据到**去相关**到**白化**之后的数据分布示意图：   


我们来看看真实数据集上的操作与得到的结果，也许能对这些过程有更清晰一些的认识。大家都还记得CIFAR-10图像数据集吧。训练集大小为50000\*3072，也就是说，每张图片都被展成一个3072维度的列向量了。然后我们对原始50000\*3072数据矩阵做SVD分解，进行上述一些操作，再可视化一下，得到的结果示意图如下：



我们稍加解释一下，最左边是49张原始图片；左起第2幅图是最3072个特征向量中最top的144个，这144个特征向量包含了绝大多数数据变量信息，而其实它们代表的是图片中低频的信息；左起第3幅图表示PCA降维操作之后的49张图片，使用上面求得的144个特征向量。我们可以观察到图片好像被蒙上了一层东西一样，模糊化了，这也就表明了我们的top144个特征向量捕捉到的都是图像的低频信息，不过我们发现图像的绝大多数信息确实被保留下来了；最右图是whitening的144个数通过乘以U.transpose()[:144,:]还原回图片的样子，有趣的是，我们发现，现在低频信息基本都被滤掉了，剩下一些高频信息被放大呈现。

**实际工程中**，因为这个部分讲到数据预处理，我们就把基本的几种数据预处理都讲了一遍，但实际卷积神经网中，我们并没有用到去相关和whitening操作。当然，去均值是非常非常重要的，而每个像素维度的归一化也是常用的操作。

**特别说明**，需要特别说明的一点是，上述的预处理操作，**一定都是在训练集上先预算的，然后应用在交叉验证/测试集上的**。举个例子，有些同学会先把所有的图片放一起，求均值，然后减掉均值，再把这份数据分作训练集和测试集，这是不对的亲！！！

### 2.2 权重初始化

我们之前已经看过一个完整的神经网络，是怎么样通过神经元和连接搭建起来的，以及如何对数据做预处理。在训练神经网络之前，我们还有一个任务要做，那就是初始化参数。

**错误的想法：全部初始化为0**，有些同学说，那既然要训练和收敛嘛，初始值就随便设定，简单一点就全设为0好了。亲，这样是绝对不行的！！！为啥呢？我们在神经网络训练完成之前，是不可能预知神经网络最后的权重具体结果的，但是根据我们归一化后的数据，我们可以假定，大概有半数左右的权重是正数，而另外的半数是负数。但设定全部初始权重都为0的结果是，网络中每个神经元都计算出一样的结果，然后在反向传播中有一样的梯度结果，因此迭代之后的变化情况也都一样，这意味着这个神经网络的权重没有办法差异化，也就没有办法学习到东西。

**很小的随机数**，其实我们依旧希望初始的权重是较小的数，趋于0，但是就像我们刚刚讨论过的一样，不要真的是0。综合上述想法，在实际场景中，我们通常会把初始权重设定为非常小的数字，然后正负尽量一半一半。这样，初始的时候权重都是不一样的很小随机数，然后迭代过程中不会再出现迭代一致的情况。举个例子，我们可能可以这样初始化一个权重矩阵W=0.0001\*np.random.randn(D,H)。这个初始化的过程，使得每个神经元的权重向量初始化为多维高斯中的随机采样向量，所以神经元的初始权重值指向空间中的随机方向。

**特别说明**：其实不一定更小的初始值会比大值有更好的效果。我们这么想，一个有着非常小的权重的神经网络在后向传播过程中，回传的梯度也是非常小的。这样回传的”信号”流会相对也较弱，对于层数非常多的深度神经网络，这也是一个问题，回传到最前的迭代梯度已经很小了。

**方差归一化**，上面提到的建议有一个小问题，对于随机初始化的神经元参数下的输出，其分布的方差随着输入的数量，会增长。我们实际上可以通过除以总输入数目的平方根，归一化每个神经元的输出方差到1。也就是说，我们倾向于初始化神经元的权重向量为w = np.random.randn(n) / sqrt(n)，其中n为输入数。

我们从数学的角度，简单解释一下，为什么上述操作可以归一化方差。考虑在激励函数之前的权重w与输入x的内积*s*=∑*niwixi*

部分，我们计算一下*s*

的方差：

Var(*s*)=Var(∑*inwixi*)=∑*in*Var(*wixi*)=∑*in*[*E*(*wi*)]2Var(*xi*)+*E*[(*xi*)]2Var(*wi*)+Var(*xi*)Var(*wi*)=∑*in*Var(*xi*)Var(*wi*)=(*n*Var(*w*))Var(*x*)

注意，这个推导的前2步用到了方差的性质。第3步我们假定输入均值为0，因此*E*[*xi*]=*E*[*wi*]=0

。不过这是我们的一个假设，实际情况下并不一定是这样的，比如ReLU单元的均值就是正的。最后一步我们假定*wi*,*xi*是独立分布。我们想让s的方差和输入x的方差一致，因此我们想让w的方差取值为1/n，又因为我们有公式Var(*aX*)=*a*2Var(*X*)，所以a应该取值为*a*=1/*n*−−−√

，numpy里的实现为w = np.random.randn(n) / sqrt(n)。

对于初始化权重还有一些类似的研究和建议，比如说Glorot在论文[Understanding the difficulty of training deep feedforward neural networks](http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf)就推荐使用能满足Var(*w*)=2/(*nin*+*nout*)

的权重初始化。其中*nin*,*nout*

是前一层和后一层的神经元个数。而另外一篇比较新的论文[Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](http://arxiv.org/abs/1502.01852)，则指出尤其对于ReLU神经元，我们初始化方差应该为2.0/n，也**就是w = np.random.randn(n) \* sqrt(2.0/n)，**目前的神经网络中使用了很多ReLU单元，因此这个设定其实在实际应用中使用最多。

**偏移量/bias初始化**：相对而言，bias项初始化就简单一些。我们很多时候简单起见，直接就把它们都设为0.在ReLU单元中，有些同学会使用很小的数字(比如0.01)来代替0作为所有bias项的初始值，他们解释说这样也能保证ReLU单元一开始就是被激活的，因此反向传播过程中不会终止掉回传的梯度。不过似乎实际的实验过程中，这个优化并不是每次都能起到作用的，因此很多时候我们还是直接把bias项都初始化为0。

### 2.3 正则化

在前一节里我们说了我们要通过正则化来控制神经网络，使得它不那么容易过拟合。有几种正则化的类型供选择：

* **L2正则化**，这个我们之前就提到过，非常常见。实现起来也很简单，我们在损失函数里，加入对每个参数的惩罚度。也就是说，对于每个权重*w*

，我们在损失函数里加入一项12*λw*2，其中*λ*是我们可调整的正则化强度。顺便说一句，这里在前面加上1/2的原因是，求导/梯度的时候，刚好变成*λw*而不是2*λw*

 。L2正则化理解起来也很简单，它对于特别大的权重有很高的惩罚度，以求让权重的分配均匀一些，而不是集中在某一小部分的维度上。我们再想想，加入L2正则化项，其实意味着，在梯度下降参数更新的时候，每个权重以W += -lambda\*W的程度被拉向0。

 **L1正则化**，这也是一种很常见的正则化形式。在L1正则化中，我们对于每个权重*w*

的惩罚项为*λ*|*w*|。有时候，你甚至可以看到大神们混着L1和L2正则化用，也就是说加入惩罚项*λ*1∣*w*∣+*λ*2*w*2

 ，L1正则化有其独特的特性，它会让模型训练过程中，权重特征向量逐渐地稀疏化，这意味着到最后，我们只留下了对结果影响最大的一部分权重，而其他不相关的输入(例如『噪声』)因为得不到权重被抑制。**所以通常L2正则化后的特征向量是一组很分散的小值，而L1正则化只留下影响较大的权重。在实际应用中，如果你不是特别要求只保留部分特征，那么L2正则化通常能得到比L1正则化更好的效果**

 **最大范数约束**，另外一种正则化叫做最大范数约束，它直接限制了一个上行的权重边界，然后约束每个神经元上的权重都要满足这个约束。实际应用中是这样实现的，我们不添加任何的惩罚项，就按照正常的损失函数计算，只不过在得到每个神经元的权重向量*w*⃗

之后约束它满足∥*w*⃗ ∥2<*c*

* 。有些人提到这种正则化方式帮助他们提高最后的模型效果。另外，这种正则化方式倒是有一点很吸引人：在神经网络训练学习率设定很高的时候，它也能很好地约束住权重更新变化，不至于直接挂掉。
* **Dropout**，亲，这个是我们实际神经网络训练中，用的非常多的一种正则化手段，同时也相当有效。Srivastava等人的论文[Dropout: A Simple Way to Prevent Neural Networks from Overfitting](http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf)最早提到用dropout这种方式作为正则化手段。一句话概括它，就是：在训练过程中，我们对每个神经元，都以概率p保持它是激活状态，1-p的概率直接关闭它。

下图是一个3层的神经网络的dropout示意图：



可以这么理解，在训练过程中呢，我们对全体神经元，以概率p做了一个采样，只有选出的神经元要进行参数更新。所以最后就从左图的全连接到右图的Dropout过后神经元连接图了。需要多说一句的是，在测试阶段，我们不用dropout，而是直接从概率的角度，对权重配以一个概率值。

简单的Dropout代码如下(这是简易实现版本，但是不建议使用，我们会分析为啥，并在之后给出优化版)：

p = 0.5 # 设定dropout的概率，也就是保持一个神经元激活状态的概率

def train\_step(X):

""" X contains the data """

# 3层神经网络前向计算

H1 = np.maximum(0, np.dot(W1, X) + b1)

U1 = np.random.rand(\*H1.shape) < p # 第一次Dropout

H1 \*= U1 # drop!

H2 = np.maximum(0, np.dot(W2, H1) + b2)

U2 = np.random.rand(\*H2.shape) < p # 第二次Dropout

H2 \*= U2 # drop!

out = np.dot(W3, H2) + b3

# 反向传播: 计算梯度... (这里省略)

# 参数更新... (这里省略)

def predict(X):

# 加上Dropout之后的前向计算

H1 = np.maximum(0, np.dot(W1, X) + b1) \* p

H2 = np.maximum(0, np.dot(W2, H1) + b2) \* p

out = np.dot(W3, H2) + b3

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

上述代码中，在train\_step函数中，我们做了2次Dropout。我们甚至可以在输入层做一次dropout。反向传播过程保持不变，除了我们要考虑一下U1,U2

很重要的一点是，大家仔细看predict函数部分，我们不再dropout了，而是对于每个隐层的输出，都用概率p做了一个幅度变换。可以从数学期望的角度去理解这个做法，我们考虑一个神经元的输出为x(没有dropout的情况下)，它的输出的数学期望为*px*+(1−*p*)0

，那我们在测试阶段，如果直接把每个输出x都做变换*x*→*px*

，其实是可以保持一样的数学期望的。

上述代码的写法有一些缺陷，我们必须在测试阶段对每个神经的输出都以p的概率输出。考虑到实际应用中，测试阶段对于时间的要求非常高，我们可以考虑反着来，代码实现的时候用inverted dropout，即在训练阶段就做相反的幅度变换/scaling(除以p)，这样在测试阶段，我们可以直接把权重拿来使用，而不用附加很多步用p做scaling的过程。inverted dropout的示例代码如下：

"""

Inverted Dropout的版本，把本该花在测试阶段的时间，转移到训练阶段，从而提高testing部分的速度

"""

p = 0.5 # dropout的概率，也就是保持一个神经元激活状态的概率

def train\_step(X):

# f3层神经网络前向计算

H1 = np.maximum(0, np.dot(W1, X) + b1)

U1 = (np.random.rand(\*H1.shape) < p) / p # 注意到这个dropout中我们除以p，做了一个inverted dropout

H1 \*= U1 # drop!

H2 = np.maximum(0, np.dot(W2, H1) + b2)

U2 = (np.random.rand(\*H2.shape) < p) / p # 这个dropout中我们除以p，做了一个inverted dropout

H2 \*= U2 # drop!

out = np.dot(W3, H2) + b3

# 反向传播: 计算梯度... (这里省略)

# 参数更新... (这里省略)

def predict(X):

# 直接前向计算，无需再乘以p

H1 = np.maximum(0, np.dot(W1, X) + b1)

H2 = np.maximum(0, np.dot(W2, H1) + b2)

out = np.dot(W3, H2) + b3

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

对于dropout这个部分如果你有更深的兴趣，欢迎阅读以下文献：   
1) 2014 Srivastava 的论文[Dropout paper](http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf)   
2) [Dropout Training as Adaptive Regularization](http://papers.nips.cc/paper/4882-dropout-training-as-adaptive-regularization.pdf)

* **bias项的正则化**，其实我们在[之前的博客](http://blog.csdn.net/han_xiaoyang/article/details/49999583)中提到过，我们大部分时候并不对偏移量项做正则化，因为它们也没有和数据直接有乘法等交互，也就自然不会影响到最后结果中某个数据维度的作用。不过如果你愿意对它做正则化，倒也不会影响最后结果，毕竟总共有那么多权重项，才那么些bias项，所以一般也不会影响结果。

**实际应用中**：我们最常见到的是，在全部的交叉验证集上使用L2正则化，同时我们在每一层之后用dropout，很常见的dropout概率为p=0.5，你也可以通过交叉验证去调整这个值。

### 2.4 损失函数

刚才讨论了数据预处理、权重初始化与正则化相关的问题。现在我们回到训练需要的关键之一：损失函数。对于这么复杂的神经网络，我们也得有一个评估准则去评估预测值和真实结果之间的吻合度，也就是损失函数。神经网络里的损失函数，实际上是计算出了每个样本上的loss，再求平均之后的一个形式，即*L*=1*N*∑*iLi*

，其中N是训练样本数。

#### 2.4.1 分类问题

* **分类问题**是到目前为止我们一直在讨论的。我们假定一个数据集中每个样本都有唯一一个正确的标签/类别。我们之前提到过有两种损失函数可以使用，其一是SVM的hinge loss:

*Li*=∑*j*≠*yi*max(0,*fj*−*fyi*+1)

另外一个是Softmax分类器中用到的互熵损失:

*Li*=−log(*efyi*∑*jefj*)

* **问题：特别多的类别数**。当类别标签特别特别多的时候(比如ImageNet包含22000个类别)，[层次化的Softmax](http://arxiv.org/pdf/1310.4546.pdf)，它将类别标签建成了一棵树，这样任何一个类别，其实就对应tree的一条路径，然后我们在每个树的结点上都训练一个Softmax以区分是左分支还是右分支。
* **属性分类**，上述的两种损失函数都假定，对于每个样本，我们只有一个正确的答案*yi*

。但是在有些场景下，*yi*

* 是一个二值的向量，每个元素都代表有没有某个属性，这时候我们怎么办呢？举个例子说，Instagram上的图片可以看作一大堆hashtag里的一个tag子集，所有一张图片可以有多个tag。对于这种情况，大家可能会想到一个最简单的处理方法，就是对每个属性值都建一个二分类的分类器。比如，对应某个类别的二分类器可能有如下形式的损失函数：

*Li*=∑*j*max(0,1−*yijfj*)

其中的求和是针对有所的类别j，而*yij*

是1或者-1(取决于第i个样本是否有第j个属性的标签)，打分向量*fj*

在类别/标签被预测到的情况下为正，其他情况为负。注意到如果正样本有比+1小的得分，或者负样本有比-1大的得分，那么损失/loss就一直在累积。

另外一个也许有效的解决办法是，我们可以对每个属性，都单独训练一个逻辑回归分类器，一个二分类的逻辑回归分类器只有0，1两个类别，属于1的概率为：

*P*(*y*=1∣*x*;*w*,*b*)=11+*e*−(*wTx*+*b*)=*σ*(*wTx*+*b*)

又因为0，1两类的概率和为1，所以归属于类别0的概率为*P*(*y*=0∣*x*;*w*,*b*)=1−*P*(*y*=1∣*x*;*w*,*b*)

。一个样本在*σ*(*wTx*+*b*)>0.5的情况下被判定为1，对应sigmoid函数化简一下，对应的是得分*wTx*+*b*>0

。这时候的损失函数可以定义为最大化似然概率的形式，也就是：

*Li*=∑*jyij*log(*σ*(*fj*))+(1−*yij*)log(1−*σ*(*fj*))

其中标签*yij*

为1(正样本)或者0(负样本)，而*δ*

是sigmoid函数。

#### 2.4.2 回归问题

回归是另外一类机器学习问题，主要用于预测连续值属性，比如房子的价格或者图像中某些东西的长度等。对于回归问题，我们一般计算预测值和实际值之间的差值，然后再求L2范数或者L1范数用于衡量。其中对一个样本(一张图片)计算的L2范数损失为：

*Li*=∥*f*−*yi*∥22

而L1范数损失函数是如下的形式：

*Li*=∥*f*−*yi*∥1=∑*j*∣*fj*−(*yi*)*j*∣

**注意**：

* **回归问题中用到的L2范数损失，比分类问题中的Softmax分类器用到的损失函数，更难优化。**直观想一想这个问题，一个神经网络最后输出离散的判定类别，比训练它去输出一个个和样本结果对应的连续值，要简单多了。
* 我们前面的[博文](http://blog.csdn.net/han_xiaoyang/article/details/49999583)中提到过，其实Softmax这种分类器，对于输出的打分结果具体值是不怎么在乎的，它只在乎各个类别之间的打分幅度有没有差很多(比如二分类两个类别的得分是1和9，与0.1和0.9)。
* 再一个，L2范数损失健壮性更差一些，异常点和噪声都可能改变损失函数的幅度，而带来大的梯度偏差。
* 一般情况下，**对于回归问题，我们都会首先考虑，这个问题能否转化成对应的分类问题**，比如说我们把输出值划分成不同的区域(切成一些桶)。举个例子，如果我们要预测一部电影的豆瓣打分，我们可以考虑把得分结果分成1-5颗星，而转化成一个分类问题。
* 如果你觉得问题确实没办法转化成分类问题，那要小心使用L2范数损失：举个例子，**在神经网络中，在L2损失函数之前使用dropout是不合适的。**

如果我们遇到回归问题，首先要想想，是否完全没有可能把结果离散化之后，把这个问题转化成一个分类问题。

## 3. 总结

总结一下：

* 在很多神经网络的问题中，我们都建议**对数据特征做预处理，去均值，然后归一化到[-1,1]之间。**
* 从一个标准差为2/*n*−−−√
* 的高斯分布中初始化权重，其中n为输入的个数。
* **使用L2正则化(或者最大范数约束)和dropout来减少神经网络的过拟合。**
* 对于分类问题，我们最常见的损失函数依旧是SVM hinge loss和Softmax互熵损失。

## 参考资料与原文

[cs231n 神经网络数据预处理正则化与损失函数](http://cs231n.github.io/neural-networks-2/)